С. Макконнелл


# СОВЕРШЕННЫЙ КОД

## ПРАКТИЧЕСКОЕ РУКОВОДСТВО ПО РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

РУССКАЯ РЕДАКЦИЯ

Microsoft

«Великолепное руководство по стилю программирования и конструированию ПО».

*Мартин Фаулер, автор книги «Refactoring»*

«Книга Стива Макконнелла... это быстрый путь к мудрому программированию... Его книги увлекательны, и вы никогда не забудете то, что он рассказывает, опираясь на свой с трудом полученный опыт».

*Джон Бентли, автор книги «Programming Pearls, 2d ed»*

«Это просто самая лучшая книга по конструированию ПО из всех, что когда-либо попадались мне в руки. Каждый разработчик должен иметь ее и перечитывать от корки до корки каждый год. Я ежегодно перечитываю ее на протяжении вот уже девяти лет и все еще узнаю много нового!»

*Джон Роббинс, автор книги «Debugging Applications for Microsoft .NET and Microsoft Windows»*

«Современное ПО *должно* быть надежным и гибким, а создание защищенного кода начинается с дисциплинированного конструирования программы. За десять лет так и не появилось лучшего руководства по этой теме, чем эта книга.

*Майкл Ховард, специалист по защите ПО, корпорация Microsoft; один из авторов книги «Writing Secure Code»*

«Это исчерпывающее исследование тактических аспектов создания хорошо спроектированных программ. Книга Макконнелла охватывает такие разные темы, как архитектура, стандарты кодирования, тестирование, интеграция и суть разработки ПО».

*Гради Буч, автор книги «Object Solutions»*

«Авторитетная энциклопедия для разработчиков ПО — вот что такое „Совершенный код“. Подзаголовок „Практическое руководство по конструированию ПО“ характеризует эту 850-страничную книгу абсолютно точно. Как утверждает автор, она призвана сократить разрыв между знаниями „гуру и лучших специалистов отрасли“ (например, Йордона и Прессмана) и общепринятыми методиками разработки коммерческого ПО, а также „помочь создавать более качественные программы за меньшее время с меньшей головной болью“... Эту книгу следует иметь каждому разработчику. Ее стиль и содержание в высшей степени практичны».

*Крис Лузли, автор книги «High-Performance Client/Server»*

«Полная плодотворных идей книга Макконнелла „Совершенный код“ — это одна из самых понятных работ, посвященных подробному обсуждению методик разработки ПО...»

*Эрик Бетке, автор книги «Game Development and Production»*

«Кладезь полезной информации и рекомендаций по общим вопросам проектирования и разработки хорошего ПО».

*Джон Демпстер, автор книги «The Laboratory Computer: A Practical Guide for Physiologists and Neuroscientists»*

«Если вы действительно хотите улучшить навыки программирования, обязательно прочтите книгу „Совершенный код“ Стива Макконнелла».

*Джин Дж. Лаброссе, автор книги «Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C»*

«Стив Макконнелл написал одну из лучших книг по разработке ПО, не привязанных к конкретной среде...»

*Кеннет Розен, один из авторов книги «Unix: The Complete Reference»*

«Пару раз в поколение или около того появляются книги, обобщающие накопленный опыт и избавляющие вас от многих лет мучений... Не могу найти слов, чтобы адекватно описать все великолепие этой книги. „Совершенный код“ — довольно жалкое название для такой превосходной работы».

*Джефф Дантеманн, журнал «PC Techniques»*

«Издательство Microsoft Press опубликовало то, что я считаю самой лучшей книгой по конструированию ПО. Эта книга должна занять место на книжной полке каждого программиста».

*Уоррен Кеуффель, журнал «Software Development»*

«Эту выдающуюся книгу следует прочесть каждому программисту».

*Т. Л. (Фрэнк) Паппас, журнал «Computer»*

«Если вы собираетесь стать профессиональным программистом, покупка этой книги, пожалуй, станет самым мудрым вложением средств. Можете не читать этот обзор дальше — просто идите в магазин и купите ее. Как пишет сам Макконнелл, его целью было сокращение разрыва между знаниями гуру и общепринятыми методиками разработки коммерческого ПО... Удивительно, но ему это удалось».

*Ричард Матеосян, журнал «IEEE Micro»*

«„Совершенный код“ — обязательное чтение для всех... кто имеет отношение к разработке ПО».

*Томми Ашер, журнал «C Users Journal»*

«Я вынужден сделать чуть более категоричное заявление, чем обычно, и рекомендовать книгу Стива Макконнелла „Совершенный код“ всем разработчикам без всяких оговорок... Если раньше во время работы я держал ближе всего к клавиатуре руководства по API, то теперь их место заняла книга Макконнелла».

*Джим Кайл, журнал «Windows Tech Journal»*

«Это лучшая книга по разработке ПО из всех, что я читал».

*Эдвард Кенворт, журнал «EXE»*

«Эта книга заслуживает статуса классической, и ее в обязательном порядке должны прочесть все разработчики и те, кто ими управляет».

*Питер Райт, «Program Now»*

*Посвящаю эту книгу своей жене Эшли, которая не имеет особого отношения к программированию, но настолько обогащает всю мою остальную жизнь, что я не могу выразить это в словах.*

**Steve McConnell**

# CODE COMPLETE

Second Edition

Microsoft® Press

Стив Макконнелл

# Совершенный КОД

## МАСТЕР-КЛАСС

РУССКАЯ РЕДАКЦИЯ

2010

**УДК 004.45**
**ББК 32.973.26–018.2**
**М15**

**Макконнелл С.**

М15 Совершенный код. Мастер-класс / Пер. с англ. — М. : Издательство «Русская редакция», 2010. — 896 стр. : ил.

**ISBN 978-5-7502-0064-1**

Более 10 лет первое издание этой книги считалось одним из лучших практических руководств по программированию. Сейчас эта книга полностью обновлена с учетом современных тенденций и технологий и дополнена сотнями новых примеров, иллюстрирующих искусство и науку программирования. Опираясь на академические исследования, с одной стороны, и практический опыт коммерческих разработок ПО — с другой, автор синтезировал из самых эффективных методик и наиболее эффективных принципов ясное прагматичное руководство. Каков бы ни был ваш профессиональный уровень, с какими бы средствами разработками вы ни работали, какова бы ни была сложность вашего проекта, в этой книге вы найдете нужную информацию, она заставит вас размышлять и поможет создать совершенный код.

Книга состоит из 35 глав, предметного указателя и библиографии.

**УДК 004.45**
**ББК 32.973.26–018.2**

# Содержание

# Предисловие

*Разрыв между самыми лучшими и средними методиками разработки ПО очень широк — вероятно, шире, чем в любой другой инженерной дисциплине. Средство распространения информации о хороших методиках сыграло бы весьма важную роль.*

*Фред Брукс (Fred Brooks)*

Моей главной целью при написании этой книги было сокращение разрыва между знаниями гуру и лучших специалистов отрасли, с одной стороны, и общепринятыми методиками разработки коммерческого ПО — с другой. Многие эффективные методики программирования годами скрываются в журналах и научных работах, прежде чем становятся доступными программистской общественности.

Хотя передовые методики разработки ПО в последние годы быстро развивались, общепринятые практически стояли на месте. Многие программы все еще полны ошибок, поставляются с опозданием и не укладываются в бюджет, а многие не отвечают требованиям пользователей. Ученые обнаружили эффективные методики, устраняющие большинство проблем, которые отравляют нашу жизнь с 1970-х годов. Однако из-за того, что эти методики редко покидают страницы узкоспециализированных технических изданий, в большинстве компаний по разработке ПО они еще не используются. Установлено, что для широкого распространения исследовательских разработок обычно требуется от 5 до 15 и более лет (Raghavan and Chand, 1989; Rogers, 1995; Parnas, 1999). Данная книга призвана ускорить этот процесс и сделать важные открытия доступными средним программистам.

## Кому следует прочитать эту книгу?

Исследования и опыт программирования, отраженные в этой книге, помогут вам создавать высококачественное ПО и выполнять свою работу быстрее и эффективнее. Прочитав ее, вы поймете, почему вы сталкивались с проблемами в прошлом, и узнаете, как избегать их в будущем. Описанные мной методики программирования помогут вам сохранять контроль над крупными проектами, а также успешно сопровождать и изменять ПО при изменении требований.

### Опытные программисты

Эта книга окажется полезной опытным программистам, желающим получить всестороннее и удобное руководство по разработке ПО. Так как эта книга фокусируется на конструировании — самой известной части жизненного цикла ПО, — описанные в ней методики будут понятны и программистам, имеющим соответствующее образование, и программистам-самоучкам.

### Технические лидеры

Многие технические лидеры используют первое издание этой книги для обучения менее опытных членов своих групп. Вы также можете использовать эту книгу для восполнения пробелов в своих знаниях. Если вы — опытный программист, то, наверное, согласитесь не со всеми моими выводами (обратное было бы странным), но, если вы прочитаете весь материал и обдумаете каждый поднятый вопрос, едва ли какая-то возникшая проблема конструирования окажется для вас новой.

## Программисты-самоучки

Если вы не имеете специального образования, вы не одиноки. Ежегодно программистами становятся около 50 000 человек (BLS, 2004, Hecker 2004), однако число дипломов, вручаемых ежегодно в нашей отрасли, составляет лишь около 35 000 (NCES, 2002). Легко прийти к выводу, что многие программисты изучают разработку ПО самостоятельно. Программисты-самоучки встречаются среди инженеров, бухгалтеров, ученых, преподавателей, владельцев малого бизнеса и представителей других профессий, которые занимаются программированием в рамках своей работы, но не всегда считают себя программистами. Каким бы ни было ваше программистское образование, в этом руководстве вы найдете информацию об эффективных методиках программирования.

## Студенты

В отличие от программистов, которые обладают опытом, но не могут похвастаться специальным обучением, недавние выпускники вузов часто имеют обширные теоретические знания, но плохо владеют практическими ноу-хау, связанными с созданием реальных программ. Передача практических навыков хорошего кодирования зачастую идет медленно, в ритуальных танцах архитекторов ПО, лидеров проектов, аналитиков и более опытных программистов. Еще чаще эти навыки приобретаются программистами в результате собственных проб и ошибок. Эта книга — альтернатива традиционным неспешным интеллектуальным ритуалам. В ней собраны полезные советы и эффективные стратегии разработки, которые ранее можно было узнать главным образом только непосредственно у других людей. Это трамплин для студентов, переходящих из академической среды в профессиональную.

# Где еще можно найти эту информацию?

В этой книге собраны методики конструирования из самых разнообразных источников. Многие знания о конструировании не только разрозненны, но и годами не попадают в печатные издания (Hildebrand, 1989; McConnell, 1997a). В эффективных, мощных методиках программирования, используемых лучшими программистами, нет ничего мистического, однако в повседневной череде неотложных задач очень немногие эксперты выкраивают время на то, чтобы поделиться своим опытом. Таким образом, программистам трудно найти хороший источник информации о программировании.

Методики, описанные в этой книге, заполняют пустоту, остающуюся в знаниях программистов после прочтения вводных и более серьезных учебников по программированию. Что читать человеку, изучившему книги типа «Introduction to Java», «Advanced Java» и «Advanced Advanced Java» и желающему узнать о программировании больше? Вы можете читать книги о процессорах Intel или Motorola, функциях ОС Microsoft Windows или Linux или о другом языке программирования — невозможно эффективно программировать, не имея хорошего представления о таких деталях. Но эта книга относится к числу тех немногих, в которых обсуждается программирование как таковое. Наибольшую пользу приносят те методики, которые можно использовать независимо от среды или языка. В других источниках такие методики обычно игнорируются, и именно поэтому я сосредоточился на них.

Как показано ниже, информация, представленная в этой книге, выжата из многих источников. Единственным другим способом получения этой информации является изучение горы книг и нескольких сотен технических журналов, дополненное значительным реальным опытом. Если вы уже проделали все это, данная книга все равно окажется вам полезной как удобный справочник.


# Главные достоинства этой книги

Какой бы ни была ваша ситуация, эта книга поможет вам создавать более качественные программы за меньшее время с меньшей головной болью.

***Полное руководство по конструированию ПО*** В этой книге обсуждаются такие общие аспекты конструирования, как качество ПО и подходы к размышлению о программировании. В то же время мы погрузимся в такие детали конструирования, как этапы создания классов, использование данных и управляющих структур, отладка, рефакторинг и методики и стратегии оптимизации кода. Чтобы изучить эти вопросы, вам не нужно читать книгу от корки до корки. Материал организован так, чтобы вы могли легко найти конкретную интересующую вас информацию.

***Готовые к использованию контрольные списки*** Эта книга включает десятки контрольных списков, позволяющих оценить архитектуру программы, подход к проектированию, качество классов и методов, имена переменных, управляющие структуры, форматирование, тесты и многое другое.

***Самая актуальная информация*** В этом руководстве вы найдете описания ряда самых современных методик, многие из которых еще не стали общепринятыми. Так как эта книга основана и на практике, и на исследованиях, рассмотренные в ней методики будут полезны еще многие годы.

***Более общий взгляд на разработку ПО*** Эта книга даст вам шанс подняться над суетой повседневной борьбы с проблемами и узнать, что работает, а что нет. Мало кто из практикующих программистов обладает временем, необходимым для прочтения сотен книг и журнальных статей, обобщенных в этом руководстве. Исследования и реальный опыт, на которых основана данная книга, помогут вам проанализировать ваши проекты и позволят принимать стратегические решения, чтобы не приходилось бороться с теми же врагами снова и снова.

***Объективность*** Некоторые книги по программированию содержат 1 грамм информации на 10 граммов рекламы. Здесь вы найдете сбалансированные обсуждения достоинств и недостатков каждой методики. Вы знаете свой конкретный проект лучше всех, и эта книга предоставит вам объективную информацию, нужную для принятия грамотных решений в ваших обстоятельствах.

***Независимость от языка*** Описанные мной методики позволяют выжать максимум почти из любого языка, будь то C++, C#, Java, Microsoft Visual Basic или другой похожий язык.

***Многочисленные примеры кода*** Эта книга содержит почти 500 примеров хорошего и плохого кода. Их так много потому, что лично я лучше всего учусь на примерах. Думаю, это относится и к другим программистам.

Примеры написаны на нескольких языках, потому что освоение более одного языка часто является поворотным пунктом в карьере профессионального программиста. Как только программист понимает, что принципы программирования не зависят от синтаксиса конкретного языка, он начинает приобретать знания, позволяющие достичь новых высот качества и производительности труда.

Чтобы как можно более облегчить бремя применения нескольких языков, я избегал редких возможностей языков, кроме тех фрагментов, в которых именно они и обсуждаются. Вам не нужно понимать каждый нюанс фрагментов кода, чтобы понять их суть. Если вы сосредоточитесь на обсуждаемых моментах, вы сможете читать код на любом языке. Чтобы сделать вашу задачу еще легче, я пояснил важные части примеров.

***Доступ к другим источникам информации*** В данном руководстве приводятся подробные сведения о конструировании ПО, но едва ли это последнее слово. В разделах «Дополнительные ресурсы» я указал другие книги и статьи, которые вы можете прочитать, если заинтересуетесь той или иной темой.

http://cc2e.com/1234

***Web-сайт книги*** Обновленные контрольные списки, списки книг и журнальных статей, Web-ссылки и другую информацию можно найти на Web-сайте *cc2e.com*. Для получения информации, связанной с «Code Complete, 2d ed.», введите в браузере *cc2e.com/* и четырехзначное число, пример которого показан слева. Читая книгу, вы много раз натолкнетесь на такие ссылки.

# Что побудило меня написать эту книгу?

Необходимость руководств, отражающих знания об эффективных методиках разработки ПО, ясна всем членам сообщества разработчиков. Согласно отчету совета Computer Science and Technology Board максимальное повышение качества и продуктивности разработки ПО будет достигнуто благодаря систематизации, унификации и распространению существующих знаний об эффективных методиках разработки (CSTB, 1990; McConnell, 1997a). Совет пришел к выводу, что стратегия распространения этих знаний должна быть основана на концепции руководств по разработке ПО.

## Тема конструирования игнорировалась

Одно время разработка ПО и кодирование рассматривались как одно и то же. Однако по мере идентификации разных процессов цикла разработки ПО лучшие умы отрасли стали посвящать время анализу и обсуждению методик управления проектами, выработки требований, проектирования и тестирования. Из-за пристального внимания к этим новым областям конструирование кода превратилось в бедного родственника разработки ПО.

Кроме того, обсуждению конструирования препятствовало предположение, согласно которому подход к конструированию как к отдельному *процессу* разработки ПО подразумевает, что конструирование нужно рассматривать при этом как отдельный *этап*. На самом деле процессы и этапы разработки не обязаны быть связаны какими-то отношениями, и обсуждение процесса конструирования полезно независимо от того, выполняются ли другие процессы разработки ПО как этапы, итерации или как-то иначе.

## Конструирование важно

Другая причина того, что конструирование игнорируется учеными и авторами, заключается в ошибочной идее, что в сравнении с другими процессами разработки ПО конструирование является относительно механическим процессом, допускающим мало возможностей улучшения. Ничто не может быть дальше от истины.

На конструирование кода обычно приходятся около 65% работы в небольших и 50% в средних проектах. Во время конструирования допускаются около 75% ошибок в неболь-

ших проектах и от 50 до 75% в средних и крупных. Очевидно, что любой процесс, связанный с такой долей ошибок, можно значительно улучшить (подробнее эти статистические данные рассматриваются в главе 27).

Некоторые авторы указывают, что, хотя ошибки конструирования и составляют высокий процент от общего числа ошибок, их обычно дешевле исправлять, чем ошибки в требованиях или архитектуре, поэтому они менее важны. Утверждение, что ошибки конструирования дешевле исправлять, верно, но вводит в заблуждение, потому что стоимость неисправленной ошибки конструирования может быть крайней высокой. Ученые обнаружили, что одними из самых дорогих ошибок в истории, приведшими к убыткам в сотни миллионов долларов, были мелкие ошибки кодирования (Weinberg, 1983; SEN, 1990). Невысокая стоимость исправления ошибок не подразумевает, что их исправление можно считать низкоприоритетной задачей.

Ирония ослабления внимания к конструированию состоит в том, что конструирование — единственный процесс, который выполняется всегда. Требования можно предположить, а не разработать, архитектуру можно обрисовать в самых общих чертах, а тестирование можно сократить или вообще опустить. Но если вы собираетесь написать программу, избежать конструирования не удастся, и это делает конструирование на редкость плодотворной областью улучшения методик разработки.

## Отсутствие похожих книг

Когда я начал подумывать об этой книге, я был уверен, что кто-то другой уже написал об эффективных методиках конструирования. Необходимость такой книги казалась очевидной. Но я обнаружил лишь несколько книг о конструировании, описывающих лишь некоторые его аспекты. Одни были написаны 15 или более лет назад и были основаны на относительно редких языках, таких как ALGOL, PL/I, Ratfor и Smalltalk. Другие были написаны профессорами, не работавшими над реальным кодом. Профессора писали о методиках, работающих в студенческих проектах, но часто не имели представления о том, как эти методики проявят себя в полномасштабных средах разработки. В третьих книгах авторы рекламировали новейшие методологии, игнорируя многие зрелые методики, эффективность которых прошла проверку временем.

Короче говоря, я не смог найти ни одной книги, автор которой хотя бы попытался отразить в ней практические приемы программирования, возникшие благодаря накоплению профессионального опыта, отраслевым исследованиям и академическим изысканиям. Обсуждение конструирования нужно было привести в соответствие современным языкам программирования, объектно-ориентированному программированию и ведущим методикам разработки. Ясно, что книгу о программировании должен был написать человек, знакомый с последними достижениями в области теории и в то же время создавший достаточно реального кода, чтобы хорошо представлять состояние практической сферы. Я писал эту книгу как всестороннее обсуждение конструирования кода, имеющее целью передачу знаний от одного программиста другому.

> Когда вместе собираются критики, они говорят о Теме, Композиции и Идее. Когда вместе собираются художники, они говорят о том, где купить дешевый скипидар.
>
> *Пабло Пикассо*

# К читателям

Я буду рад получить от вас вопросы по темам, обсуждаемым в этой книге, сообщения об обнаруженных ошибках, комментарии и предложения. Для связи со мной используйте адрес *stevemcc@construx.com* или мой Web-сайт *www.stevemcconnell.com*.

*Беллвью, штат Вашингтон*
*30 мая 2004 года*

## Служба поддержки Microsoft Learning Technical Support

Мы приложили все усилия, чтобы обеспечить точность сведений, изложенных в этой книге. Поправки к книгам издательства Microsoft Press публикуются в Интернете по адресу: *http://www.microsoft.com/learning/support/*

Чтобы подключиться к базе знаний Microsoft и задать вопрос или запросить ту или иную информацию, откройте страницу:

*http://www.microsoft.com/learning/support/search.asp*

Если у вас есть замечания, вопросы или предложения по поводу этой книги, присылайте их в Microsoft Press по обычной почте:

*Microsoft Press*
*Attn: Code Complete 2E Editor*
*One Microsoft Way*
*Redmond, WA 98052-6399*

или по электронной почте:

*mspinput@microsoft.com*

## Примечание издателя перевода

В книге приняты следующие условные графические обозначения:

Ключевой момент

Достоверные данные

Ужасный код

# Благодарности

Книги никогда не создаются в одиночку (по крайней мере это относится ко всем моим книгам), а работа над вторым изданием — еще более коллективное предприятие.

Мне хотелось бы поблагодарить всех, кто принял участие в обзоре данной книги: это Хакон Агустссон (Hбkon Bgъstsson), Скотт Эмблер (Scott Ambler), Уилл Барнс (Will Barns), Уильям Д. Бартоломью (William D. Bartholomew), Ларс Бергстром (Lars Bergstrom), Ян Брокбанк (Ian Brockbank), Брюс Батлер (Bruce Butler), Джей Цинкотта (Jay Cincotta), Алан Купер (Alan Cooper), Боб Коррик (Bob Corrick), Эл Корвин (Al Corwin), Джерри Девилль (Jerry Deville), Джон Ивз (Jon Eaves), Эдвард Эстрада (Edward Estrada), Стив Гоулдстоун (Steve Gouldstone), Оуэйн Гриффитс (Owain Griffiths), Мэтью Харрис (Matthew Harris), Майкл Ховард (Michael Howard), Энди Хант (Andy Hunt), Кевин Хатчисон (Kevin Hutchison), Роб Джаспер (Rob Jasper), Стивен Дженкинс (Stephen Jenkins), Ральф Джонсон (Ralph Johnson) и его группа разработки архитектуры ПО из Иллинойского университета, Марек Конопка (Marek Konopka), Джефф Лэнгр (Jeff Langr), Энди Лестер (Andy Lester), Митика Ману (Mitica Manu), Стив Маттингли (Steve Mattingly), Гарет Маккоан (Gareth McCaughan), Роберт Макговерн (Robert McGovern), Скотт Мейерс (Scott Meyers), Гарет Морган (Gareth Morgan), Мэтт Пелокин (Matt Peloquin), Брайан Пфладж (Bryan Pflug), Джеффри Рихтер (Jeffrey Richter), Стив Ринн (Steve Rinn), Даг Розенберг (Doug Rosenberg), Брайан Сен-Пьер (Brian St. Pierre), Диомидис Спиннелис (Diomidis Spinellis), Мэтт Стивенс (Matt Stephens), Дэйв Томас (Dave Thomas), Энди Томас-Крамер (Andy Thomas-Cramer), Джон Влиссидес (John Vlissides), Павел Возенилек (Pavel Vozenilek), Денни Уиллифорд (Denny Williford), Джек Вули (Jack Woolley) и Ди Зомбор (Dee Zsombor).

Сотни читателей прислали комментарии к первому изданию этой книги, и еще больше — ко второму. Спасибо всем, кто потратил время, чтобы поделиться в той или иной форме своим мнением.

Хочу особо поблагодарить рецензентов из Construx Software, которые провели формальную инспекцию всей рукописи: это Джейсон Хиллз (Jason Hills), Брейди Хонсингер (Bradey Honsinger), Абдул Низар (Abdul Nizar), Том Рид (Tom Reed) и Памела Перро (Pamela Perrott). Я был поистине удивлен тщательностью их обзора, особенно если учесть, сколько глаз изучило эту книгу до того, как они начали работать с ней. Спасибо также Брейди, Джейсону и Памеле за помощь в создании Web-сайта *cc2e.com*.

Мне было очень приятно работать с Девон Масгрейв (Devon Musgrave) — редактором этой книги. Я работал со многими прекрасными редакторами в других проектах, но даже на их фоне Девон выделяется добросовестностью и легким

характером. Спасибо, Девон! Благодарю Линду Энглман (Linda Engleman), которая поддержала идею второго издания — без нее эта книга не появилась бы. Благодарю также других сотрудников издательства Microsoft Press, в их число входят Робин ван Стинбург (Robin Van Steenburgh), Элден Нельсон (Elden Nelson), Карл Дилтц (Carl Diltz), Джоэл Панчо (Joel Panchot), Патрисия Массерман (Patricia Masserman), Билл Майерс (Bill Myers), Сэнди Резник (Sandi Resnick), Барбара Норфлит (Barbara Norfleet), Джеймс Крамер (James Kramer) и Прескотт Классен (Prescott Klassen).

Я хочу еще раз сказать спасибо сотрудникам Microsoft Press, участвовавшим в подготовке первого издания книги: это Элис Смит (Alice Smith), Арлен Майерс (Arlene Myers), Барбара Раньян (Barbara Runyan), Кэрол Люк (Carol Luke), Конни Литтл (Connie Little), Дин Холмс (Dean Holmes), Эрик Стру (Eric Stroo), Эрин О'Коннор (Erin O'Connor), Джинни Макгиверн (Jeannie McGivern), Джефф Кэри (Jeff Carey), Дженнифер Харрис (Jennifer Harris), Дженнифер Вик (Jennifer Vick), Джудит Блох (Judith Bloch), Кэтрин Эриксон (Katherine Erickson), Ким Эгглстон (Kim Eggleston), Лиза Сэндбург (Lisa Sandburg), Лиза Теобальд (Lisa Theobald), Маргарет Харгрейв (Margarite Hargrave), Майк Халворсон (Mike Halvorson), Пэт Фоджетт (Pat Forgette), Пегги Герман (Peggy Herman), Рут Петтис (Ruth Pettis), Салли Брунсмен (Sally Brunsman), Шон Пек (Shawn Peck), Стив Мюррей (Steve Murray), Уоллис Болц (Wallis Bolz) и Заафар Хаснаин (Zaafar Hasnain).

Наконец, я хотел бы выразить благодарность рецензентам, внесшим такой большой вклад в первое издание книги: это Эл Корвин (Al Corwin), Билл Кистлер (Bill Kiestler), Брайан Догерти (Brian Daugherty), Дэйв Мур (Dave Moore), Грег Хичкок (Greg Hitchcock), Хэнк Меуре (Hank Meuret), Джек Вули (Jack Woolley), Джой Уайрик (Joey Wyrick), Марго Пейдж (Margot Page), Майк Клейн (Mike Klein), Майк Зевенберген (Mike Zevenbergen), Пэт Форман (Pat Forman), Питер Пэт (Peter Pathe), Роберт Л. Гласс (Robert L. Glass), Тэмми Форман (Tammy Forman), Тони Пискулли (Tony Pisculli) и Уэйн Бердсли (Wayne Beardsley). Особо благодарю Тони Гарланда (Tony Garland) за его обстоятельный обзор: за 12 лет я еще лучше понял, как выиграла эта книга от тысяч комментариев Тони.

# Контрольные списки

Часть I

# ОСНОВЫ РАЗРАБОТКИ ПО

- **Глава 1.** Добро пожаловать в мир конструирования ПО!
- **Глава 2.** Метафоры, позволяющие лучше понять разработку ПО
- **Глава 3.** Семь раз отмерь, один раз отрежь: предварительные условия
- **Глава 4.** Основные решения, которые приходится принимать при конструировании

ГЛАВА 1

# Добро пожаловать в мир конструирования ПО!

http://cc2e.com/0178

**Содержание**



**Связанные темы**

- Кому следует прочитать эту книгу? (см. предисловие)
- Какую выгоду можно извлечь, прочитав эту книгу? (см. предисловие)
- Что побудило меня написать эту книгу? (см. предисловие)

Значение слова «конструирование» вне контекста разработки ПО известно всем: это то, что делают строители при сооружении жилого дома, школы или небоскреба. В детстве вы наверняка собирали разные предметы из «конструктора». Вообще под «конструированием» понимают процесс создания какого-нибудь объекта. Этот процесс может включать некоторые аспекты планирования, проектирования и тестирования, но чаще всего «конструированием» называют практическую часть создания чего-либо.

## 1.1. Что такое конструирование ПО?

Разработка ПО — непростой процесс, который может включать множество компонентов. Вот какие составляющие разработки ПО определили ученые за последние 25 лет:

- определение проблемы;
- выработка требований;
- создание плана конструирования;
- разработка архитектуры ПО, или высокоуровневое проектирование;
- детальное проектирование;
- кодирование и отладка;

- блочное тестирование;
- интеграционное тестирование;
- интеграция;
- тестирование системы;
- корректирующее сопровождение.

Если вы работали над неформальными проектами, то можете подумать, что этот список весьма бюрократичен. Если вы работали над слишком формальными проектами, вы это *знаете*! Достичь баланса между слишком слабым и слишком сильным формализмом нелегко — об этом мы еще поговорим.

Если вы учились программировать самостоятельно или работали преимущественно над неформальными проектами, вы, возможно, многие действия по созданию продукта объединили в одну категорию «программирование». Если вы работаете над неформальными проектами, то скорее всего, думая о создании ПО, вы представляете себе тот процесс, который ученые называют «конструированием».

Такое интуитивное представление о «конструировании» довольно верно, однако оно страдает от недостатка перспективы. Поэтому конструирование целесообразно рассматривать в контексте других процессов: это помогает сосредоточиться на задачах конструирования и уделять адекватное внимание другим важным действиям, к нему не относящимся. На рис. 1-1 показано место конструирования среди других процессов разработки ПО.


***Рис. 1-1.*** *Процессы конструирования изображены внутри серого эллипса. Главными компонентами конструирования являются кодирование и отладка, однако оно включает и детальное проектирование, блочное тестирование, интеграционное тестирование и другие процессы*

Как видите, конструирование состоит преимущественно из кодирования и отладки, однако включает и детальное проектирование, создание плана конструирования, блочное тестирование, интеграцию, интеграцион-

ное тестирование и другие процессы. Если бы эта книга была посвящена всем аспектам разработки ПО, в ней было бы приведено сбалансированное обсуждение всех процессов. Однако это руководство по методам конструирования, так что остальных тем я почти не касаюсь. Если бы эта книга была собакой, она тщательно принюхивалась бы к конструированию, виляла хвостом перед проектированием и тестированием и лаяла на прочие процессы.

Иногда конструирование называют «кодированием» или «программированием». «Кодирование» кажется мне в данном случае не лучшим термином, так как он подразумевает механическую трансляцию разработанного плана в команды языка программирования, тогда как конструирование вовсе не механический процесс и часто связано с творчеством и анализом. Смысл слов «программирование» и «конструирование» кажется мне похожим, и я буду использовать их как равноправные.

На рис. 1-1 разработка ПО была изображена в «плоском» виде; более точным отражением содержания этой книги является рис. 1-2.


***Рис. 1-2.*** *Кодирование и отладка, детальное проектирование, создание плана конструирования, блочное тестирование, интеграция, интеграционное тестирование и другие процессы обсуждаются в данной книге примерно в такой пропорции*

На рис. 1-1 и 1-2 процессы конструирования представлены с общей точки зрения. Но что можно сказать об их деталях? Вот некоторые конкретные задачи, связанные с конструированием:

- проверка выполнения условий, необходимых для успешного конструирования;
- определение способов последующего тестирования кода;
- проектирование и написание классов и методов;
- создание и присвоение имен переменным и именованным константам;
- выбор управляющих структур и организация блоков команд;

- блочное тестирование, интеграционное тестирование и отладка собственного кода;
- взаимный обзор кода и низкоуровневых программных структур членами группы;
- «шлифовка» кода путем его тщательного форматирования и комментирования;
- интеграция программных компонентов, созданных по отдельности;
- оптимизация кода, направленная на повышение его быстродействия, и снижение степени использования ресурсов.

Еще более полное представление о процессах и задачах конструирования вы получите, просмотрев содержание книги.

Конструирование включает так много задач, что вы можете спросить: «Ладно, а что *не* является частью конструирования?» Хороший вопрос. В конструирование не входят такие важные процессы, как управление, выработка требований, разработка архитектуры приложения, проектирование пользовательского интерфейса, тестирование системы и ее сопровождение. Все они не меньше, чем конструирование, влияют на конечный успех проекта — по крайней мере любого проекта, который требует усилий более одного-двух человек и длится больше нескольких недель. Все эти процессы стали предметом хороших книг, многие из которых я указал в разделах «Дополнительные ресурсы» и в главе 35.

## 1.2. Почему конструирование ПО так важно?

Раз уж вы читаете эту книгу, вы наверняка понимаете важность улучшения качества ПО и повышения производительности труда разработчиков. Многие из самых удивительных современных проектов основаны на применении ПО: Интернет и спецэффекты в кинематографе, медицинские системы жизнеобеспечения и космические программы, высокопроизводительный анализ финансовых данных и научные исследования. Эти, а также более традиционные проекты имеют много общего, поэтому применение улучшенных методов программирования окупится во всех случаях.

Признавая важность улучшения разработки ПО в целом, вы можете спросить: «Почему именно конструированию в этой книге уделяется такое внимание?»

Ответы на этот вопрос приведены ниже.

***Конструирование — крупная часть процесса разработки ПО*** В зависимости от размера проекта на конструирование обычно уходит 30–80 % общего времени работы. Все, что занимает так много времени работы над проектом, неизбежно влияет на его успешность.

**Перекрестная ссылка** О связи между размером проекта и долей времени, уходящего на конструирование, см. подраздел «Соотношение между выполняемыми операциями и размер» раздела 27.5.

***Конструирование занимает центральное место в процессе разработки ПО*** Требования к приложению и его архитектура разрабатываются до этапа конструирования, чтобы гарантировать его эффективность. Тестирование системы (в строгом смысле независимого тестирования) выполняется после конструирования и служит для проверки его правильности. Конструирование — центр процесса разработки ПО.

**Перекрестная ссылка** О производительности труда программистов см. подраздел «Индивидуальные различия» раздела 28.5.

***Повышенное внимание к конструированию может намного повысить производительность труда отдельных программистов*** В своем классическом исследовании Сэкман, Эриксон и Грант показали, что производительность труда отдельных программистов во время конструирования изменяется в 10–20 раз (Sackman, Erikson, and Grant, 1968). С тех пор эти данные были подтверждены другими исследованиями (Curtis, 1981; Mills, 1983; Curtis et al., 1986; Card, 1987; Valett and McGarry, 1989; DeMarco and Lister, 1999№; Boehm et al., 2000). Эта книга поможет всем программистам изучить методы, которые уже используются лучшими разработчиками.

***Результат конструирования — исходный код — часто является единственным верным описанием программы*** Зачастую единственным видом доступной программистам документации является сам исходный код. Спецификации требований и проектная документация могут устареть, но исходный код актуален всегда, поэтому он должен быть максимально качественным. Последовательное применение методов улучшения исходного кода — вот что отличает детальные, корректные и поэтому информативные программы от устройств Руба Голдберга[1]. Эффективнее всего применять эти методы на этапе конструирования.

***Конструирование — единственный процесс, который выполняется во всех случаях*** Идеальный программный проект до начала конструирования проходит стадии тщательной выработки требований и проектирования архитектуры. После конструирования в идеале должно быть выполнено исчерпывающее, статистически контролируемое тестирование системы. Однако в реальных проектах нашего несовершенного мира разработчики часто пропускают этапы выработки требований и проектирования, начиная прямо с конструирования программы. Тестирование также часто выпадает из расписания из-за огромного числа ошибок и недостатка времени. Но каким бы срочным или плохо спланированным ни был проект, куда без конструирования деться? Так что повышение эффективности конструирования ПО позволяет оптимизировать любой проект, каким бы несовершенным он ни был.

## 1.3. Как читать эту книгу

Вы можете читать эту книгу от корки до корки или по отдельным темам. Если вы предпочитаете первый вариант, переходите к главе 2. Если второй — можете начать с главы 6 и переходить по перекрестным ссылкам к другим темам, которые вас заинтересуют. Если вы не уверены, какой из этих вариантов вам подходит, начните с раздела 3.2.

---

[1] Голдберг, Рубен Лушес («Руб») [Goldberg, «Rube» (Reuben Lucius)] (1883–1970) — карикатурист, скульптор. Известен своими карикатурами, в которых выдуманное им сложное оборудование («inventions») выполняет примитивные и никому не нужные операции. Лауреат Пулитцеровской премии 1948 г. за политические карикатуры. — *Прим. перев.*

## Ключевые моменты

- Конструирование — главный этап разработки ПО, без которого не обходится ни один проект.
- Основные этапы конструирования: детальное проектирование, кодирование, отладка, интеграция и тестирование приложения разработчиками (блочное тестирование и интеграционное тестирование).
- Конструирование часто называют «кодированием» и «программированием».
- От качества конструирования во многом зависит качество ПО.
- В конечном счете ваша компетентность в конструировании ПО определяет то, насколько хороший вы программист. Совершенствованию ваших навыков и посвящена оставшаяся часть этой книги.

ГЛАВА 2

# Метафоры, позволяющие лучше понять разработку ПО

http://cc2e.com/0278

**Содержание**



**Связанная тема**

- Эвристика при проектировании: подраздел «Проектирование — эвристический процесс» в разделе 5.1

Терминология компьютерных наук — одна из самых красочных. Действительно, в какой еще области существуют стерильные комнаты с тщательно контролируемой температурой, заполненные вирусами, троянскими конями, червями, жучками и прочей живностью и нечистью?

Все эти яркие метафоры описывают специфические аспекты мира программирования. Более общие явления характеризуются столь же красочными метафорами, позволяющих лучше понять процесс разработки ПО.

Остальная часть книги не зависит от обсуждения метафор в этой главе. Можете пропустить ее, если хотите быстрее добраться до практических советов. Если хотите яснее представлять разработку ПО, читайте дальше.

## 2.1. Важность метафор

Проведение аналогий часто приводит к важным открытиям. Сравнив не совсем понятное явление с чем-то похожим, но более понятным, вы можете догадаться, как справиться с проблемой. Такое использование метафор называется «моделированием».

История науки полна открытий, сделанных благодаря метафорам. Так, химик Кекуле однажды во сне увидел змею, схватившую себя за хвост. Проснувшись, он понял, что свойства бензола объяснила бы молекулярная структура, имеющая похожую кольцевую форму. Дальнейшие эксперименты подтвердили его гипотезу (Barbour, 1966).

Кинетическая теория газов была создана на основе модели «бильярдных шаров», согласно которой молекулы газа, подобно бильярдным шарам, имеют массу и совершают упругие соударения.

Волновая теория света была разработана преимущественно путем исследования сходств между светом и звуком. И свет, и звук имеют амплитуду (яркость — громкость), частоту (цвет — высота) и другие общие свойства. Сравнение волновых теорий звука и света оказалось столь продуктивным, что ученые потратили много сил, пытаясь обнаружить среду, которая распространяла бы свет, как воздух распространяет звук. Они даже дали этой среде название — «эфир», но так и не смогли ее обнаружить. В данном случае аналогия была такой убедительной, что ввела ученых в заблуждение.

В целом эффективность моделей объясняется их яркостью и концептуальной целостностью. Модели подсказывают ученым свойства, отношения и перспективные области исследований. Иногда модели вводят в заблуждение; как правило, к этому приводит чрезмерное обобщение метафоры. Поиск эфира — наглядный пример чрезмерного обобщения модели.

Разумеется, некоторые метафоры лучше других. Хорошими метафорами можно считать те, что отличаются простотой, согласуются с другими релевантными метафорами и объясняют многие экспериментальные данные и наблюдаемые явления.

Рассмотрим, к примеру, колебания камня, подвешенного на веревке. До Галилея сторонники Аристотеля считали, что тяжелый объект перемещается из верхней точки в нижнюю, переходя в состояние покоя. В данном случае они подумали бы, что камень падает, но с осложнениями. Когда Галилей смотрел на раскачивающийся камень, он видел маятник, поскольку камень снова и снова повторял одно и то же движение.

Указанные модели фокусируют внимание на совершенно разных факторах. Последователи Аристотеля, рассматривавшие раскачивающийся камень как падающий объект, принимали в расчет вес камня, высоту, на которую он был поднят, и время, проходящее до достижения камнем состояния покоя. В модели Галилея важными были другие факторы. Он обращал внимание на вес камня, угловое смещение, радиус и период колебаний маятника. Благодаря этому Галилей открыл законы, которые последователи Аристотеля открыть не смогли, так как их модель заставила их наблюдать за другими явлениями и задавать другие вопросы.

Аналогичным образом метафоры способствуют и лучшему пониманию вопросов разработки ПО. Во время лекции по случаю получения премии Тьюринга в 1973 г., Чарльз Бахман (Charles Bachman) упомянул переход от доминировавшего геоцентрического представления о Вселенной к гелиоцентрическому. Геоцентрическая модель Птолемея не вызывала почти никаких сомнений целых 1400 лет. Затем, в 1543 г., Коперник выдвинул гелиоцентрическую теорию, предположив, что центром Вселенной на самом деле является Солнце, а не Земля. В конечном итоге такое изменение умозрительных моделей привело к открытию новых планет, исключению Луны из категории планет и переосмыслению места человечества во Вселенной.

> Не стоит недооценивать важность метафор. Метафоры имеют одно неоспоримое достоинство: описываемое ими поведение предсказуемо и понятно всем людям. Это сокращает объем ненужной коммуникации, способствует достижению взаимопонимания и ускоряет обучение. По сути метафора — это способ осмысления и абстрагирования концепций, позволяющий думать в более высокой плоскости и избегать низкоуровневых ошибок.
>
> *Фернандо Дж. Корбати (Fernando J. Corbaty)*

Переход от геоцентрического представления к гелиоцентрическому в астрономии Бахман сравнил с изменением, происходившим в программировании в начале 1970-х. В это время центральное место в моделях обработки данных стали отводить не компьютерам, а базам данных. Бахман указал, что создатели ранних моделей стремились рассматривать все данные как последовательный поток карт, «протекающий» через компьютер (компьютеро-ориентированный подход). Суть изменения заключалась в отведении центрального места пулу данных, над которыми компьютер выполняет некоторые действия (подход, ориентированный на БД).

Сегодня почти невозможно найти человека, считающего, что Солнце вращается вокруг Земли. Столь же трудно представить программиста, который бы думал, что все данные можно рассматривать как последовательный поток карт. После опровержения старых теорий трудно понять, как кто-то когда-то мог в них верить. Еще удивительнее то, что приверженцам этих старых теорий новые теории казались такими же нелепыми, как нам старые.

Геоцентрическое представление о Вселенной мешало астрономам, которые сохранили верность этой теории после появления лучшей. Аналогично компьютеро-ориентированное представление о компьютерной вселенной тянуло назад компьютерных ученых, которые продолжили придерживаться его после появления теории, ориентированной на БД.

Иногда люди упрощают суть метафор. На каждый из описанных примеров так и тянет ответить: «Разумеется, правильная метафора более полезна. Другая метафора была неверной!» Так-то оно так, но это слишком упрощенное представление. История науки — не серия переходов от «неверных» метафор к «верным». Это серия переходов от «менее хороших» метафор к «лучшим», от менее полных теорий к более полным, от адекватного описания одной области к адекватному описанию другой.

В действительности многие модели, на смену которым приходят лучшие модели, сохраняют свою полезность. Так, инженеры до сих пор решают большинство своих задач, опираясь на ньютонову динамику, хотя в теоретической физике ее вытеснила теория Эйнштейна.

Разработка ПО — относительно молодая область науки. Она еще недостаточно зрелая, чтобы иметь набор стандартных метафор. Поэтому она включает массу второстепенных и противоречивых метафор. Одни из них лучше другие — хуже. Оттого, насколько хорошо вы понимаете метафоры, зависит и ваше понимание разработки ПО.

## 2.2. Как использовать метафоры?

Метафора, характеризующая разработку ПО, больше похожа на прожектор, чем на дорожную карту. Она не говорит, где найти ответ — она говорит, как его искать. Метафора — это скорее эвристический подход, а не алгоритм.

Алгоритмом называют последовательность четко определенных команд, которые необходимо выполнить для решения конкретной задачи. Алгоритм предсказуем, детерминирован и не допускает случайностей. Алгоритм говорит, как пройти из точки А в точку Б не дав крюку, без посещения точек В, Г и Д и без остановок на чашечку кофе.

Эвристика — это метод, помогающий искать ответ. Результаты его применения могут быть в некоторой степени случайными, потому что эвристика указывает только способ поиска, но не говорит, что искать. Она не говорит, как дойти прямо из точки А в точку Б; даже положение этих точек может быть неизвестно. Эвристика — это алгоритм в шутовском наряде. Она менее предсказуема, более забавна и поставляется без 30-дневной гарантии с возможностью возврата денег.

Вот алгоритм, позволяющий добраться до чьего-то дома: поезжайте по шоссе 167 на юг до городка Пюиолап. Сверните на аллею Сауз-Хилл, а дальше 4,5 мили вверх по холму. Поверните у продуктового магазина направо, а на следующем перекрестке — налево. Доехав до дома 714, расположенного на левой стороне улицы, остановитесь и выходите из автомобиля.

А эвристическое правило может быть таким: найдите наше последнее письмо. Езжайте в город, указанный на конверте. Оказавшись в этом городе, спросите кого-нибудь, где находится наш дом. Все нас знают — кто-нибудь с радостью вам поможет. Если никого не встретите, позвоните нам из телефона-автомата, и мы за вами приедем.

**Перекрестная ссылка** Об использовании эвристики при проектировании ПО см. подраздел «Проектирование — эвристический процесс» раздела 5.1.

Различия между алгоритмом и эвристикой тонки, и в чем-то эти два понятия перекрываются. В этой книге основным различием между ними я буду считать степень косвенности решения. Алгоритм предоставляет вам сами команды. Эвристика сообщает вам, как обнаружить команды самостоятельно или по крайней мере где их искать.

Наличие директив, точно определяющих способ решения проблем программирования, непременно сделало бы программирование более легким, а результаты более предсказуемыми. Но наука программирования еще не продвинулась так далеко, а может, этого никогда и не случится. Самая сложная часть программирования — концептуализация проблемы, и многие ошибки программирования являются концептуальными. С концептуальной точки зрения каждая программа уникальна, поэтому трудно или даже невозможно разработать общий набор директив, приводящих к решению во всех случаях. Так что знание общего подхода к проблемам не менее, а то и более ценно, чем знание точных решений конкретных проблем.

Как использовать метафоры, характеризующие разработку ПО? Используйте так, чтобы лучше разобраться в проблемах и процессах программирования, чтобы они помогали вам размышлять о выполняемых действиях и придумывать лучшие решения задач. Конечно, вы не сможете взглянуть на строку кода и сказать, что она противоречит одной из метафор, описываемых в этой главе. Но со временем тот, кто использует метафоры, лучше поймет программирование и будет быстрее создавать более эффективный код, чем тот, кто их не использует.

## 2.3. Популярные метафоры, характеризующие разработку ПО

Множество метафор, описывающих разработку ПО, смутит кого угодно. Дэвид Грайс утверждает, что написание ПО — это наука (Gries, 1981). Дональд Кнут считает это искусством (Knuth, 1998). Уоттс Хамфри говорит, что это процесс (Humphrey, 1989). Ф. Дж. Плоджер и Кент Бек утверждают, что разработка ПО похожа на управление автомобилем, однако приходят к почти противоположным выводам (Plauger, 1993, Beck, 2000). Алистер Кокберн сравнивает разработку ПО с игрой (Cockburn, 2002), Эрик Реймонд — с базаром (Raymond, 2000), Энди Хант (Andy Hunt) и Дэйв Томас (Dave Thomas) — с работой садовника, Пол Хекель — со съемкой фильма «Белоснежка и семь гномов» (Heckel, 1994). Фред Брукс упоминает фермерство, охоту на оборотней и динозавров, завязших в смоляной яме (Brooks, 1995). Какие метафоры самые лучшие?

### Литературная метафора: написание кода

Самая примитивная метафора, описывающая разработку ПО, берет начало в выражении «написание кода». Согласно литературной метафоре разработка программы похожа на написание письма: вы садитесь за стол, берете бумагу, перо и пишете письмо с начала до конца. Это не требует никакого формального планирования, а мысли, выражаемые в письме, формулируются автором по ходу дела.

На этой метафоре основаны и многие другие идеи. Джон Бентли (Jon Bentley) говорит, что программист должен быть способен сесть у камина со стаканом бренди, хорошей сигарой, охотничьей собакой у ног и «сформулировать программу» подобно тому, как писатели создают романы. Брайан Керниган и Ф. Дж. Плоджер назвали свою книгу о стиле программирования «The Elements of Programming Style» (Kernighan and Plauger, 1978), обыгрывая название книги о литературном стиле «The Elements of Style» (Strunk and White, 2000). Программисты часто говорят об «удобочитаемости программы».

Индивидуальную работу над небольшими проектами метафора написания письма характеризует довольно точно, но в целом она описывает разработку ПО неполно и неадекватно. Письма и романы обычно принадлежат перу одного человека, тогда как над программами обычно работают группы людей с разными сферами ответственности. Закончив писать письмо, вы запечатываете его в конверт и отправляете. С этого момента изменить вы его не можете, и письмо во всех отношениях является завершенным. Изменить ПО не так уж трудно, и вряд ли работу над ним можно когда-нибудь признать законченной. Из общего объема работы над типичной программной системой две трети обычно выполняются после выпуска первой версии программы, а иногда эта цифра достигает целых 90 % (Pigoski, 1997). В литературе поощряется оригинальность. При конструировании ПО оригинальный подход часто оказывается менее эффективным, чем повторное использование идей, кода и тестов из предыдущих проектов. Словом, процесс разработки ПО, соответствующий литературной метафоре, является слишком простым и жестким, чтобы быть полезным.

К сожалению, литературная метафора была увековечена в одной из самых популярных книг по разработке ПО — книге Фреда Брукса «The Mythical Man-Month» («Мифический человеко-месяц») (Brooks, 1995). Брукс пишет: «Планируйте выбросить первый экземпляр программы: вам в любом случае придется это сделать». Перед глазами невольно возникает образ мусорного ведра, полного черновиков (рис. 2-1).

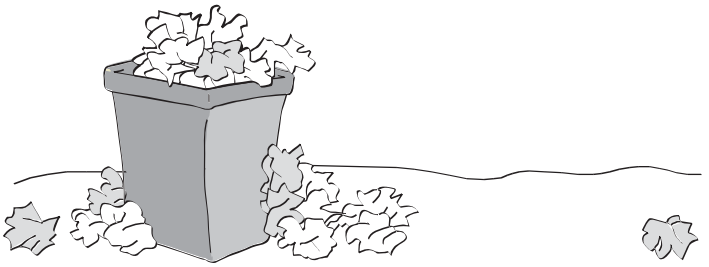


***Рис. 2-1.*** *Литературная метафора наводит на мысль, что процесс разработки ПО основан на дорогостоящем методе проб и ошибок, а не на тщательном планировании и проектировании*

> Планируйте выбросить первый экземпляр программы: вам в любом случае придется это сделать.
>
> *Фред Брукс*

> Если вы планируете выбросить первый экземпляр программы, вы выбросите и второй.
>
> *Крейг Зеруни (Craig Zerouni)*

Подобный подход может быть практичным, если вы пишете банальное письмо своей тетушке. Однако расширение метафоры «написания» ПО вплоть до выбрасывания первого экземпляра программы — не лучший совет в мире разработки ПО, где крупная система по стоимости уже сравнялась с 10-этажным офисным зданием или океанским лайнером. Конечно, это не имело бы значения, если б вы имели бесконечные запасы времени и средств. Однако реальные условия таковы, что разработчики должны создавать программы с первого раза или хотя бы минимизировать объем дополнительных расходов в случае неудач. Другие метафоры лучше иллюстрируют достижение таких целей.

## Сельскохозяйственная метафора: выращивание системы

Некоторые разработчики заявляют, что создание ПО следует рассматривать по аналогии с выращиванием сельскохозяйственных культур. Вы проектируете отдельный блок, кодируете его, тестируете и добавляете в систему, чуть расширяя с каждым разом ее функциональность. Такое разбиение задачи на множество небольших действий позволяет минимизировать проблемы, с которыми можно столкнуться на каждом этапе.

Иногда хороший метод описывается плохой метафорой. В таких случаях попытайтесь сохранить метод и обнаружить лучшую метафору. В данном случае инкрементный подход полезен, но сельскохозяйственная метафора просто ужасна.

**Дополнительные сведения** О другой сельскохозяйственной метафоре, употребляемой в контексте сопровождения ПО, см. главу «On the Origins of Designer Intuition» книги «Rethinking Systems Analysis and Design» (Weinberg, 1988).

Возможно, идея выполнения небольшого объема работы зараз и напоминает рост растений, но сельскохозяйственная аналогия неубедительна и неинформативна, к тому же ее легко заменить лучшими метафорами, которые описаны ниже. Сельскохозяйственную метафору трудно расширить за пределы идеи выполнения чего-либо небольшими частями. Ее приверженцы (рис. 2-2) рискуют в итоге заговорить об удобрении плана системы, прореживании детального проекта, повышении урожайности кода путем оптимизации землеустройства и уборке урожая самого кода. Вы начнете говорить о чередовании посевов C++ и о том, что было бы неплохо оставить систему под паром для повышения концентрации азота на жестком диске.

Слабость данной метафоры заключается в предположении, что у вас нет прямого контроля над развитием ПО. Вы просто сеете семена кода весной и, если на то будет воля Великой Тыквы, осенью получите невиданный урожай кода.

***Рис. 2-2.*** *Нелегко адекватно расширить сельскохозяйственную метафору на область разработки ПО*

## Метафора жемчужины: медленное приращение системы

Иногда, говоря о выращивании ПО, на самом деле имеют в виду приращение, или аккрецию (accretion). Две этих метафоры тесно связаны, но вторая более убедительна. Приращение характеризует процесс формирования жемчужины за счет отложения небольших объемов карбоната кальция. В геологии и юриспруденции под аккрецией понимают увеличение территории суши посредством отложения содержащихся в воде пород.

**Перекрестная ссылка** О применении инкрементных стратегий при интеграции системы см. раздел 29.2.

Это не значит, что вы должны освоить создание кода из осадочных пород; это означает, что вы должны научиться добавлять в программные системы по небольшому фрагменту за раз. Другими словами, которые в связи с этим приходят на ум, являются термины «инкрементный», «итеративный», «адаптивный» и «эволюционный». Инкрементное проектирование, конструирование и тестирование — одни из самых эффективных концепций разработки ПО.

При инкрементной разработке вы сначала создаете самую простую версию системы, которую можно было бы запустить. Она может не принимать реальных данных, может не выполнять над ними реальных действий, может не генерировать реальные результаты — она должна быть просто скелетом, достаточно крепким,

чтобы поддерживать реальную систему по мере ее разработки. Она может вызывать поддельные классы для каждой из определенных вами основных функций. Такая система похожа на песчинку, с которой начинается образование жемчужины.

Создав скелет, вы начинаете понемногу наращивать плоть. Каждый из фиктивных классов вы заменяете реальным. Вместо того чтобы имитировать ввод данных, вы пишете код, на самом деле принимающий реальные данные. А вместо имитации вывода данных — код, на самом деле выводящий данные. Вы продолжаете добавлять нужные фрагменты, пока не получаете полностью рабочую систему.

Эффективность такого подхода можно подтвердить двумя впечатляющими примерами. Фред Брукс, который в 1975 г. предлагал выбрасывать первый экземпляр программы, заявил, что за десять лет, прошедших с момента написания им знаменитой книги «Мифический человеко-месяц», ничто не изменяло его работу и ее эффективность так радикально, как инкрементная разработка (Brooks, 1995). Аналогичное заявление было сделано Томом Гилбом в революционной книге «Principles of Software Engineering Management» (Gilb, 1988), в которой он представил метод эволюционной поставки программы (evolutionary delivery) и разработал многие основы современного гибкого программирования (agile programming). Многие другие современные методологии также основаны на идее инкрементной разработки (Beck, 2000; Cockburn, 2002; Highsmith, 2002; Reifer, 2002; Martin, 2003; Larman, 2004).

Достоинство инкрементной метафоры в том, что она не дает чрезмерных обещаний. Кроме того, она не так легко поддается неуместному расширению, как сельскохозяйственная метафора. Раковина, формирующая жемчужину, — хороший вариант визуализации инкрементной разработки, или аккреции.

## Строительная метафора: построение ПО

Метафора «построения» ПО полезнее, чем метафоры «написания» или «выращивания» ПО, так как согласуется с идеей аккреции ПО и предоставляет более детальное руководство. Построение ПО подразумевает наличие стадий планирования, подготовки и выполнения, тип и степень выраженности которых зависят от конкретного проекта. При изучении этой метафоры вы найдете и другие параллели.

Для построения метровой башни требуется твердая рука, ровная поверхность и 10 пивных банок, для башни же в 100 раз более высокой недостаточно иметь в 100 раз больше пивных банок. Такой проект требует совершенно иного планирования и конструирования.

Если вы строите простой объект, скажем, собачью конуру, вы можете пойти в хозяйственный магазин, купить доски, гвозди, и к вечеру у Фидо будет новый дом. Если вы забудете про лаз или допустите какую-нибудь другую ошибку, ничего страшного: вы можете ее исправить или даже начать все сначала (рис. 2-3). Все, что вы при этом потеряете, — время. Такой свободный подход уместен и в небольших программных проектах. Если вы плохо спроектируете 1000 строк кода, то сможете выполнить рефакторинг или даже начать проект заново, и это не приведет к крупным потерям.

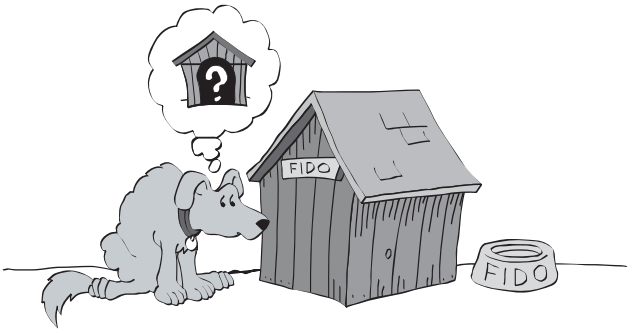



***Рис. 2-3.*** *За ошибку, допущенную при создании простого объекта, приходится расплачиваться лишь потраченным временем и, возможно, некоторым разочарованием*

Построить дом сложнее, и плохое проектирование при этом приводит к куда более серьезным последствиям. Сначала вы должны решить, какой тип здания вы хотите построить, что аналогично определению проблемы при разработке ПО. Затем вы с архитектором должны разработать и утвердить общий план, что похоже на разработку архитектуры. Далее вы чертите подробные чертежи и нанимаете бригаду строителей — это аналогично детальному проектированию ПО. Вы готовите стройплощадку, закладываете фундамент, создаете каркас дома, обшиваете его, кроете крышу и проводите в дом все коммуникации — это похоже на конструирование ПО. Когда строительство почти завершено, в дело вступают ландшафтные дизайнеры, маляры и декораторы, делающие дом максимально удобным и привлекательным. Это напоминает оптимизацию ПО. Наконец, на протяжении всего строительства вас посещают инспекторы, проверяющие стройплощадку, фундамент, электропроводку и все, что можно проверить. При разработке ПО этому соответствуют обзоры и инспекция проекта.

И в строительстве, и в программировании увеличение сложности и масштаба проекта сопровождается ростом цены ошибок. Конечно, для создания дома нужны довольно дорогие материалы, однако главной статьей расходов является оплата труда рабочих. Перемещение стены на 15 см обойдется дорого не потому, что при этом будет потрачено много гвоздей, а потому, что вам придется оплатить дополнительное время работы строителей. Чтобы не тратить время на исправление ошибок, которых можно избежать, вы должны как можно лучше выполнить проектирование (рис. 2-4). Материалы, необходимые для создания программного продукта, стоят дешевле, чем стройматериалы, однако затраты на рабочую силу в обоих случаях примерно одинаковы. Изменение формата отчета обходится ничуть не дешевле, чем перемещение стены дома, потому что главным компонентом затрат в обоих случаях является время людей.

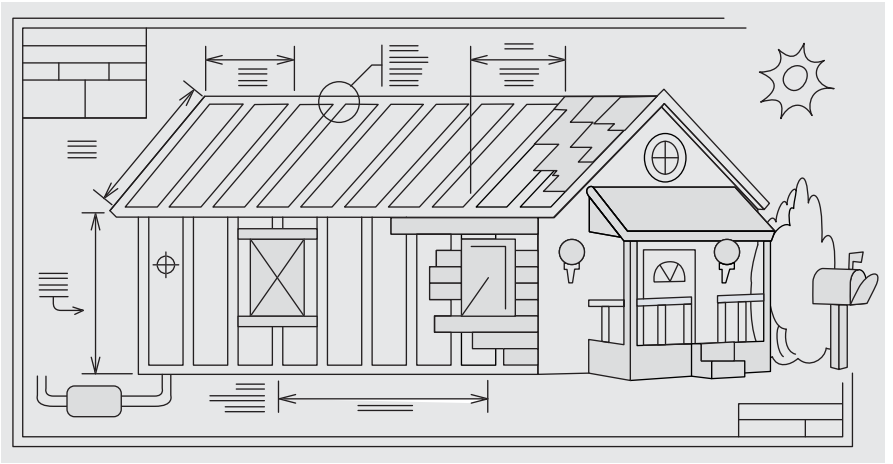


***Рис. 2-4.*** *Более сложные объекты требуют более тщательного планирования*

Какие еще параллели можно провести между сооружением дома и разработкой ПО? При возведении дома никто не пытается конструировать вещи, которые можно купить. Здравомыслящему человеку и в голову не придет самостоятельно разрабатывать и создавать стиральную машину, холодильник, шкафы, окна и двери, если все это можно приобрести. Создавая программную систему, вы поступите так же. Вы будете в полной мере использовать возможности высокоуровневого языка вместо того, чтобы писать собственный код на уровне ОС. Возможно, вы используете также встроенные библиотеки классов-контейнеров, научные функции, классы пользовательского интерфейса и классы для работы с БД. Обычно невыгодно писать компоненты, которые можно купить готовыми.

Однако если вы хотите построить нестандартный дом с первоклассной меблировкой, мебель, возможно, придется заказать. Вы можете заказать встроенные посудомоечную машину и холодильник, чтобы они выглядели как часть обстановки. Вы можете заказать окна необычных форм и размеров. Такое изготовление предметов на заказ имеет параллели и в мире разработки ПО. При работе над приложением высшего класса для достижения более высокой скорости и точности расчетов или реализации необычного интерфейса иногда приходится создавать собственные научные функции, собственные классы-контейнеры и другие компоненты.

И конструирование дома, и конструирование ПО можно оптимизировать, выполнив адекватное планирование. Если создавать ПО в неверном порядке, его будет трудно писать, тестировать и отлаживать. Сроки затянутся, да и весь проект может завершиться неудачей из-за чрезмерной сложности отдельных компонентов, не позволяющей разработчикам понять работу всей системы.

Тщательное планирование не значит исчерпывающее или чрезмерное. Вы можете спланировать основные структурные компоненты и позднее решать, чем покрыть пол, в какой цвет окрасить стены, какой использовать кровельный материал и т. д. Хорошо спланированный проект открывает больше возможностей для изменения решения на более поздних этапах работы. Чем лучше вам известен тип создаваемого ПО, тем больше деталей вы можете принимать как данное. Вы про-

сто должны убедиться в проведении достаточного планирования, чтобы его недостаток не привел позднее к серьезным проблемам.

Аналогия конструирования также помогает понять, почему разные программные проекты призывают к разным подходам разработки. Склад и медицинский центр или ядерный реактор также требуют разных степеней планирования, проектирования и контроля качества, и никто не стал бы одинаково подходить к строительству школы, небоскреба и жилого дома с тремя спальнями. Работая над ПО, вы обычно можете использовать гибкие упрощенные подходы, но иногда для обеспечения безопасности и других целей необходимы и жесткие, тщательно продуманные подходы.

Проблема изменения ПО приводит нас к еще одной параллели. Перемещение несущей стены на 15 см обходится гораздо дороже, чем перемещение перегородки между комнатами. Аналогично внесение структурных изменений в программу требует больших затрат, чем добавление или удаление второстепенных возможностей.

Наконец, проведение аналогии с домостроительством позволяет лучше понять работу над очень крупными программными проектами. При создании очень крупного объекта цена неудачи слишком высока, поэтому объект надо спроектировать тщательнейшим образом. Строительные организации скрупулезно разрабатывают и инспектируют свои планы. Все крупные здания создаются с большим запасом прочности; лучше заплатить на 10 % больше за более прочный материал, чем рисковать крушением небоскреба. Кроме того, большое внимание уделяется времени. При возведении Эмпайр Стейт Билдинг время прибытия каждого грузовика, поставлявшего материалы, задавалось с точностью до 15 минут. Если грузовик не прибывал в нужное время, задерживалась работа над всем проектом.

Аналогично, очень крупные программные проекты требуют планирования более высокого порядка, чем просто крупные проекты. Кейперс Джонс сообщает, что программная система из одного миллиона строк кода требует в среднем 69 *видов* документации (Jones, 1998). Спецификация требований к такой системе обычно занимает 4000–5000 страниц, а проектная документация вполне может быть еще в 2 или 3 раза более объемной. Маловероятно, чтобы один человек мог понять весь проект такого масштаба или даже прочитать всю документацию, поэтому подобные проекты требуют более тщательной подготовки.

По экономическому масштабу некоторые программные проекты сравнимы с возведением «Эмпайр Стейт Билдинг», и контролироваться они должны соответствующим образом.

**Дополнительные сведения** Грамотные комментарии по поводу расширения метафоры конструирования см. в статье «What Supports the Roof?» (Starr 2003).

Метафора построения-конструирования может быть расширена во многих других направлениях, именно поэтому она столь эффективна. Благодаря этой метафоре отрасль разработки ПО обогатилась многими популярными терминами, такими как архитектура ПО, леса (scaffolding), конструирование и фундаментальные классы. Наверное, вы сможете назвать и другие примеры.

## Применение методов разработки ПО: интеллектуальный инструментарий

Люди, эффективно разрабатывающие высококачественное ПО, многие годы посвятили сбору методов, хитростей и магических заклинаний. Эти методы — не правила, а аналитические инструменты. Хороший рабочий знает предназначение каждого инструмента и умеет эффективно его использовать. Программисты не исключение. Чем больше вы узнаете о программировании, тем больше аналитических инструментов и знаний об их своевременном и правильном использовании накапливается в вашем интеллектуальном инструментарии.

Консультанты по вопросам разработки ПО иногда советуют программистам придерживаться одних методов разработки ПО в ущерб другим. Это печально, потому что, если вы станете использовать только одну методологию, вы увидите весь мир в терминах этой методологии. В некоторых случаях это сделает недоступными для вас другие методы, лучше подходящие для решения текущей проблемы. Метафора инструментария поможет вам держать все методы, способы и хитрости в пределах досягаемости и применять их в уместных обстоятельствах.

**Перекрестная ссылка** О выборе методов проектирования и их комбинировании см. раздел 5.3.

## Комбинирование метафор

Метафоры имеют эвристическую, а не алгоритмическую природу, поэтому они не исключают друг друга. Вы можете использовать и метафору аккреции, и метафору конструирования. Если хотите, можете представлять разработку ПО как написание письма, комбинируя эту метафору с вождением автомобиля, охотой на оборотней или образом динозавра, увязшего в смоляной луже. Используйте любые метафоры или их комбинации, которые стимулируют ваше мышление или помогают общаться с другими членами группы.

Использование метафор — дело тонкое. Чтобы метафора привела вас к ценным эвристическим догадкам, вы должны ее расширить. Но если ее расширить чересчур или в неверном направлении, она может ввести в заблуждение. Как и любой мощный инструмент, метафоры можно использовать неверным образом, однако благодаря своей мощи они могут стать ценным компонентом вашего интеллектуального инструментария.

# Дополнительные ресурсы

Среди книг общего плана, посвященных метафорам, моделям и парадигмам, главное место занимает «The Structure of Scientific Revolutions» (3d ed. Chicago, IL: The University of Chicago Press, 1996) Томаса Куна (Thomas S. Kuhn). В своей книге, увидевшей свет в 1962 г., Кун рассказывает о возникновении, развитии и смене теорий. Этот труд, вызвавший множество споров по вопросам философии науки, отличается ясностью, лаконичностью и включает массу интересных примеров взлетов и падений научных метафор, моделей и парадигм.

http://cc2e.com/0285

Статья «The Paradigms of Programming». 1978 Turing Award Lecture («Communications of the ACM», August 1979, pp. 455–60) Роберта У. Флойда (Robert W. Floyd) представляет собой увлекательное обсуждение использования моделей при разработке ПО; некоторые аспекты рассматриваются в ней в контексте идей Томаса Куна.

## Ключевые моменты

- Метафоры являются по природе эвристическими, а не алгоритмическими, поэтому зачастую они немного небрежны.
- Метафоры помогают понять процесс разработки ПО, сопоставляя его с другими, более знакомыми процессами.
- Некоторые метафоры лучше, чем другие.
- Сравнение конструирования ПО с возведением здания указывает на необходимость тщательной подготовки к проекту и проясняет различие между крупными и небольшими проектами.
- Аналогия между методами разработки ПО и инструментами в интеллектуальном инструментарии программиста наводит на мысль, что в распоряжении программистов имеется множество разных инструментов и что ни один инструмент не является универсальным. Выбор правильного инструмента — одно из условий эффективного программирования.
- Метафоры не исключают друг друга. Используйте комбинацию метафор, наиболее эффективную в вашем случае.

ГЛАВА 3

# Семь раз отмерь, один раз отрежь: предварительные условия

**Содержание**

http://cc2e.com/0309



**Связанные темы**

- Основные решения, которые приходится принимать при конструировании: глава 4
- Влияние размера проекта на предварительные условия и процесс конструирования ПО: глава 27
- Связь между качеством ПО и аспектами его конструирования: глава 20
- Управление конструированием ПО: глава 28
- Проектирование ПО: глава 5

Перед началом конструирования дома строители просматривают чертежи, проверяют, все ли разрешения получены, и исследуют фундамент. К сооружению небоскреба, жилого дома и собачьей конуры строители готовились бы по-разному, но, каким бы ни был проект, перед началом конструирования они провели бы добросовестную подготовку с учетом всех особенностей проекта.

В этой главе мы рассмотрим компоненты подготовки к конструированию ПО. Как и в строительстве, конечный успех программного проекта во многом определяется до начала конструирования. Если фундамент ненадежен или планирование выполнено небрежно, на этапе конструирования вы в лучшем случае сможете только свести вред к минимуму.

Популярная у плотников поговорка «семь раз отмерь, один раз отрежь» очень актуальна на этапе конструирования ПО, затраты на который иногда составляют аж 65% от общего бюджета проекта. В неудачных программных проектах конструирование иногда приходится выполнять дважды, трижды и даже больше. Как и в любой другой отрасли, повторение самой дорогостоящей части программного проекта ни к чему хорошему привести не может.

Хотя чтение этой главы является залогом успешного конструирования ПО, само конструирование в ней не обсуждается. Если вы уже хорошо разбираетесь в цикле разработки ПО или вам не терпится добраться до обсуждения конструирования, можете перейти к главе 5. Если вам не нравится идея выполнения предварительных условий конструирования, просмотрите раздел 3.2, чтобы узнать, какую роль они играют в вашем случае, а затем вернитесь к разделу 3.1, в котором описываются расходы, связанные с их невыполнением.

## 3.1. Важность выполнения предварительных условий

**Перекрестная ссылка** Повышение внимания к качеству ПО — самый эффективный способ повышения производительности труда программистов. (см. раздел 20.5).

Общей чертой всех программистов, создающих высококачественное ПО, является использование высококачественных методов, ставящих ударение на качестве ПО в самом начале, середине и конце проекта.

Если вы подчеркиваете качество в конце проекта, это приходится на этап тестирования системы. Именно о тестировании думают многие люди, представляя процесс гарантии качества ПО, но тестирование — только один из компонентов стратегии гарантии качества, и не самый важный. Тестирование не позволяет обнаружить такие ошибки, как создание не того приложения или создание нужного приложения не тем образом; эти ошибки должны быть определены и устранены раньше — до начала конструирования.

Если вы уделяете повышенное внимание качеству в середине работы над проектом, вы подчеркиваете методы конструирования, которым и посвящена большая часть этой книги.

Если вы подчеркиваете качество в начале проекта, вы качественно выполняете планирование, определение требований и проектирование. Если вы спроектировали автомобиль «Понтиак Ацтек», то сколько бы вы его ни тестировали, он никогда не превратится в «Роллс-Ройс». Вы можете создать самый лучший «Ацтек», но если вам нужен «Роллс-Ройс», это нужно планировать с самого начала. При разработке ПО такому планированию соответствуют определение проблемы и определение и проектирование решения.

Конструирование — средний этап работы, поэтому ко времени начала конструирования успех проекта уже частично предопределен. И все же во время конструирования вы хотя бы должны быть в состоянии определить, насколько благополучна ваша ситуация, и вернуться назад, если на горизонте показались черные тучи неудачи. В оставшейся части этой главы я подробно расскажу, почему адекватная подготовка к конструированию так важна и как определить, действительно ли вы готовы перейти к нему.

## Актуальны ли предварительные условия для современных программных проектов?

Выбор методологии не должен быть невежественным. Она должна быть основана на самом новом и эффективном и дополнена старым и заслуживающим доверия.

*Харлан Миллз (Harlan Mills)*

Порой говорят, что предварительные действия, такие как разработка архитектуры, проектирование и планирование проекта, в современных условиях бесполезны. Такие заявления не подтверждаются ни прошлыми, ни современными исследованиями (подробности см. ниже). Оппоненты предварительных условий обычно приводят примеры неудачного выполнения предварительных условий и делают вывод, что такая работа неэффективна. Тем не менее подготовку к конструированию можно выполнить успешно, и данные, накопленные с 1970-х, свидетельствуют о том, что в таких случаях работа над проектом оказывается эффективнее.

Общая цель подготовки — снижение риска: адекватное планирование позволяет исключить главные аспекты риска на самых ранних стадиях работы, чтобы основную часть проекта можно было выполнить максимально эффективно. Безусловно, главные факторы риска в создании ПО — неудачная выработка требований и плохое планирование проекта, поэтому подготовка направлена в первую очередь на оптимизацию этих этапов.

Так как подготовка к конструированию не является точной наукой, специфический подход к снижению риска будет в значительной степени определяться особенностями проекта (см. раздел 3.2).

## Причины неполной подготовки

Возможно, вам кажется, что все профессионалы знают о важности подготовки и всегда до начала конструирования проверяют выполнение предварительных условий. Увы, это не так.

**Дополнительные сведения** О профессиональной программе разработки ПО, поощряющей применение этих навыков, см. главу 16 книги «Professional Software Development» (McConnell, 2004).

http://cc2e.com/0316

Зачастую причина неполной подготовки к конструированию ПО объясняется тем, что отвечающие за нее разработчики не имеют нужного опыта. Для планирования проекта, создания адекватной бизнес-модели, разработки полных и точных требований и высококачественной архитектуры нужно обладать далеко не тривиальными навыками, однако большинство разработчиков этому не обучены. Если разработчики не знают, как выполнять предварительную работу, рекомендация «выполнять больше такой работы» не имеет смысла: если работа изначально выполняется некачественно, ее выполнение в *больших* объемах не принесет никакой пользы! Объяснение выполнения этих действий не является предметом данной книги, однако в разделе «Дополнительные ресурсы» в конце главы я привел массу источников, позволяющих получить такой опыт.

Некоторые программисты умеют готовиться к конструированию, но пренебрегают подготовкой, потому что не могут устоять перед искушением пораньше приступить к кодированию. Если вы принадлежите к их числу, могу дать два совета. Первый: прочитайте следующий раздел. Возможно, у вас откроются глаза на не-

которые вещи. Второй: уделяйте внимание проблемам, с которыми сталкиваетесь. Поработав над несколькими крупными программами, вы прекрасно поймете пользу заблаговременного планирования. Положитесь на свой опыт.

Наконец, еще одна причина пренебрежения подготовкой к конструированию состоит в том, что менеджеры прохладно относятся к программистам, которые тратят на это время. Это довольно странно: такие люди, как Барри Бом (Barry Boehm), Гради Буч (Grady Booch) и Карл Вигерс (Karl Wiegers), отстаивают важность выработки требований и проектирования уже 25 лет, и менеджеры, казалось бы, уже должны понимать, что разработка ПО не ограничивается кодированием, но...

**Дополнительные сведения** Ряд интересных вариаций на эту тему см. в классическом труде Джеральда Вайнберга «The Psychology of Computer Programming» (Weinberg, 1998).

Несколько лет назад я работал над проектом Минобороны, и как-то на этапе выработки требований нас посетил куратор проекта — генерал. Мы сказали ему, что работаем над требованиями: большей частью общаемся с клиентами, определяем их потребности и разрабатываем проект приложения. Он, однако, настаивал на том, чтобы увидеть код. Мы сказали, что у нас нет кода, и тогда он отправился в рабочий отдел, намереваясь хоть кого-нибудь из 100 человек поймать за программированием. Огорченный тем, что почти все из них находились не за своими компьютерами, этот крупный человек наконец указал на инженера рядом со мной и проревел: «А он что делает? Он ведь пишет код!» Вообще-то этот инженер работал над утилитой форматирования документов, но генерал хотел увидеть код, нашел что-то похожее на него и хотел, чтобы хоть кто-то писал код, так что мы сказали ему, что он прав: это код.

Этот феномен известен как синдром WISCA или WIMP: «Why Isn't Sam Coding Anything? (Почему Сэм не пишет код?)» или «Why Isn't Mary Programming (Почему Мэри не программирует)?»

Если менеджер проекта претендует на роль бригадного генерала и приказывает вам немедленно начать программировать, вы можете с легкостью ответить: «Есть, сэр!» (И впрямь, какое вам дело? Умудренные опытом ветераны должны отвечать за свои слова.) Это плохой ответ, и у вас есть несколько лучших вариантов. Во-первых, вы можете решительно отвергнуть неэффективную методику работы. Если у вас нормальные отношения с начальником и все в порядке с банковским счетом, это может сработать.

Во-вторых, вы можете притвориться, что работаете над кодом. Разложите на столе листинги старой программы и продолжайте работать над требованиями и архитектурой как ни в чем не бывало. Так вы выполните проект быстрее и качественнее. Порой этот подход находят неэтичным, но начальник-то останется доволен!

В-третьих, вы можете посвятить руководителя в нюансы технических проектов. Это хороший подход, потому что он увеличивает число грамотных руководителей в мире. В следующем подразделе приведено подробное обоснование важности выполнения предварительных условий до начала конструирования.

Наконец, вы можете найти другую работу. Независимо от экономических подъемов и спадов хороших программистов всегда не хватает (BLS, 2002), а жизнь слишком коротка, чтобы тратить ее на работу в отсталом учреждении при наличии множества лучших вариантов.

## Самый веский аргумент в пользу выполнения предварительных условий перед началом конструирования

Допустим, вы уже забрались на гору определения проблемы, прошли милю по пути выработки требований, сбросили грязную одежду у фонтана архитектуры и искупались в чистых водах подготовленности. Следовательно, вы знаете, что перед реализацией системы нужно понимать, что и как она будет делать.

Один из аспектов профессии разработчика — посвящение профанов в особенности процесса разработки ПО. Этот раздел поможет вам в общении с менеджерами и руководителями, еще блуждающих во тьме. В нем подробно описан веский аргумент в пользу адекватного определения требований и проектирования архитектуры до начала кодирования, тестирования и отладки. Изучите его, сядьте перед начальником и поговорите о процессе программирования по душам.

### Обращение к логике

Подготовка к проекту — одно из главных условий эффективного программирования, и это логично. Объем планирования зависит от масштаба проекта. С управленческой точки зрения, планирование подразумевает определение сроков, числа людей и компьютеров, необходимых для выполнения работ. С технической — планирование подразумевает получение представления о создаваемой системе, позволяющего не истратить деньги на создание неверной системы. Иногда пользователи не четко знают, что желают получить, и для определения их требований может понадобиться больше усилий, чем хотелось бы. Как бы то ни было, это дешевле, чем создать не то, что нужно, похерить результат и начать все заново.

До начала создания системы не менее важно подумать и о том, как вы собираетесь ее создавать. Никому не хочется тратить время и деньги на бесплодные блуждания по лабиринту.

### Обращение к аналогии

Создание программной системы похоже на любой другой проект, требующий людских и финансовых ресурсов. Возведение дома начинается не с забивания гвоздей, а с создания, анализа и утверждения чертежей. При разработке ПО наличие технического плана означает не меньше.

Никто не наряжает новогоднюю елку, не установив ее. Никто не разводит огонь, не открыв дымоход. Никто не отправляется в долгий путь с пустым бензобаком. Никто не принимает душ в одежде и не надевает носки после обуви. И т. д., и т. п.

Программисты — последнее звено пищевой цепи разработки ПО. Архитекторы поглощают требования, проектировщики потребляют архитектуру, а программисты — проект приложения.

Сравните пищевую цепь разработки ПО с реальной пищевой цепью. В экологически чистой среде водные жучки служат пищей рыбам, которыми в свою очередь питаются чайки. Это здоровая пищевая цепь. Если на каждом этапе разработки ПО у вас будет здоровая пища, результатом станет здоровый код, написанный довольными программистами.

Если среда загрязнена, жучки плавают в ядерных отходах, а рыба плещется в нефтяных пятнах. Чайкам не повезло больше всего: находясь в конце пищевой цепи, они травятся и нефтью, и ядерными отходами. Если ваши требования неудачны, они отравляют архитектуру, которая в свою очередь травит процесс конструирования. Результат? Раздражительные программисты и полное изъянов ПО.

При планировании в высокой степени итеративного проекта вы до начала конструирования должны определить важнейшие требования и архитектурные элементы, влияющие на каждый конструируемый фрагмент программы. Строителям, собирающимся строить поселок, не нужна полная информация о каждом доме до начала возведения первого дома, однако они должны исследовать место, составить план канализации и электрических линий и т. д. Если строители плохо подготовятся, канализационные трубы, возможно, придется проводить в уже построенный дом.

### Обращение к данным

Исследования последних 25 лет убедительно доказали выгоду правильного выполнения проектов с первого раза и дороговизну внесения изменений, которых можно было избежать.

Ученые из компаний Hewlett-Packard, IBM, Hughes Aircraft, TRW и других организаций обнаружили, что исправление ошибки к началу конструирования обходится в 10–100 раз дешевле, чем ее устранение в конце работы над проектом, во время тестирования приложения или после его выпуска (Fagan, 1976; Humphrey, Snyder, and Willis, 1991; Leffingwell 1997; Willis et al., 1998; Grady, 1999; Shull et al., 2002; Boehm and Turner, 2004).

Общий принцип прост: исправлять ошибки нужно как можно раньше. Чем дольше дефект сохраняется в пищевой цепи разработки ПО, тем больше вреда он приносит на следующих этапах. Так как раньше всего вырабатываются требования, ошибки, допущенные на этом этапе, присутствуют в системе дольше и обходятся дороже. Кроме того, дефекты, внесенные в систему раньше, оказывают более широкое влияние, чем дефекты, внесенные позднее. Это также повышает цену более ранних дефектов.

Вот данные об относительной дороговизне исправления дефектов в зависимости от этапов их внесения и обнаружения (табл. 3-1):

**Табл. 3-1. Средняя стоимость исправления дефектов в зависимости от времени их внесения и обнаружения**

| | Время обнаружения дефекта | | | | |
|---|---|---|---|---|---|
| **Время внесения дефекта** | **Выработка требований** | **Проектирование архитектуры** | **Конструирование** | **Тестирование системы** | **После выпуска ПО** |
| Выработка требований | 1 | 3 | 5–10 | 10 | 10–100 |
| Проектирование архитектуры | — | 1 | 10 | 15 | 25–100 |
| Конструирование | — | — | 1 | 10 | 10–25 |

**Источник:** адаптировано из работ «Design and Code Inspections to Reduce Errors in Program Development» (Fagan 1976), «Software Defect Removal» (Dunn, 1984), «Software Process Improvement at Hughes Aircraft» (Humphrey, Snyder, and Willis, 1991), «Calculating the Return on Investment from More Effective Requirements Management» (Leffingwell, 1997), «Hughes Aircraft's Widespread Deployment of a Continuously Improving Software Process» (Willis et al., 1998), «An Economic Release Decision Model: Insights into Software Project Management» (Grady, 1999), «What We Have Learned About Fighting Defects» (Shull et al., 2002) и «Balancing Agility and Discipline: A Guide for the Perplexed» (Boehm and Turner, 2004).

Эти данные говорят, например, о том, что дефект архитектуры, исправление которого при проектировании архитектуры обходится в $1000, может во время тестировании системы вылиться в $15 000 (рис. 3-1).

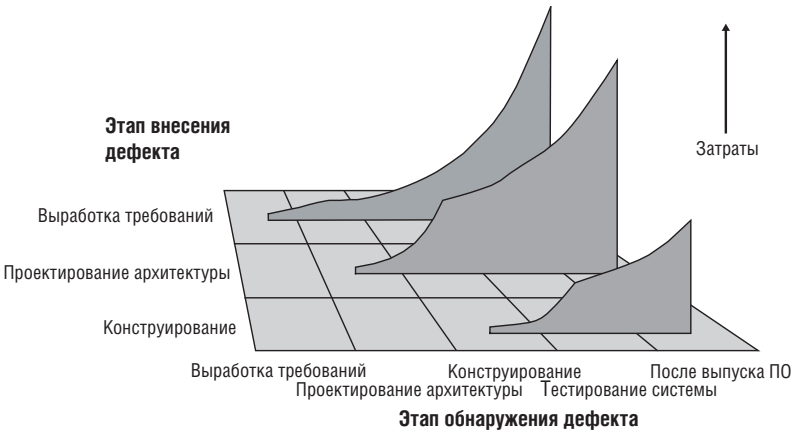



***Рис. 3-1.*** *С увеличением интервала между моментами внесения и обнаружения дефекта стоимость его исправления сильно возрастает. Это верно и для очень последовательных проектов (выработка требований и проектирование на 100% выполняются заблаговременно), и для очень итеративных (аналогичный показатель равен 5%)*

В большинстве проектов основная часть усилий по исправлению дефектов все еще приходится на правую часть рис. 3-1, а значит, на отладку и переделывание работы уходит около 50% времени типичного цикла разработки ПО (Mills, 1983; Boehm, 1987; Cooper and Mullen, 1993; Fishman, 1996; Haley, 1996; Wheeler, Brykczynski, and Meeson, 1996; Jones, 1998; Shull et al., 2002; Wiegers, 2002). В десятках компаний было обнаружено, что политика раннего исправления дефектов может в два и более раз снизить финансовые и временные затраты на разработку ПО (McConnell, 2004). Это очень веский довод в пользу как можно более раннего нахождения и решения проблем.

### Тест готовности руководителя

Если вам кажется, что ваш руководитель понимает важность выполнения предварительных условий до начала конструирования, попросите его пройти следующий тест, чтобы убедиться в этом.

Какие из этих утверждений являются самоисполняющимися пророчествами?

- Предлагаю приступить к кодированию прямо сейчас, потому что нам предстоит потратить много времени на отладку.
- Мы не выделили много времени на тестирование, поскольку не ожидаем обнаружить много дефектов.
- Мы изучили требования и проект приложения так хорошо, что я не представляю, какие крупные проблемы могут у нас возникнуть во время кодирования или отладки.

Все эти высказывания — самоисполняющиеся пророчества. Стремитесь к тому, чтобы исполнилось последнее.

Если вы еще не уверены, что предварительные условия актуальны для вашего проекта, следующий раздел поможет вам принять окончательное решение.

## 3.2. Определите тип ПО, над которым вы работаете

Обобщая 20 лет исследований разработки ПО, Кейперс Джонс, руководитель исследовательских работ в компании Software Productivity Research, заявил, что он и его коллеги сталкивались с 40 разными методами сбора требований, 50 вариантами проектирования ПО и 30 видами тестирования, применявшимися в проектах, реализуемых более чем на 700 языках программирования (Jones, 2003).

Разные типы проектов призывают к разным сочетаниям подготовки и конструирования. Каждый проект уникален, однако обычно проекты подпадают под общие стили разработки. Вот три самых популярных типа проектов, а также оптимальные в большинстве случаев методы работы над ними (табл. 3-2):

**Табл. 3-2. Оптимальные методы работы над программными проектами трех популярных типов**

| | Тип ПО | | |
|---|---|---|---|
| | **Бизнес-системы** | **Системы целевого назначения** | **Встроенные системы, от которых зависит жизнь людей** |
| **Типичные приложения** | Интернет-сайты.<br>Сайты в интрасетях.<br>Системы управления материально-техническим снабжением.<br>Игры.<br>Системы управления информацией.<br>Системы выплаты заработной платы. | Встроенное ПО.<br>Игры.<br>Интернет-сайты.<br>Пакетное ПО.<br>Программные инструменты.<br>Web-сервисы. | Авиационное ПО.<br>Встроенное ПО.<br>ПО для медицинских устройств.<br>Операционные системы.<br>Пакетное ПО. |
| **Модели жизненного цикла** | Гибкая разработка (экстремальное программирование, методология Scrum, разработка на основе временных окон и т. д.).<br>Эволюционное. прототипирование. | Поэтапная поставка.<br>Эволюционная поставка.<br>Спиральная разработка. | Поэтапная поставка.<br>Спиральная разработка.<br>Эволюционная поставка. |
| **Планирование и управление** | Инкрементное планирование проекта.<br>Планирование тестирования и гарантии качества по мере надобности.<br>Неформальный контроль над изменениями. | Базовое заблаговременное планирование.<br>Базовое планирование тестирования.<br>Планирование гарантии качества по мере надобности.<br>Формальный контроль над изменениями. | Исчерпывающее заблаговременное планирование.<br>Исчерпывающее планирование тестирования.<br>Исчерпывающее планирование гарантии качества.<br>Тщательный контроль над изменениями. |
| **Выработка требований** | Неформальная спецификация требований. | Полуформальная спецификация требований.<br>Обзоры требований по мере надобности. | Формальная спецификация требований.<br>Формальные инспекции требований. |

(*см. след. стр.*)

**Табл. 3-2.** (*окончание*)

| | Тип ПО | | |
|---|---|---|---|
| **от которых зависит** | **Бизнес-системы** | **назначения** | **Встроенные системы, Системы целевого жизнь людей** |
| **Проектирование** | Комбинация проектирования и кодирования. | Проектирование архитектуры. Неформальное детальное проектирование. Обзоры проекта по мере надобности. | Проектирование архитектуры. Формальные инспекции архитектуры. Формальное детальное проектирование. Формальные инспекции детального проекта. |
| **Конструирование** | Парное или индивидуальное программирование. Неформальная процедура регистрации кода или ее отсутствие. | Парное или индивидуальное программирование. Неформальная процедура регистрации кода. Обзоры кода по мере надобности. | Парное или индивидуальное программирование. Формальная процедура регистрации кода. Формальные инспекции кода. |
| **Тестирование и гарантия качества** | Разработчики тестируют собственный код. Предварительная разработка тестов. Тестирование отдельной группой проводится в малом объеме или не проводится вообще. | Разработчики тестируют собственный код. Предварительная разработка тестов. Отдельная группа тестирования. | Разработчики тестируют собственный код. Предварительная разработка тестов. Отдельная группа тестирования. Отдельная группа гарантии качества. |
| **Внедрение приложения** | Неформальная процедура внедрения. | Формальная процедура внедрения. | Формальная процедура внедрения. |

Работая над реальными проектами, вы столкнетесь с бесчисленными вариациями на три темы, указанные в таблице, однако можно сделать и некоторые общие выводы. При разработке бизнес-систем предпочтительно использовать высокоитеративные подходы, при которых планирование, выработка требований и проектирование архитектуры перемежаются с конструированием, тестированием системы и гарантией качества. Системы, от которых зависит жизнь людей, требуют более последовательных подходов, поскольку стабильность требований — одно из условий высочайшей надежности системы.

## Влияние итеративных подходов на предварительные условия

Кое-кто утверждает, что при использовании итеративных методов не нужно особо возиться с предварительными условиями, но эта точка зрения неверна. Итера-

тивные подходы ослабляют следствия неадекватной подготовки, но не устраняют их. Давайте изучим табл. 3-3, в которой приведены данные о проектах, в начале которых не были выполнены предварительные условия. Первый проект выполняется последовательно, при этом дефекты обнаруживаются только на этапе тестирования; второй выполняется итеративно, и разработчики находят дефекты по мере работы. В первом случае основной объем работы по исправлению дефектов откладывается на конец проекта, что приводит к росту расходов (табл. 3-1). При итеративном подходе дефекты исправляются по мере развития проекта, что позволяет снизить общие расходы. Табл. 3-3 и 3-4 приведены исключительно в иллюстративных целях, однако соотношение затрат при этих двух общих подходах хорошо подтверждается исследованием, описанным выше.

**Табл. 3-3. Влияние невыполнения предварительных условий на последовательный и итеративный проекты**

| | **Подход 1: последовательный подход без выполнения предварительных условий** | | **Подход 2: итеративный подход без выполнения предварительных условий** | |
|---|---|---|---|---|
| Степень завершенности проекта | Затраты на работу | Затраты на исправление дефектов | Затраты на работу | Затраты на исправление дефектов |
| 20% | $100 000 | $0 | $100 000 | $75 000 |
| 40% | $100 000 | $0 | $100 000 | $75 000 |
| 60% | $100 000 | $0 | $100 000 | $75 000 |
| 80% | $100 000 | $0 | $100 000 | $75 000 |
| 100% | $100 000 | $0 | $100 000 | $75 000 |
| Затраты на исправление дефектов в конце проекта | $0 | $500 000 | $0 | $0 |
| СУММА | $500 000 | $500 000 | $500 000 | $375 000 |
| ОБЩАЯ СУММА | $1 000 000 | | $875 000 | |

Итеративный проект с сокращенной программой выполнения предварительных условий или без нее отличается от аналогичного последовательного проекта двумя аспектами. Во-первых, при итеративном подходе затраты на исправление дефектов обычно ниже, потому что дефекты выявляются раньше. И все же это происходит в конце каждой итерации, и исправление дефектов требует повторного проектирования, кодирования и тестирования фрагментов ПО, что делает затраты более крупными, чем они могли бы быть.

Во-вторых, при итеративных подходах затраты распределяются по всему проекту, а не группируются в его конце. В конце концов и при итеративных, и при последовательных подходах общая сумма затрат окажется похожей, но в первом случае она не будет казаться столь крупной, потому что будет уплачена по частям.

Адекватное внимание к выполнению предварительных условий позволяет снизить затраты независимо от типа используемого подхода (табл. 3-4). Итеративные подходы обычно по многим параметрам лучше последовательных, однако итеративный подход, при котором пренебрегают предварительными условиями, может в итоге оказаться гораздо дороже, чем должным образом подготовленный последовательный проект.

**Табл. 3-4. Влияние выполнения предварительных условий на последовательный и итеративный проекты**

| | **Подход 3: последовательный подход с выполнением предварительных условий** | | **Подход 4: итеративный подход с выполнением предварительных условий** | |
|---|---|---|---|---|
| Степень завершенности проекта | Затраты на работу | Затраты на исправление дефектов | Затраты на работу | Затраты на исправление дефектов |
| 20% | $100 000 | $20 000 | $100 000 | $10 000 |
| 40% | $100 000 | $20 000 | $100 000 | $10 000 |
| 60% | $100 000 | $20 000 | $100 000 | $10 000 |
| 80% | $100 000 | $20 000 | $100 000 | $10 000 |
| 100% | $100 000 | $20 000 | $100 000 | $10 000 |
| Затраты на исправление дефектов в конце проекта | $0 | $0 | $0 | $0 |
| СУММА | $500 000 | $100 000 | $500 000 | $50 000 |
| ОБЩАЯ СУММА | $600 000 | | $550 000 | |

Эти данные наводят на мысль, что большинство проектов не являются ни полностью последовательными, ни полностью итеративными. Определять требования или выполнять проектирование на 100% наперед непрактично, однако обычно определение самых важных требований и архитектурных элементов на раннем этапе оказывается выгодным.

**Перекрестная ссылка** Об адаптации подхода разработки к программам разных размеров см. главу 27.

Одно популярное практическое правило состоит в том, чтобы заблаговременно определить около 80% требований, предусмотреть время для более позднего определения дополнительных требований и выполнять по мере работы систематичный контроль изменений, принимая только самые важные требования. Возможен и другой вариант: вы можете определить заранее 20% только самых важных требований и разрабатывать оставшуюся часть ПО небольшими фрагментами, определяя дополнительные требования и дорабатывая проект приложения по мере прогресса (рис. 3-2 и 3-3).

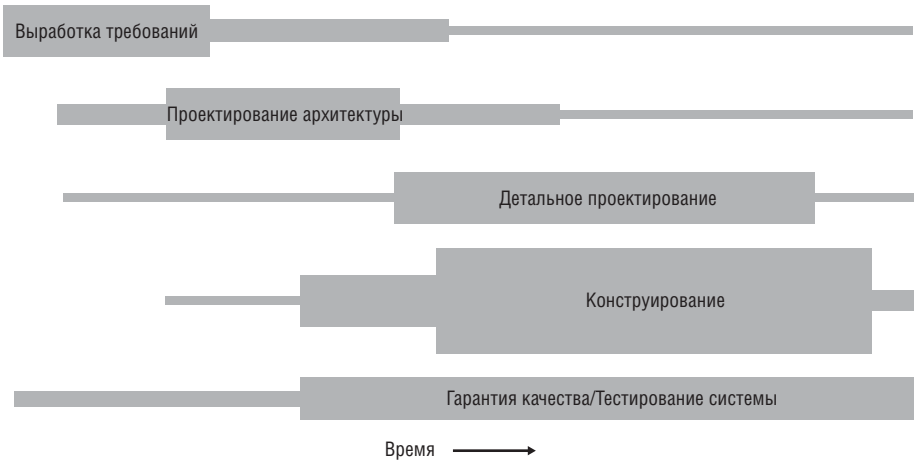



***Рис. 3-2.*** *Этапы работы над проектами — даже самыми последовательными — обычно несколько перекрываются*


***Рис. 3-3.*** *В других случаях этапы перекрываются на всем протяжении проекта. Понимание степени выполнения предварительных условий и соответствующая адаптация проекта — одно из условий успешного конструирования*

## Выбор между итеративным и последовательным подходом

Степень, в которой предварительные условия должны быть выполнены наперед, зависит от типа проекта (табл. 3-2), формальности проекта, технической среды, возможностей сотрудников и бизнес-модели проекта. Вы можете выбрать более последовательный подход (при котором вопросы решаются заблаговременно), если:

- требования довольно стабильны;
- проект приложения прост и относительно понятен;
- группа разработчиков знакома с прикладной областью;
- проект не связан с особым риском;

- важна долговременная предсказуемость проекта;
- затраты на изменение требований, проекта приложения и кода скорее всего окажутся высокими.

Более итеративный подход (при котором вопросы решаются по мере работы) можно предпочесть, если:

- требования относительно непонятны или вам кажется, что они могут оказаться нестабильными по другим причинам;
- проект приложения сложен, не совсем ясен или и то и другое;
- группа разработчиков незнакома с прикладной областью;
- проект сопряжен с высоким риском;
- долговременная предсказуемость проекта не играет особой роли;
- затраты на изменение требований, проекта приложения и кода скорее всего будут низкими.

Как бы то ни было, итеративные подходы эффективны гораздо чаще, чем последовательные. Вы можете адаптировать предварительные условия к своему конкретному проекту, как считаете нужным, сделав их более или менее формальными или полными. Мы подробнее обсудим разные подходы к крупным и небольшим (или формальным и неформальным) проектам в главе 27.

Суть предварительных условий конструирования в том, что вам следует сначала определить, какие из них уместны для вашего проекта. В некоторых проектах предварительным условиям уделяется слишком мало времени, что приводит к дестабилизации на этапе конструирования и препятствует планомерному развитию проекта, хотя этого можно было избежать. В других проектах слишком много работы выполняется наперед; программисты, работающие над такими проектами, слепо следуют требованиям и планам, которые впоследствии оказываются неудачными, и это также может снижать эффективность конструирования.

Итак, изучив табл. 3-2, вы определили, какие предварительные условия уместны для вашего проекта, поэтому в оставшейся части главы я расскажу, как определить, были ли выполнены отдельные предварительные условия конструирования.

## 3.3. Предварительные условия, связанные с определением проблемы

> Если ограничения и условия описываются как «коробка», то хитрость в том, чтобы найти именно коробку… Не думайте о чем-то глобальном — найдите коробку.
>
> *Энди Хант и Дэйв Томас (Andy Hunt and Dave Thomas)*

Первое предварительное условие, которое нужно выполнить перед конструированием, — ясное формулирование проблемы, которую система должна решать. Это еще иногда называют «видением продукции», «формулированием точки зрения», «формулированием задачи» или «определением продукции». Я буду называть это условие «определением проблемы». Так как книга посвящена конструированию программ, прочитав этот раздел, вы не научитесь разрабатывать определение проблемы, но узнаете, как определить, есть ли оно вообще и станет ли оно надежной основой конструирования.

Определение проблемы — это просто формулировка сути проблемы без каких-либо намеков на ее возможные решения. Оно может занимать пару страниц, но обязательно должно звучать как проблема. Фраза «наша система Gigatron не справляется с обработкой заказов» звучит как проблема и является хорошим ее определением. Однако фраза «мы должны оптимизировать модуль автоматизированного ввода данных, чтобы система Gigatron справлялась с обработкой заказов» — плохое определение проблемы. Оно похоже не на проблему, а на решение.

Определение проблемы предшествует выработке детальных требований, которая является более глубоким исследованием проблемы (рис. 3-4).


***Рис. 3-4.*** *Определение проблемы — фундамент всего процесса программирования*

Проблему следует формулировать на языке, понятном пользователю, а сама проблема должна быть описана с пользовательской точки зрения. Обычно проблему не следует формулировать в компьютерных терминах, потому что оптимальным ее решением может оказаться не компьютерная программа. Допустим, вы нуждаетесь в вычислении годовой прибыли. Вычисление квартальной прибыли вы уже компьютеризировали. Если вы застрянете на программировании, то решите, что в имеющееся приложение нужно просто добавить функцию вычисления годовой прибыли со всеми вытекающими отсюда последствиями: затратами на оплату труда разработчиков и т. п. Если же вы малость подумаете, то просто чуть поднимете зарплату секретарю, который будет один раз в год суммировать четыре числа на калькуляторе.

Исключения из этого правила допустимы, если проблема связана с компьютерами: программы компилируются слишком медленно, или инструменты программирования полны ошибок. В подобных случаях проблему можно сформулировать в компьютерных, привычных для программистов терминах.

Не имея хорошего определения проблемы, можно потратить усилия на решение не той проблемы (рис. 3-5).

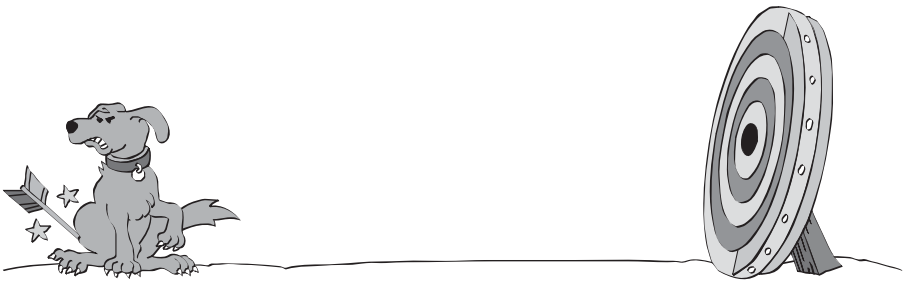


***Рис. 3-5.*** *Прежде чем стрелять, убедитесь в том, что знаете, куда целитесь*

Неудачное определение проблемы грозит пустой тратой времени на решение не той проблемы. Разумеется, нужную проблему вы при этом тоже не решите.

## 3.4. Предварительные условия, связанные с выработкой требований

Требования подробно описывают, что должна делать программная система, а их выработка — первый шаг к решению проблемы. Выработка требований также известна как «разработка требований», «анализ требований», «анализ», «определение требований», «спецификация», «функциональная спецификация».

### Зачем нужны официальные требования?

Важность явного набора требований объясняется несколькими причинами.

Явные требования помогают гарантировать, что функциональность системы определяется пользователем, а не программистом. Если требования сформулированы явно, пользователь может проанализировать и утвердить их. Если явных требований нет, программисту обычно самому приходится вырабатывать их во время программирования. Явные требования позволяют не гадать, чего хочет пользователь.

Кроме того, наличие явных требований помогает избегать споров. Требования позволяют определить функциональность системы до начала программирования. Если вы не согласны с другим программистом по поводу каких-то аспектов программы, вы можете разрешить споры, взглянув на написанные требования.

Внимание к требованиям помогает свести к минимуму изменения системы после начала разработки. Обнаружив при кодировании ошибку в коде, вы измените несколько строк, и работа продолжится. Если же во время кодирования вы найдете ошибку в требованиях, придется изменить проект программы, чтобы он соответствовал измененным требованиям. Возможно, при этом придется отказаться от части старого проекта, а поскольку в соответствии с ней уже написан некоторый код, на реализацию нового проекта уйдет больше времени, чем могло бы. Вы также должны будете отказаться от кода и тестов, на которые повлияло изменение требований, и написать их заново. Даже код, оставшийся нетронутым, нужно будет заново протестировать для гарантии того, что изменение не привело к появлению новых ошибок.

Как вы помните, исследования, проведенные во многих организациях, свидетельствуют о том, что при работе над крупными проектами исправление ошибки в требованиях, обнаруженной на этапе проектирования архитектуры, обычно обходится втрое дороже, чем исправление аналогичной ошибки, найденной на этапе выработки требований (табл. 3-1). Такая же ошибка, обнаруженная при кодировании, обходится дороже уже в 5–10, при тестировании системы — в 10, а после выпуска программы — в 10–100 раз. В менее крупных проектах с меньшими административными расходами последний показатель ближе к 5–10, чем к 100 (Boehm and Turner, 2004). Как бы то ни было, думаю, что дополнительные расходы нужны вам меньше всего.

Адекватное определение требований — одно из важнейших условий успеха проекта, возможно, даже более важное, чем использование эффективных методов конструирования (рис. 3-6). Определению требований посвящены многие хорошие книги, поэтому в нескольких следующих разделах я не буду рассказывать, как правильно определять требования, — вместо этого я расскажу, как узнать, хорошо ли они определены, и как выжать максимум из имеющихся требований.

***Рис. 3-6.*** *Не имея грамотно определенных требований, вы можете правильно представлять общую проблему, но упустить из виду ее специфические аспекты*

## Миф о стабильных требованиях

> Требования подобны воде. Опираться на них легче, если они заморожены.
>
> *Аноним*

Стабильные требования — Святой Грааль разработки ПО. При стабильных требованиях смена этапов разработки архитектуры, проектирования, кодирования и тестирования приложения происходит упорядоченно, предсказуемо и спокойно. Это просто рай для разработчиков! Вы можете точно планировать расходы и совсем не волнуетесь о том, что реализация какой-то функции обойдется в 100 раз дороже из-за того, что клиенту она придет в голову только после отладки.

Всем нам хотелось бы надеяться, что, как только клиент утвердил требования, никаких изменений не произойдет. Однако чаще всего клиент не может точно сказать, что ему нужно, пока не будет написан некоторый код. Проблема не в том, что клиенты — более низкая форма жизни. Подумайте: чем больше вы работаете над проектом, тем лучше вы его понимаете; то же относится и к клиентам. Про-

цесс разработки помогает им лучше понять собственные потребности, что часто приводит к изменению требований (Curtis, Krasner and Iscoe, 1988; Jones 1998; Wiegers, 2003). Если вы планируете жестко следовать требованиям, на самом деле вы собираетесь не реагировать на потребности клиента.

Какой объем изменений типичен? Исследования, проведенные в IBM и других компаниях, показали, что при реализации среднего проекта требования во время разработки изменяются примерно на 25% (Boehm, 1981; Jones, 1994; Jones, 2000), на что приходится 70–85% объема повторной работы над типичным проектом (Leffingwell, 1997; Wiegers, 2003).

Возможно, вы считаете, что «Понтиак Ацтек» — самый великолепный автомобиль из когда-либо созданных, являетесь членом Общества Верящих в Плоскую Землю и каждые четыре года совершаете паломничество в Розуэлл, штат Нью-Мексико, на место приземления инопланетян. Если это так, можете и дальше верить в то, что требования в ваших проектах меняться не будут. Если же вы уже перестали верить в Санта-Клауса или хотя бы прекратили признаваться в этом, вы можете кое-что предпринять, чтобы свести зависимость от изменений требований к минимуму.

## Что делать при изменении требований во время конструирования программы?

Следующие действия позволяют максимально легко перенести изменения требований во время конструирования.

***Оцените качество требований при помощи контрольного списка, приведенного в конце раздела*** Если требования недостаточно хороши, прекратите работу, вернитесь назад и исправьте их. Конечно, вам может показаться, что прекращение кодирования на этом этапе приведет к отставанию от графика. Но если вы едете из Чикаго в Лос-Анджелес и видите знаки, указывающие путь в Нью-Йорк, разве можно считать изучение карты пустой тратой времени? Нет. Если вы не уверены в правильности выбранного направления, остановитесь и проверьте его.

***Убедитесь, что всем известна цена изменения требований*** Думая о новой функции, клиенты приходят в возбуждение. Кровь у них разжижается, переполняет продолговатый мозг, и они впадают в эйфорию, забывая обо всех собраниях, посвященных обсуждению требований, о церемонии подписания и всех документах. Угомонить таких одурманенных новыми функциями людей проще всего, заявив: «Ого, это действительно прекрасная идея! Но ее нет в документе требований, поэтому я должен пересмотреть график работы и смету, чтобы вы могли решить, хотите ли вы реализовать это прямо сейчас или позднее». Слова «график» и «смета» отрезвляют куда лучше, чем кофе и холодный душ, и многие требования быстро превращаются в пожелания.

Если руководители вашей организации не понимают важность предварительной выработки требований, укажите им, что изменения во время выработки требований обходятся гораздо дешевле, чем на более поздних этапах. Используйте для их убеждения раздел «Самый веский аргумент в пользу выполнения предварительных условий перед началом конструирования».

***Задайте процедуру контроля изменений*** Если клиент никак не может успокоиться, подумайте об установлении стенда контроля изменений для рассмотрения вносимых предложений. В том, что клиенты изменяют точку зрения и понимают, что им нужны дополнительные возможности, нет ничего аномального. Проблема в том, что они вносят предложения так часто, что вы не поспеваете за ними. Наличие процедуры контроля изменений осчастливит всех: вы будете знать, что вам придется работать с изменениями только в определенные периоды времени, а клиенты увидят, что вам небезразличны их пожелания.

**Перекрестная ссылка** О том, что делать с изменениями проекта приложения и самого кода, см. раздел 28.2.

***Используйте те подходы к разработке, которые адаптируются к изменениям*** Некоторые подходы к разработке ПО позволяют особенно легко реагировать на изменения требований. Подход эволюционного прототипирования (evolutionary prototyping) помогает исследовать требования к системе до начала ее создания. Эволюционная поставка ПО подразумевает предоставление системы пользователям по частям. Вы можете создать фрагмент системы, получить от пользователей отзывы, немного подкорректировать проект, внести несколько изменений и приступить к созданию другого фрагмента. Суть этого подхода — короткие циклы разработки, позволяющие быстро реагировать на пожелания пользователей.

**Перекрестная ссылка** Об итеративных подходах к разработке см. подраздел «Используйте итерацию» раздела 5.4 и раздел 29.3.

***Оставьте проект*** Если требования особенно неудачны или изменчивы и никакой из предыдущих советов не работает, завершите проект. Даже если вы не можете на самом деле завершить его, подумайте об этом. Подумайте о том, насколько хуже он должен стать, чтобы вы от него отказались. Если такая ситуация возможна, спросите себя, чем она отличается от текущей ситуации.

**Дополнительные сведения** О подходах к разработке, поддерживающих гибкие требования, см. книгу «Rapid Development» (McConnell, 1996).

***Помните о бизнес-модели проекта*** Многие проблемы с требованиями исчезают при воспоминании о коммерческих предпосылках проекта. Требования, которые сначала казались прекрасными идеями, могут оказаться ужасными, когда вы оцените затраты. Программисты, которые принимают во внимание коммерческие следствия своих решений, ценятся на вес золота, и я был бы рад получить свою комиссию за этот совет.

**Перекрестная ссылка** О различиях между формальными и неформальными проектами (которые часто объясняются различиями размеров проектов) см. главу 27.

### Контрольный список: требования

http://cc2e.com/0323

Следующий контрольный список содержит вопросы, позволяющие определить качество требований. Ни книга, ни этот список не научат вас грамотно вырабатывать требования. Используйте его во время конструирования для определения того, насколько прочна земля, на которой вы стоите.

Не все вопросы будут актуальны для вашего проекта. Если вы работаете над неформальным проектом, над некоторыми вопросами даже не нужно задумываться. Другие вопросы важны, но не требуют формальных ответов. Однако если

вы работаете над крупным формальным проектом, вам, наверное, следует ответить на каждый вопрос.

**Специфические функциональные требования**

- ❑ Определены ли все способы ввода данных в систему с указанием источника, точности, диапазона значений и частоты ввода?
- ❑ Определены ли все способы вывода данных системой с указанием назначения, точности, диапазона значений, частоты и формата?
- ❑ Определены ли все форматы вывода для Web-страниц, отчетов и т. д.?
- ❑ Определены ли все внешние аппаратные и программные интерфейсы?
- ❑ Определены ли все внешние коммуникационные интерфейсы с указанием протоколов установления соединения, проверки ошибок и коммуникации?
- ❑ Определены ли все задачи, в выполнении которых нуждается пользователь?
- ❑ Определены ли данные, используемые в каждой задаче, и данные, являющиеся результатом выполнения каждой задачи?

**Специфические нефункциональные требования (требования к качеству)**

- ❑ Определено ли ожидаемое пользователем время реакции для всех необходимых операций?
- ❑ Определены ли другие временные параметры, такие как время обработки данных, скорость их передачи и пропускная способность системы?
- ❑ Определен ли уровень защищенности системы?
- ❑ Определена ли надежность системы, в том числе такие аспекты, как следствия сбоев в ее работе, информация, которая должна быть защищена от сбоев, и стратегия обнаружения и исправления ошибок?
- ❑ Определены ли минимальные требования программы к объему памяти и свободного дискового пространства?
- ❑ Определены ли аспекты удобства сопровождения системы, в том числе способность системы адаптироваться к изменениям специфических функций, ОС и интерфейсов с другими приложениями?
- ❑ Включено ли в требования определение успеха? Или неудачи?

**Качество требований**

- ❑ Написаны ли требования на языке, понятном пользователям? Согласны ли с этим пользователи?
- ❑ Нет ли конфликтов между требованиями?
- ❑ Определено ли приемлемое равновесие между параметрами-антагонистами, такими как устойчивость к нарушению исходных предпосылок и корректность?
- ❑ Не присутствуют ли в требованиях элементы проектирования?
- ❑ Согласован ли уровень детальности требований? Следует ли какое-нибудь требование определить подробнее? Менее подробно?
- ❑ Достаточно ли ясны и понятны требования, чтобы их можно было передать независимой группе конструирования? Согласны ли с этим разработчики?
- ❑ Каждое ли требование релевантно для проблемы и ее решения? Можно ли проследить каждое требование до его источника в проблемной среде?

- ❑ Можно ли протестировать каждое требование? Можно ли будет провести независимое тестирование, которое позволит сказать, выполнены ли все требования?
- ❑ Определены ли все возможные изменения требований и вероятность каждого изменения?

**Полнота требований**

- ❑ Указаны ли недостающие требования, которые невозможно определить до начала разработки?
- ❑ Полны ли требования в том смысле, что если приложение будет удовлетворять всем требованиям, то оно будет приемлемо?
- ❑ Не вызывают ли какие-нибудь требования у вас дискомфорта? Исключили ли вы требования, которые не поддаются реализации и были включены лишь для успокоения клиента или начальника?

## 3.5. Предварительные условия, связанные с разработкой архитектуры

**Перекрестная ссылка** О проектировании на всех уровнях см. главы 5–9.

Архитектура — это высокоуровневая часть проекта приложения, каркас, состоящий из деталей проекта (Buschman et al., 1996; Fowler, 2002; Bass Clements, Kazman 2003; Clements et al., 2003). Архитектуру также называют «архитектурой системы», «высокоуровневым проектом» и «проектом высокого уровня». Как правило, архитектуру описывают в единственном документе, называемом «спецификацией архитектуры» или «высокоуровневым проектом». Некоторые разработчики проводят различие между архитектурой и высокоуровневым проектом: архитектурой называют характеристики всей системы, тогда как высокоуровневым проектом — характеристики, описывающие подсистемы или наборы классов, но не обязательно в масштабе всей системы.

Так как эта книга посвящена конструированию, прочитав этот раздел, вы не узнаете, как разрабатывать архитектуру ПО, — вы научитесь определять качество имеющейся архитектуры. Однако разработка архитектуры на один шаг ближе к конструированию, чем выработка требований, поэтому архитектуру мы рассмотрим подробнее, чем требования.

Почему разработку архитектуры следует рассматривать как предварительное условие конструирования? Потому что качество архитектуры определяет концептуальную целостность системы, которая в свою очередь определяет итоговое качество системы. Продуманная архитектура предоставляет структуру, нужную для поддержания концептуальной целостности в масштабе системы. Она предоставляет программистам руководство, уровень детальности которого соответствует их навыкам и выполняемой работе. Она позволяет разделить работу на части, над которыми отдельные разработчики и группы могут трудиться независимо.

Хорошая архитектура облегчает конструирование. Плохая архитектура делает его почти невозможным. Другую проблему, связанную с плохой архитектурой, иллюстрирует рис. 3-7.

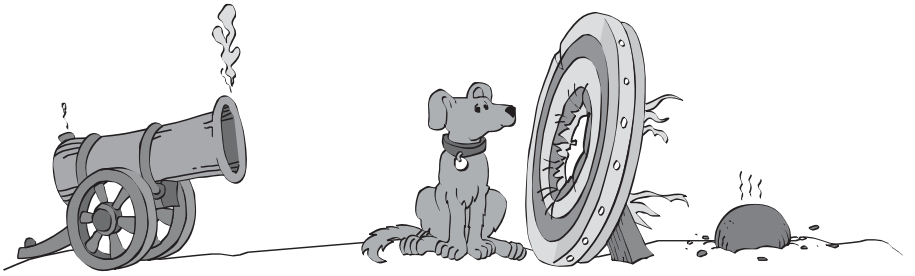


***Рис. 3-7.*** *Не имея хорошей архитектуры, вы можете решать правильную проблему, но прийти к неправильному решению. Успешное конструирование может оказаться невозможным*

Внесение изменений в архитектуру при конструировании и на последующих этапах обходится недешево. Время, необходимое для исправления ошибки в архитектуре ПО, сопоставимо со временем, нужным для исправления ошибки в требованиях, т. е. превышает временные затраты на исправление ошибки в коде (Basili and Perricone, 1984; Willis, 1998). Изменения архитектуры похожи на изменения требований еще и тем, что кажущиеся небольшими изменения могут иметь далеко идущие последствия. Чем бы ни были обусловлены изменения архитектуры — исправлением ошибок или внесением улучшений, — чем раньше вы осознаете их необходимость, тем лучше.

## Типичные компоненты архитектуры

**Перекрестная ссылка** О низкоуровневом проектировании программы см. главы 5–9.

Удачные архитектуры имеют много общего. Если всю систему вы создаете самостоятельно, работа над архитектурой будет перекрываться с более детальным проектированием. В этом случае вам следует по крайней мере обдумать каждый компонент архитектуры. Если вы работаете над системой, архитектура которой была разработана кем-то другим, вы должны уметь определять ее важные компоненты без охотничьей собаки и увеличительного стекла. Как бы то ни было, далее приведен список компонентов архитектуры, на которые следует обратить внимание.

### Организация программы

Если вы не можете объяснить что-то шестилетнему ребенку, значит, вы сами этого не понимаете.

*Альберт Эйнштейн*

В первую очередь архитектура должна включать общее описание системы. Без такого описания вам будет трудно составить согласованную картину из тысячи деталей или хотя бы десятка отдельных классов. Если бы система была мозаикой из 12 фрагментов, ее мог бы с легкостью собрать и годовалый ребенок. Головоломку из 12 подсистем собрать труднее, но, если вы не сможете сделать этого, вы не поймете, какой вклад вносит в систему разрабатываемый вами класс.

Архитектура должна включать подтверждения того, что при ее разработке были рассмотрены альтернативные варианты, и обосновывать выбор окончательной организации системы. Никому не хочется разрабатывать класс, если его роль в системе не кажется хорошо обдуманной. Описывая альтернативные варианты,

архитектура обосновывает организацию системы и показывает, что роль каждого класса была тщательно рассмотрена. В одном обзоре методик проектирования было обнаружено, что обоснование проекта программы не менее важно для ее сопровождения, чем сам проект (Rombach, 1990).

Архитектура должна определять основные компоненты программы. В зависимости от размера программы ее компонентами могут быть отдельные классы или подсистемы, состоящие из нескольких классов. Каждый компонент является классом или набором классов/методов, которые в совокупности реализуют высокоуровневые функции программы, такие как взаимодействие с пользователем, отображение Web-страниц, интерпретация команд, инкапсуляция бизнес-правил или доступ к данным. За каждую функцию приложения, указанную в требованиях, должен отвечать хотя бы один компонент. Если функцию реализуют несколько компонентов, они должны сотрудничать, а не конфликтовать.

**Перекрестная ссылка** Об используемых при проектировании компонентах разных уровней см. подраздел «Уровни проектирования» раздела 5.2.

Архитектура должна четко определять ответственность каждого компонента. Компонент должен иметь одну область ответственности и как можно меньше знать об областях ответственности других компонентов. Сведя к минимуму объем сведений, известных компонентам о других компонентах, вы сможете локализовать информацию о проекте приложения в отдельных компонентах.

**Перекрестная ссылка** Минимизация объема сведений, известных компонентам друг о друге, — главный аспект сокрытия информации. Подробности см. в подразделе «Скрывайте секреты (к вопросу о сокрытии информации)» раздела 5.3.

Архитектура должна ясно определять правила коммуникации для каждого компонента. Она должна описывать, какие другие компоненты данный компонент может использовать непосредственно, какие косвенно, а какие вообще не должен использовать.

### Основные классы

Архитектура должна определять основные классы приложения, их области ответственности и механизмы взаимодействия с другими классами. Она должна описывать иерархии классов, а также изменения состояний и время существования объектов. Если система достаточно велика, архитектура должна описывать организацию классов в подсистемы.

**Перекрестная ссылка** О проектировании классов см. главу 6.

Архитектура должна описывать другие рассматривавшиеся варианты организации классов и обосновывать итоговый вариант. Не все классы системы нужно описывать в спецификации архитектуры. Ориентируйтесь на правило 80/20: описывайте 20% классов, которыми на 80% определяется поведение системы (Jacobsen, Booch, and Rumbaugh, 1999; Kruchten, 2000).

### Организация данных

Архитектура должна описывать основные виды формата файлов и таблиц. Она должна описывать рассмотренные альтернативы и обосновывать итоговые варианты. Если приложение использует список идентификаторов клиентов и разработчики архитектуры решили реализовать его при помощи списка с после-

**Перекрестная ссылка** Об использовании переменных см. главы 10–13.

довательным доступом, в документации должно быть сказано, почему этот вид списка лучше, чем список с произвольным доступом, стек или хэш-таблица. Эта информация окажет вам неоценимую помощь во время конструирования и сопровождения программы, подсказав, чем руководствовались разработчики архитектуры. Без нее вы будете чувствовать себя зрителем, который смотрит иностранный фильм без субтитров.

Прямой доступ к данным обычно следует предоставлять только одной подсистеме или классу; исключения возможны при использовании классов или методов доступа, обеспечивающих доступ к данным, контролируемым абстрактным образом. Подробнее об этом см. подраздел «Скрывайте секреты (к вопросу о сокрытии информации)» раздела 5.3.

Архитектура должна определять высокоуровневую организацию и содержание всех используемых БД. Архитектура должна объяснять, почему одна БД предпочтительнее, чем несколько (или наоборот), почему БД предпочтительнее, чем однородные файлы, определять возможные типы взаимодействия приложения с другими программами, использующими те же данные, объяснять, как будут отображаться данные, и т. д.

### Бизнес-правила

Архитектура, зависимая от специфических бизнес-правил, должна определять их и описывать их влияние на проект системы. Возьмем для примера бизнес-правило, согласно которому информация о клиентах должна устаревать не более чем на 30 секунд. В данном случае в спецификации архитектуры должно быть указано, как это правило повлияло на выбор метода обеспечения актуальности данных и их синхронизации.

### Пользовательский интерфейс

Пользовательский интерфейс (GUI) часто проектируется на этапе выработки требований. Если это не так, его следует определить на этапе разработки архитектуры. Архитектура должна описывать главные элементы формата Web-страниц, GUI, интерфейс командной строки и т. д. Удобство GUI может в итоге определить популярность или провал программы.

Архитектура должна быть модульной, чтобы GUI можно было изменить, не затронув бизнес-правил и модулей программы, отвечающих за вывод данных. Например, архитектура должна обеспечивать возможность сравнительно легкой замены группы классов интерактивного интерфейса на группу классов интерфейса командной строки. Такая возможность весьма полезна; во многом это объясняется тем, что интерфейс командной строки удобен для тестирования ПО на уровне блоков или подсистем.

http://cc2e.com/0393

Проектирование GUI заслуживает отдельной книги, и мы его рассматривать не будем.

### Управление ресурсами

Архитектура должна включать план управления ограниченными ресурсами, такими как соединения с БД, потоки и дескрипторы. При разработке драйверов, встроенных систем и других приложений, которые будут работать в условиях ограни-

ченной памяти, архитектура должна также определять способ управления памятью. Архитектура должна включать оценку ресурсов, используемых в номинальном режиме и при экстремальной нагрузке. В простейшем случае эти оценки должны подтвердить, что предполагаемая среда использования приложения будет располагать нужными ресурсами. В более сложной ситуации в приложении, возможно, придется реализовать более активное управление выделенными ему ресурсами. Если это так, архитектуру менеджера ресурсов нужно спроектировать не менее тщательно, чем любой другой компонент системы.

### Безопасность

Архитектура должна определять подход к безопасности на уровне проекта приложения и на уровне кода. Если модель угроз до сих пор не разработана, это следует сделать при проектировании архитектуры. О безопасности нужно помнить и при разработке принципов кодирования, в том числе методик обработки буферов и ненадежных данных (данных, вводимых пользователями, файлов «cookie», конфигурационных данных и данных других внешних интерфейсов), подходов к шифрованию, уровню подробности сообщений об ошибках, защите секретных данных, находящихся в памяти, и другим вопросам.

http://cc2e.com/0330

**Дополнительные сведения** Прекрасное обсуждение защиты ПО см. в книге «Writing Secure Code, 2d Ed.» (Howard and LeBlanc 2003) и в январском номере журнала «IEEE Software» за 2002 год.

### Производительность

В требованиях следует определить показатели производительности. Если они связаны с использованием ресурсов, надо определить приоритеты для разных ресурсов, в том числе соотношение быстродействия, использования памяти и затрат.

**Дополнительные сведения** О проектировании высокопроизводительных систем см. книгу Конни Смит «Performance Engineering of Software Systems» (Smith, 1990).

Архитектура должна включать оценки производительности и объяснять, почему разработчики архитектуры считают эти показатели достижимыми. Если они могут быть не достигнуты, это тоже должно быть отражено в архитектуре. Если для достижения некоторых показателей требуются специфические алгоритмы или типы данных, также укажите это в спецификации архитектуры. Кроме того, в архитектуре можно указать объем пространства и время, выделяемые каждому классу или объекту.

### Масштабируемость

Масштабируемостью называют возможность системы адаптироваться к росту требований. Архитектура должна описывать, как система будет реагировать на рост числа пользователей, серверов, сетевых узлов, записей в БД, транзакций и т. д. Если развитие системы не предполагается и ее масштабируемость не играет роли, это должно быть явно указано в архитектуре.

### Взаимодействие с другими системами

Если некоторые данные или ресурсы будут общими для разрабатываемой системы и других программ или устройств, в архитектуре нужно указать, как это будет реализовано.

### Интернационализация/локализация

«Интернационализацией» называют реализацию в программе поддержки региональных стандартов. Вместо слова «internationalization» часто используется аббревиатура «I18n», составленная из первой и последней букв слова и числа букв между ними. «Локализацией» (известной как «L10n» по той же причине) называют перевод интерфейса программы и реализацию в ней поддержки конкретного языка.

Вопросы интернационализации заслуживают особого внимания при разработке архитектуры интерактивной системы. Большинство интерактивных систем включает десятки или сотни подсказок, индикаторов состояния, вспомогательных сообщений, сообщений об ошибках и т. д., поэтому нужно оценить объем ресурсов, используемых строками. Если разрабатывается коммерческая программа, архитектура должна показывать, что при ее создании были рассмотрены типичные вопросы, связанные со строками и наборами символов, такие как выбор набора символов (ASCII, DBCS, EBCDIC, MBCS, Unicode, ISO 8859 и т. д.) и типа строк (строки C, строки Visual Basic и т. д.), а также способа изменения строк, который не требовал бы изменения кода, и метода перевода строк на иностранные языки, оказывающего минимальное влияние на код и GUI. Строки можно встроить в код, инкапсулировать в класс и использовать посредством интерфейса или сохранить в файле ресурсов. Архитектура должна объяснять, какой вариант выбран и почему.

### Ввод-вывод

Ввод-вывод — еще одна область, на которую стоит обратить внимание при проектировании архитектуры. Архитектура должна определять схему чтения данных: упреждающее чтение, чтение с задержкой или по требованию. Кроме того, она должна описывать уровень, на котором будут определяться ошибки ввода-вывода: на уровне полей, записей, потоков данных или файлов.

### Обработка ошибок

Обработка ошибок — одна из самых сложных проблем современной информатики, и к ней нельзя относиться с пренебрежением. По оценкам некоторых ученых код на целых 90% состоит из блоков обработки исключительных ситуаций, ошибок и т. п., из чего следует, что только 10% кода отвечают за номинальный режим работы программы (Shaw in Bentley, 1982). Раз уж на обработку ошибок приходится такая большая часть кода, стратегия их согласованной обработки должна быть выражена в архитектуре.

Обработку ошибок часто рассматривают на уровне конвенции кодирования, если вообще рассматривают. Однако она оказывает влияние на всю систему, поэтому лучше всего рассматривать ее на уровне архитектуры. Вот некоторые вопросы, на которые нужно обратить внимание.

- Является ли обработка ошибок корректирующей или ориентированной на их простое обнаружение? В первом случае программа может попытаться восстановиться от последствий ошибки. Во втором — может продолжить работу как ни в чем не бывало или завершиться. Как бы то ни было, она должна известить пользователя об ошибке.

- Является ли обнаружение ошибок активным или пассивным? Система может активно предвосхищать ошибки (например, проверяя корректность данных, введенных пользователем) или пассивно реагировать на них только в том случае, если избежать их не удалось (например, когда введенные пользователем данные привели к численному переполнению). В обоих случаях сделанный выбор повлияет на дизайн GUI.
- Как программа поступает при обнаружении ошибки? Обнаружив ошибку, программа может отбросить данные, вызвавшие ошибку, может отнестись к ошибке как положено и перейти в состояние обработки ошибки или может выполнить оставшиеся действия и уведомить пользователя о том, что (где-то) были обнаружены ошибки.
- Каковы соглашения обработки сообщений об ошибках? Если в спецификации архитектуры не определена единственная согласованная стратегия, GUI покажется непонятной комбинацией разных интерфейсов, относящихся к разным частям программы. Чтобы избежать этого, при проектировании архитектуры нужно определить соглашения вывода сообщений об ошибках.
- Как обрабатываются исключения? Архитектура должна определять, когда код может генерировать исключения, где они будут перехватываться, как регистрироваться в журнале, документироваться и т. д.
- На каком уровне программы обрабатываются ошибки? Вы можете обрабатывать их в точке обнаружения, передавать классу обработки ошибок или возвращать по цепи вызовов.
- Какова ответственность каждого класса за проверку получаемых данных? Каждый класс отвечает за проверку собственных данных или есть группа классов, проверяющих данные для всей системы? Могут ли классы конкретного уровня полагать, что полученные ими данные корректны?
- Хотите ли вы использовать механизм обработки ошибок, встроенный в среду программирования, или создать собственный? Если в среде реализован конкретный подход к обработке ошибок, это не значит, что он лучше всего соответствует вашим требованиям.

**Перекрестная ссылка** Еще один аспект стратегии обработки ошибок, который следует рассмотреть на архитектурном уровне, — согласованный метод обработки недопустимых параметров. Примеры см. в главе 8.

### Отказоустойчивость

При разработке архитектуры системы следует указать ожидаемый уровень ее отказоустойчивости. Отказоустойчивостью называют совокупность свойств системы, повышающих ее надежность путем обнаружения ошибок, восстановления, если это возможно, и изоляции их плохих последствий, если восстановление невозможно.

Например, вычисление системой квадратного корня можно сделать отказоустойчивым несколькими способами.

- В случае неудачи система может вернуться в предыдущее состояние и попробовать вычислить корень еще раз. Если первый ответ неверен, она может вернуться в состояние, при котором все наверняка было правильно, и продолжить работу с этого момента.

**Дополнительные сведения** Хорошее введение в вопросы отказоустойчивости см. в июльском номере журнала «IEEE Software» за 2001 год. Кроме того, в статьях этого номера есть ссылки на многие отличные книги и статьи, посвященные данной теме.

- Система может включать вспомогательный код, выполняемый при обнаружении ошибки в основном коде. Так, если первый ответ кажется ошибочным, система может переключиться на альтернативный метод вычисления квадратного корня.
- Система может применять алгоритм голосования. Она может включать три класса, вычисляющих квадратный корень разными способами. Каждый класс вычисляет квадратный корень, после чего система сравнивает полученные результаты. В зависимости от реализованного типа отказоустойчивости система может использовать среднее, срединное или наиболее вероятное значение из трех.
- Система может заменять ошибочное значение поддельным значением, которое положительно скажется на работе оставшейся части системы.

Другими подходами к отказоустойчивости являются перевод системы при обнаружении ошибки в состояние частичной работоспособности или ограниченной фунциональности. Система может отключиться или автоматически перезапустить себя. Конечно, эти примеры упрощены. Отказоустойчивость — захватывающая и сложная область, но не она является темой этой книги.

### Возможность реализации архитектуры

Разработчики могут сомневаться в том, способна ли система достигнуть заданных показателей производительности, работать при ограниченности ресурсов и будет ли она адекватно поддержана средами реализации. Архитектура должна подтверждать, что система технически осуществима. Если невозможность реализации какого-то компонента может сделать проект неработоспособным, в архитектуре должно быть отражено, как изучались эти вопросы: при помощи прототипов, исследований или иначе. Эти аспекты риска следует устранить до начала полномасштабного конструирования.

### Избыточная функциональность

Надежностью называют способность системы продолжать работу после обнаружения ошибки. Частенько в спецификации архитектуры разработчики определяют более надежную систему, чем указано в требованиях. Одна из причин этого в том, что система, состоящая из многих частей, удовлетворяющих минимальным требованиям к надежности, в целом может оказаться менее надежной, чем нужно. В мире ПО цепь не так крепка, как слабейшее звено; она так слаба, как все слабые звенья, вместе взятые. В спецификации архитектуры должно быть явно указано, могут ли программисты реализовать в своих блоках программы избыточную функциональность или они должны создать простейшую работоспособную систему.

Определить отношение к реализации избыточной функциональности особенно важно потому, что многие программисты делают это автоматически, из чувства профессиональной гордости. Явно выразив ожидания в архитектуре, вы сможете избежать феномена, при котором некоторые классы исключительно надежны, а другие лишь отвечают требованиям.

### Купить или создавать самим?

Самый радикальный подход к созданию ПО — не создавать его вообще, а купить или загрузить из Интернета бесплатное ПО с открытым исходным кодом. Вы можете приобрести элементы управления GUI, менеджеры БД, процессоры изображений, компоненты для работы с графикой и диаграммами, компоненты для коммуникации по Интернету, компоненты обеспечения безопасности и шифрования, обработки электронных таблиц и текста — список почти бесконечен. Одним из главных достоинств программирования с использованием современных GUI-сред является объем функциональности, который вы получаете автоматически: классы для работы с графикой, менеджеры диалоговых окон, обработчики событий клавиатуры и мыши, код, поддерживающий любые принтеры и мониторы и т. д.

**Перекрестная ссылка** Список типов коммерческих программных компонентов см. в подразделе «Библиотеки кода» раздела 30.3.

Если архитектура не подразумевает применение готовых компонентов, она должна объяснять, в каких аспектах компоненты, которые будут разработаны, окажутся лучше готовых библиотек и компонентов.

### Повторное использование

Если план предусматривает применение существующего кода, тестов, форматов данных и т. д., архитектура должна объяснять, как повторно использованные ресурсы будут адаптированы к другим архитектурным особенностям, если это будет сделано.

### Стратегия изменений

Так как при создании продукта и программисты, и пользователи обучаются, приложение скорее всего в период разработки будет изменяться. Причинами этого могут быть изменения типов данных, форматов файлов, функциональности, реализация новых функций и т. д. Изменения могут быть новыми возможностями, которые были запланированы заранее или не были реализованы в первой версии системы. Поэтому разработчику архитектуры ПО следует сделать ее достаточно гибкой, чтобы в систему можно было легко внести вероятные изменения.

**Перекрестная ссылка** О систематичной обработке изменений см. раздел 28.2.

Архитектура должна четко описывать стратегию изменений. Архитектура должна показывать, что возможные улучшения рассматривались и что реализация наиболее вероятных улучшений окажется наиболее простой. Если вероятны изменения форматов ввода или вывода данных, стиля взаимодействия с пользователями или требований к обработке, архитектура должна показывать, что все эти изменения были предвосхищены и каждое из них будет ограничено небольшим числом классов. Архитектурный план внесения изменений может быть совсем простым: включить в файлы данных номера версий, зарезервировать поля на будущее, спроектировать файлы так, чтобы в них можно было добавить новые таблицы и т. д. Если применяется генератор кода, архитектура должна показывать, что он поддерживает возможность внесения предполагаемых изменений.

> Ошибки проектирования часто являются довольно тонкими и объясняются эволюцией, при которой по мере реализации новых функций и возможностей разработчики забывают о сделанных ранее предположениях.
>
> *Фернандо Дж. Корбати (Ferrnando J. Corbaty)*

**Перекрестная ссылка** О мерах, позволяющих не ограничивать возможность выбора, см. подраздел «Тщательно выбирайте время связывания» раздела 5.3.

В архитектуре должны быть отражены стратегии, которые позволяют программистам не ограничивать имеющийся у них выбор раньше времени. Так, архитектура может определять, что вместо жестко закодированных тестов *if* будет применяться метод, основанный на проверке таблиц. Данные таблиц можно хранить во внешнем файле, а не включать в программу, что позволит вносить в нее изменения без перекомпиляции.

### Общее качество архитектуры

Хорошая спецификация архитектуры должна описывать классы системы, информацию, скрываемую каждым классом, и обосновывать принятые и отвергнутые варианты проекта системы.

**Перекрестная ссылка** О соотношении атрибутов качества см. раздел 20.1.

Архитектура должна быть продуманным концептуальным целым, включающим несколько специфических дополнений. Главный тезис самой популярной книги по разработке ПО «Мифический человеко-месяц» гласит, что основной проблемой, характерной для крупных систем, является поддержание их концептуальной целостности (Brooks, 1995). Хорошая архитектура должна соответствовать проблеме. Изучая архитектуру, вы должны испытывать удовольствие от того, насколько естественным и простым кажется решение. Вам должно казаться, что проблема и архитектура неразрывно связаны.

Вам следует знать, как архитектура изменялась во время ее проектирования. Каждое изменение должно быть четко согласовано с общей концепцией. Архитектура не должна быть похожа на проект бюджета Конгресса США, предусматривающий расходы на мероприятия, повышающие популярность чиновников.

Цели архитектуры должны быть четко сформулированы. Проект системы, главным требованием к которой является модифицируемость, будет отличаться от проекта системы, которая должна показывать высочайшую производительность, даже если по функциональности обе системы будут одинаковы.

В архитектуре должны быть обоснованы важнейшие принятые решения. С подозрением относитесь к обоснованиям из разряда «мы всегда так делали». Здесь уместно вспомнить одну поучительную историю. Бет хотела приготовить тушеное мясо по прославленному рецепту, передававшемуся из поколения в поколение в семье ее мужа Абдула. Абдул сказал, что его мать солила кусок мяса, перчила, обрезала его края, укладывала в горшок, закрывала и ставила в духовку. На вопрос Бет «Зачем обрезать оба края?» Абдул ответил: «Не знаю, я всегда так делал. Спрошу у мамы». Он позвонил ей и услышал: «Не знаю, просто я так всегда делала. Спрошу у твоей бабушки». А бабушка заявила: «Понятия не имею, почему вы так делаете. Я делала так потому, что мой горшок был маловат».

Хорошая архитектура ПО не зависит ни от платформы, ни от языка. Пожалуй, вы не сможете проигнорировать среду конструирования, однако максимальная независимость от среды позволит вам устоять перед соблазном создать слишком подробную архитектуру и избежать работы, которую лучше выполнять во время конструирования. Если программа ориентирована на конкретную платформу или должна быть разработана на конкретном языке, это правило неактуально.

При разработке архитектуры следует соблюдать баланс между недостаточным и чрезмерным определением системы. Ни на какую часть архитектуры не следует обращать больше внимания, чем она заслуживает; не следует разрабатывать одну часть в ущерб другой. Архитектура должна отражать все требования, не включая ненужных элементов.

В архитектуре должны быть явно определены области риска, указаны его причины и описаны действия по сведению риска к минимуму.

Архитектура должна включать описания системы с разных точек зрения. Планы дома включают поэтажный план, план перекрытий, электрические схемы и т. д. Качество архитектуры ПО также повысится, если включить в нее описания разных взглядов на систему, которые позволят найти ошибки и помогут программистам полностью понять проект системы (Kruchten, 1995).

Наконец, элементы архитектуры не должны вызывать у вас чувство неловкости. В архитектуру не следует включать что-то только для того, чтобы угодить начальнику. Архитектура не должна включать ничего, что было бы трудно понять. Именно вы будете претворять ее в жизнь — как же вы ее реализуете, если не будете в ней разбираться?

### Контрольный список: архитектура

http://cc2e.com/0337

Следующий список вопросов позволяет сделать вывод о качестве архитектуры. Этот список не является исчерпывающим руководством по проектированию архитектуры — это прагматичный способ оценки того, что вы получаете на программистском конце пищевой цепи разработки ПО. Используйте его как основу для создания собственного контрольного списка. Как и в случае аналогичного списка вопросов о требованиях, при работе над неформальным проектом некоторые вопросы будут неактуальны, однако при работе над более крупным проектом большинство из них пригодится.

**Специфические аспекты архитектуры**

- ❑ Ясно ли описана общая организация программы? Включает ли спецификация грамотный обзор архитектуры и ее обоснование?
- ❑ Адекватно ли определены основные компоненты программы, их области ответственности и взаимодействие с другими компонентами?
- ❑ Все ли функции, указанные в спецификации требований, реализуются разумным, не слишком большим и не слишком малым, числом компонентов?
- ❑ Приведено ли описание самых важных классов и их обоснование?
- ❑ Приведено ли описание организации данных и ее обоснование?
- ❑ Приведено ли описание организации и содержания БД?
- ❑ Определены ли все важные бизнес-правила? Описано ли их влияние на систему?
- ❑ Описана ли стратегия проектирования GUI?
- ❑ Сделан ли GUI модульным, чтобы его изменения не влияли на оставшуюся часть программы?
- ❑ Приведено ли описание стратегии ввода-вывода данных и ее обоснование?

- ❑ Указаны ли оценки степени использования ограниченных ресурсов, таких как потоки, соединения с БД, дескрипторы, пропускная способность сети? Приведено ли описание стратегии управления такими ресурсами и ее обоснование?
- ❑ Описаны ли требования к защищенности архитектуры?
- ❑ Определяет ли архитектура требования к объему и быстродействию всех классов, подсистем и функциональных областей?
- ❑ Описывает ли архитектура способ достижения масштабируемости системы?
- ❑ Рассмотрены ли вопросы взаимодействия системы с другими системами?
- ❑ Описана ли стратегия интернационализации/локализации?
- ❑ Определена ли согласованная стратегия обработки ошибок?
- ❑ Определен ли подход к отказоустойчивости системы (если это требуется)?
- ❑ Подтверждена ли возможность технической реализации всех частей системы?
- ❑ Определен ли подход к реализации избыточной функциональности?
- ❑ Приняты ли необходимые решения относительно «покупки или создания» компонентов системы?
- ❑ Описано ли в спецификации, как повторно используемый код будет адаптирован к другим аспектам архитектуры?
- ❑ Сможет ли архитектура адаптироваться к вероятным изменениям?

**Общее качество архитектуры**

- ❑ Все ли требования отражены в архитектуре?
- ❑ Является ли какая-нибудь часть системы чрезмерно или недостаточно проработанной? Заданы ли явные ожидания по этому поводу?
- ❑ Является ли вся архитектура концептуально целостной?
- ❑ Независим ли высокоуровневый проект системы от платформы и языка, который будет использован для его реализации?
- ❑ Указаны ли мотивы принятия всех основных решений?
- ❑ Удовлетворяет ли вас — программиста, который будет реализовывать систему, — разработанная архитектура?

## 3.6. Сколько времени следует посвятить выполнению предварительных условий?

**Перекрестная ссылка** Объем времени, необходимый для работы над предварительными условиями, зависит от типа проекта. Об адаптации предварительных условий к специфическому проекту см. раздел 3.2.

Время, уходящее на определение проблемы, выработку требований и проектирование архитектуры ПО, зависит от особенностей проекта. Как правило, если проект развивается без проблем, работа над требованиями, архитектурой и предварительным планированием поглощает 10–20% усилий и 20–30% времени (McConnell, 1998; Kruchten, 2000). Эти показатели не включают затраты на детальное проектирование — оно является частью конструирования.

Если требования нестабильны и вы работаете над крупным формальным проектом, то для решения проблем с требованиями, которые будут обнаружены на ранних этапах конструирования, вам, вероятно, понадобятся услуги специалиста по анализу требований. Предусмотрите консультации с ним и выделите ему время на ревизию требований — и пригодная для работы версия требований в ваших руках.

Если требования нестабильны и вы работаете над небольшим неформальным проектом, вам, наверное, придется решать проблемы с требованиями самостоятельно. Выделите время на грамотное определение требований, чтобы их изменчивость как можно слабее повлияла на конструирование.

Каким бы проект ни был — формальным или неформальным, — если требования нестабильны, рассматривайте работу над ними как отдельный проект. Оцените время, нужное для выполнения оставшейся части проекта, после завершения работы над требованиями. Это разумно: трудно ожидать, что вы сможете составить график работы до того, как будете знать, что создаете. Представьте, что вас хотят нанять для строительства дома. Заказчик говорит: «Сколько будет стоить ваша работа?» Вы обоснованно спрашиваете: «Что мне нужно построить?», на что заказчик отвечает: «Я не могу сказать вам, но сколько это будет стоить?» Что вы сделаете? Попрощаетесь с заказчиком и отправитесь домой.

**Перекрестная ссылка** О подходах к изменениям требований см. подраздел «Что делать при изменении требований во время конструирования программы?» раздела 3.4.

Ясно, что клиенты не попросят вас предъявить им счет, пока не расскажут, какой дом надо построить. Им не нужно, чтобы вы пришли с досками, молотком и гвоздями и начали тратить их деньги до того, как архитектор завершит работу над чертежами. Однако люди, как правило, разбираются в создании ПО хуже, чем в лестницах и дверных проемах, поэтому ваши клиенты могут не сразу понять, почему вы хотите сделать выработку требований отдельным проектом. Возможно, вам придется объяснить им причины этого.

Выделяя время на проектирование архитектуры ПО, поступайте так же, как и при выработке требований. Если над данным типом ПО вы еще не работали, выделите больше времени. Убедитесь, что время на создание качественной архитектуры будет выделено не в ущерб другим этапам. Если нужно, сделайте отдельным проектом и работу над архитектурой.

# Дополнительные ресурсы

Ниже я привел список ресурсов, посвященных работе над требованиями.

http://cc2e.com/0344

### Выработка требований

http://cc2e.com/0351

В следующих книгах вы найдете гораздо более подробную информацию о выработке требований.

Wiegers, Karl. *Software Requirements*, 2d ed. Redmond, WA: Microsoft Press, 2003. В этом практическом руководстве описываются все детали выработки требований, в том числе сбор информации о требованиях, их анализ, составление спецификации требований, проверка требований и управление ими.

Robertson, Suzanne and James Robertson. *Mastering the Requirements Process*. Reading, MA: Addison-Wesley, 1999. Хорошая альтернатива книге Карла Вигерса, ориентированная на более подготовленных специалистов по выработке требований.

Gilb, Tom. *Competitive Engineering*. Reading, MA: Addison-Wesley, 2004. В этой книге рассматривается язык требований Гил-

http://cc2e.com/0358

ба, известный как «Planguage». Кроме того, в ней описывается специфический подход Гилба к разработке требований, проектированию, оценке проектирования и эволюционному управлению проектом. Загрузить книгу можно с Web-сайта Тома Гилба по адресу *www.gilb.com*.

*IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications*. Los Alamitos, CA: IEEE Computer Society Press. Этот документ представляет собой руководство IEEE-ANSI по созданию спецификаций требований к ПО. В нем описываются элементы, которые следует включать в документ спецификации, и рассматриваются некоторые альтернативные варианты.

http://cc2e.com/0365

Abran, Alain, et al. *Swebok: Guide to the Software Engineering Body of Knowledge*. Los Alamitos, CA: IEEE Computer Society Press, 2001. В этом руководстве приведено подробное описание выработки требований к ПО. Загрузить его можно с Web-сайта *www.swebok.org*.

Ниже указаны хорошие альтернативы названным книгам.

Lauesen, Soren. *Software Requirements: Styles and Techniques*. Boston, MA: Addison-Wesley, 2002.

Kovitz, Benjamin L. *Practical Software Requirements: A Manual of Content and Style*. Manning Publications Company, 1998.

Cockburn, Alistair. *Writing Effective Use Cases*. Boston, MA: Addison-Wesley, 2000.

### Разработка архитектуры

http://cc2e.com/0372

В последние несколько лет было опубликовано много книг, посвященных разработке архитектуры ПО. Одними из лучших я считаю следующие.

Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*, 2d ed. Boston, MA: Addison-Wesley, 2003.

Buschman, Frank, et al. *Pattern-Oriented Software Architecture*, Volume 1: A System of Patterns. New York, NY: John Wiley & Sons, 1996.

Clements, Paul, ed. *Documenting Software Architectures: Views and Beyond*. Boston, MA: Addison-Wesley, 2003.

Clements, Paul, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Boston, MA: Addison-Wesley, 2002.

Fowler, Martin. «Patterns of Enterprise Application Architecture». Boston, MA: Addison-Wesley, 2002.

Jacobson, Ivar, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999.

*IEEE Std 1471-2000. Recommended Practice for Architectural Description of Software-Intensive Systems*. Los Alamitos, CA: IEEE Computer Society Press. Этот документ является руководством IEEE-ANSI по созданию спецификаций архитектуры ПО.

### Общие подходы к разработке ПО

Издано много книг, посвященных разным подходам к выполнению программных проектов. В одних рассматриваются более последовательные подходы, в других — более итеративные.

http://cc2e.com/0379

McConnell, Steve. *Software Project Survival Guide*. Redmond, WA: Microsoft Press, 1998. В этой книге рассмотрен один конкретный способ выполнения проекта, подчеркивающий обдуманное заблаговременное планирование, выработку требований и работу над архитектурой, за которыми следует тщательное выполнение проекта. Такой подход обеспечивает долговременную предсказуемость финансовых и временных затрат, позволяет создавать высококачественное ПО и характеризуется умеренной гибкостью.

Kruchten, Philippe. *The Rational Unified Process: An Introduction*, 2d ed. Reading, MA: Addison-Wesley, 2000. В этой книге представлен «архитектурно-центрический и определяемый моделью использования» подход к выполнению проектов. Как и в «Software Project Survival Guide», здесь особое внимание уделяется предварительным действиям, обеспечивающим высокую долговременную предсказуемость финансовых и временных затрат, умеренную гибкость работы и способствуют созданию высококачественного ПО. В некоторых аспектах этот подход сложнее, чем описанные в «Software Project Survival Guide» и «Extreme Programming Explained: Embrace Change».

Jacobson, Ivar, Grady Booch and James Rumbaugh. *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999. Здесь представлено более глубокое обсуждение тем, рассматриваемых в «The Rational Unified Process: An Introduction», 2d ed.

Beck, Kent. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley, 2000. Бек описывает высокоитеративный подход, который фокусируется на итеративной разработке требований к приложению и его проектов в сочетании с конструированием. Подход «экстремального программирования» обладает невысокой долговременной предсказуемостью, но обеспечивает высокую гибкость.

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley, 1988. Подход Гилба предусматривает исследование главных вопросов планирования, выработки требований и проектирования архитектуры на ранних этапах проекта и последующую непрерывную адаптацию планов проекта по мере прогресса. Этот подход характеризуется долговременной предсказуемостью и высокой гибкостью, а создаваемые на его основе приложения отличаются высоким качеством. Он сложнее подходов, описанные в «Software Project Survival Guide» и «Extreme Programming Explained: Embrace Change».

McConnell, Steve. *Rapid Development*. Redmond, WA: Microsoft Press, 1996. В этой книге описан инструментальный подход к планированию проекта. Используя представленные в книге инструменты, опытный специалист по планированию проектов сможет создать план, прекрасно адаптированный к уникальным особенностям проекта.

Boehm, Barry and Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA: Addison-Wesley, 2003. В этой книге исследуется контраст между гибкой разработкой и разработкой, основанной на планировании. Особенно

интересны четыре раздела главы 3: «A Typical Day using PSP/TSP», «A Typical Day using Extreme Programming», «A Crisis Day using PSP/TSP» и «A Crisis Day using Extreme Programming». Глава 5 посвящена использованию рискованных подходов с целью уравновешивания гибкости, что может служить руководством по выбору между гибким методом или методом, основанным на планировании. В главе 6 приводится хорошо сбалансированная перспектива. Приложение E включает подробные опытные данные о гибких методах разработки.

Larman, Craig. *Agile and Iterative Development: A Manager's Guide*. Boston, MA: Addison Wesley, 2004. Это основанное на тщательных исследованиях введение в гибкие эволюционные стили разработки включает обзор подходов Scrum, Extreme Programming, Unified Process и Evo.

**Контрольный список: предварительные условия**

http://cc2e.com/0386

- ❑ Установили ли вы тип проекта, над которым работаете, и адаптировали ли вы к нему свой подход?
- ❑ Достаточно ли хорошо определены и достаточно ли стабильны требования для начала конструирования? (См. контрольный список вопросов о требованиях).
- ❑ Достаточно ли хорошо определена архитектура для начала конструирования? (См. контрольный список вопросов об архитектуре).
- ❑ Рассмотрели ли вы другие, уникальные для конкретного проекта факторы риска, чтобы они не снижали эффективность конструирования?

# Ключевые моменты

- Главной целью подготовки к конструированию является снижение риска. Убедитесь, что проводимая вами подготовка снижает риск, а не повышает его.
- Если вы хотите разрабатывать высококачественное ПО, внимание к качеству должно быть частью процесса разработки ПО с начала до конца. Внимание к качеству в начале процесса оказывает наибольшее влияние на итоговое качество приложения.
- Одним из аспектов профессии программиста является объяснение руководителям и коллегам процесса разработки ПО, в том числе важности адекватной подготовки к программированию.
- Предварительные условия конструирования в большой степени зависят от типа проекта, над которым вы работаете: многие проекты призывают к использованию высокоитеративного подхода, другие — более последовательного.
- При отсутствии грамотного определения проблемы вы можете на этапе конструирования потратить силы на решение неверной проблемы.
- Если не проведена адекватная выработка требований, вы можете упустить важные детали проблемы. Изменения требований после конструирования обходятся в 20–100 раз дороже, чем на предыдущих этапах, поэтому перед началом программирования обязательно убедитесь в правильности требований.

- Если не проведено адекватное проектирование архитектуры, во время конструирования вы можете решать верную проблему неверным способом. По мере написания кода для неверной архитектуры цена изменений архитектуры возрастает, так что перед началом программирования вы должны проверить и правильность архитектуры.
- Выбор подхода к конструированию должен определяться принятым подходом к предварительным условиям конструирования.

ГЛАВА 4

# Основные решения, которые приходится принимать при конструировании

http://cc2e.com/0489

**Содержание**



**Связанные темы**

- Предварительные условия: глава 3
- Определение типа ПО, над которым вы работаете: раздел 3.2
- Влияние размера программы на ее конструирование: глава 27
- Управление конструированием: глава 28
- Проектирование ПО: главы 5–9

Как только вы убедились, что адекватный фундамент для конструирования программы создан, фокусом подготовки становятся решения, более специфичные для конструирования. В главе 3 мы обсудили программные эквиваленты чертежей и разрешений на конструирование. Как правило, у программистов нет особого контроля над этими подготовительными действиями, поэтому главной темой главы 3 была оценка того, с чем приходится работать в начале конструирования. Эта глава посвящена тем аспектам подготовки, за которые прямо или косвенно отвечают отдельные программисты и технические руководители проекта. Мы рассмотрим выбор специфических инструментов и непосредственную подготовку к работе над приложением.

Если вы думаете, что уже достаточно знаете о подготовке к конструированию, можете сразу перейти к главе 5.

# 4.1. Выбор языка программирования

*Избавляя разум от всей ненужной работы, хорошая нотация позволяет сосредоточиться на более сложных проблемах и в конечном счете повышает интеллект человечества. До появления арабской нотации умножение было весьма сложным, а деление даже целых чисел требовало усилий ведущих математиков. Возможно, ничто в современном мире не смогло бы удивить греческого математика сильнее, чем то, что большинство европейцев умеют делить крупные числа. Это показалось бы ему абсолютно невозможным... Легкость выполнения операций над десятичными дробями — почти сверхъестественный результат постепенного обнаружения отличной нотации.*

*Альфред Норт Уайтхед (Alfred North Whitehead)*

Язык программирования, на котором будет реализована система, заслуживает большого внимания, так как вы будете погружены в него с начала конструирования программы до самого конца.

Исследования показали, что выбор языка программирования несколькими способами влияет на производительность труда программистов и качество создаваемого ими кода.

Если язык хорошо знаком программистам, они работают более производительно. Данные, полученные при помощи модели оценки Cocomo II, показывают, что программисты, использующие язык, с которым они работали три года или более, примерно на 30% более продуктивны, чем программисты, обладающие аналогичным опытом, но для которых язык является новым (Boehm et al., 2000). В более раннем исследовании, проведенном в IBM, было обнаружено, что программисты, обладающие богатым опытом использования языка программирования, были более чем втрое производительнее программистов, имеющих минимальный опыт (Walston and Felix, 1977). (Различия результатов двух исследований объясняются тем, что в модели Cocomo II более тщательно изолируется влияние отдельных факторов.)

Программмисты, использующие языки высокого уровня, достигают более высокой производительности и создают более качественный код, чем программисты, работающие с языками низкого уровня. Утверждается, что при работе с такими языками, как C++, Java, Smalltalk и Visual Basic, производительность труда программистов, а также надежность, простота и понятность программ в 5–15 раз выше, чем при использовании низкоуровневых языков, таких как ассемблер и C (Brooks, 1987; Jones, 1998; Boehm, 2000). Избавившись от необходимости проводить праздничную церемонию каждый раз, когда оператор языка C делает то, что было задумано, вы сэкономите время. Более того, высокоуровневые языки выразительнее низкоуровневых. Каждая строка кода выполняет больший объем работы. В табл. 4-1 указано типичное отношение функциональности команд некоторых языков к функциональности операторов языка C. Показатель, превышающий 1, означает, что строка кода на указанном языке выполняет больше работы, чем строка кода на C.

**Табл. 4-1. Сравнение функциональности операторов высокоуровневых языков с функциональностью операторов C**

| Язык | Функциональность операторов в сравнении с языком C |
|---|---|
| C | 1 |
| C++ | 2,5 |
| Fortran 95 | 2 |
| Java | 2,5 |
| Perl | 6 |
| Python | 6 |
| Smalltalk | 6 |
| Microsoft Visual Basic | 4,5 |

**Источники:** «Estimating Software Costs» (Jones, 1998), «Software Cost Estimation with Cocomo II» (Boehm, 2000) и «An Empirical Comparison of Seven Programming Languages» (Prechelt, 2000).

Некоторые языки лучше выражают концепции программирования, чем другие. Здесь уместно провести параллель между естественными языками — скажем, английским — и языками программирования, такими как Java и C++. Изучая естественные языки, лингвисты Сапир и Уорф (Sapir and Whorf) высказали предположение, что способность к размышлению над определенными идеями связана с выразительной силой языка. Согласно гипотезе Сапира-Уорфа способность человека к обдумыванию определенных мыслей зависит от знания слов, при помощи которых можно выразить эту мысль. Если вы не знаете слов, то не сможете выразить мысль и, возможно, даже сформулировать ее (Whorf, 1956).

Программисты испытывают аналогичное влияние языков программирования. «Слова», которые язык предоставляет программисту для выражения мыслей, несомненно, влияют на способ их выражения, а возможно, даже определяют, какие мысли можно выразить на данном языке.

За доказательствами влияния, оказываемого языками программирования на мышление программистов, далеко ходить не надо. Типичная история такова: «Мы писали новую систему на C++, но большинство наших программистов не имели особого опыта работы на C++. Раньше они использовали Fortran. Они писали код, который компилировался на C++, но на самом деле это был замаскированный Fortran. В итоге они заставили C++ эмулировать недостатки языка Fortran (такие как операторы *goto* и глобальные данные) и проигнорировали богатый набор объектно-ориентированных возможностей C++». Данный феномен наблюдается в отрасли уже много лет (Hanson, 1984; Yourdon, 1986a).

## Описания языков

История разработки некоторых языков и их общие возможности довольно интересны. Ниже приведены описания языков, наиболее популярных в настоящее время.

### Ada

Высокоуровневый язык общего назначения, основанный на языке Pascal. Разработанный под патронажем Минобороны США, он особенно хорошо подходит для

создания встроенных систем и систем, работающих в реальном времени. В языке Ada особое внимание уделяется абстракции данных и сокрытию информации, а также проводится различие между открытыми и закрытыми частями каждого класса и пакета. Название «Ada» было присвоено языку в честь Ады Лавлейс (Ada Lovelace) — женщины-математика, которую считают первым программистом в мире. Сегодня язык Ada используется преимущественно для разработки военных, космических и авиационных систем.

### Ассемблер

Низкоуровневый язык, каждая команда которого соответствует одной команде компьютера. Вследствие этого ассемблер специфичен для отдельных процессоров — например, для конкретных процессоров Intel или Motorola. Ассемблер считается языком второго поколения. Большинство программистов избегают его и используют, только если к быстродействию или компактности кода программы предъявляются повышенные требования.

### C

Среднеуровневый язык общего назначения, первоначально тесно связанный с ОС UNIX. Некоторые свойства (структурированные данные, структурированная управляющая логика, машинная независимость и богатый набор операторов) делают его похожим на высокоуровневый язык. Язык C также называют «портируемым языком ассемблера», поскольку он не строго типизирован, поощряет применение указателей и адресов и поддерживает некоторые низкоуровневые возможности, такие как побитовые операции.

Язык C, разработанный в 1970-х компанией Bell Labs, предназначался для систем DEC PDP-11. На C были написаны ОС, компилятор C и приложения UNIX для систем DEC PDP-11. В 1988 г. для систематизации C был издан стандарт ANSI, который в 1999 г. был пересмотрен. В 1980-х и 1990-х гг. язык C был стандартом «де-факто» в области разработки программ для микрокомпьютеров и рабочих станций.

### C++

Этот объектно-ориентированный язык был разработан на базе C в компании Bell Labs в 1980-х. Совместимый с языком C, он поддерживает классы, полиморфизм, обработку исключений, шаблоны и обеспечивает более надежную проверку типов, чем C. Кроме того, он предоставляет разработчикам богатую и эффективную стандартную библиотеку.

### C#

Эта комбинация объектно-ориентированного языка общего назначения и среды программирования разработана в Microsoft. C# имеет синтаксис, похожий на синтаксис C, C++ и Java, и включает богатый инструментарий, помогающий разрабатывать приложения на платформах Microsoft.

### Cobol

Напоминает английский язык и был разработан в 1959–1961 гг. для нужд Минобороны США. Cobol служит преимущественно для разработки бизнес-приложений и до сих пор является одним из самых популярных языков, уступая лишь Visual Basic (Feiman and Driver, 2002). По мере развития языка в нем была реализована поддержка дополнительных математических функций и ряда объектно-ориентированных возможностей. Аббревиатура «Cobol» расшифровывается как «COmmon Business-Oriented Language» (универсальный язык, ориентированный на коммерческие задачи).

### Fortran

В этом первом высокоуровневом языке программирования были представлены концепции переменных и высокоуровневых циклов. Название расшифровывается как «FORmula TRANslation» (транслятор формул). Разработанный в 1950-х, Fortran претерпел несколько значительных ревизий: так, в 1977 г. была разработана версия Fortran 77, в которой была реализована поддержка блочных операторов if-then-else и манипуляций над символьными строками. В Fortran 90 были включены средства работы с пользовательскими типами данных, указателями, классами, а также богатый набор функций для работы с массивами. Fortran применяется преимущественно для разработки научных и инженерных приложений.

### Java

Синтаксис этого объектно-ориентированного языка, разработанного Sun Microsystems, Inc., напоминает C и C++. Java — платформенно-независимый язык: исходный код Java сначала преобразуется в байт-код, который может выполняться на любой платформе в среде, известной как «виртуальная машина». Java широко используется для создания Web-приложений.

### JavaScript

Этот интерпретируемый язык сценариев мало чем связан с Java. Чаще всего его используют для создания кода, выполняющегося на клиентской стороне, например, для разработки несложных функций и интерактивных приложений для Web-страниц.

### Perl

Этот язык обработки строк основан на C и нескольких утилитах ОС UNIX. Perl часто используется для решения задач системного администрирования, таких как создание сценариев сборки программ, а также для генерации и обработки отчетов. Кроме того, на нем создают Web-приложения, такие как Slashdot. Аббревиатура «Perl» расшифровывается как «Practical Extraction and Report Language» (практический язык извлечений и отчетов).

### PHP

Этот язык с открытым исходным кодом предназначен для разработки сценариев и имеет простой синтаксис, похожий на синтаксис языков Perl, JavaScript, C и оболочки Bourne Shell. PHP поддерживается всеми основными ОС и служит для

создания интерактивных функций, выполняющихся на стороне сервера. PHP-код может быть встроен в Web-страницы для получения доступа к БД и отображения содержащейся в ней информации. Аббревиатура «PHP» первоначально расшифровывалась как «Personal Home Page», но теперь означает «PHP: Hypertext Processor».

### Python

Этот интерпретируемый интерактивный объектно-ориентированный язык поддерживает множество сред. Чаще всего его используют для написания сценариев и небольших Web-приложений, однако он поддерживает и некоторые средства, помогающие создавать более крупные программы.

### SQL

SQL (Structured Query Language, язык структурированных запросов) «де-факто» является стандартным языком выполнения запросов, обновлений реляционнных БД и управления ими. В отличие от других языков, описанных в этом разделе, SQL является «декларативным языком», т. е. определяет не последовательность, а результат выполнения некоторых операций.

### Visual Basic

Basic (Beginner's All-purpose Symbolic Instruction Code, универсальная система символического кодирования для начинающих) — это высокоуровневый язык, первая версия которого была разработана в Дартмутском колледже в 1960-х. Visual Basic — это высокоуровневая объектно-ориентированная версия Basic, предназначенная для визуального программирования. Изначально Visual Basic был разработан в Microsoft для создания приложений Microsoft Windows. Позднее в нем была реализована поддержка настройки Microsoft Office и других приложений для настольных ПК, создания Web-приложений и других программ. По оценкам экспертов в самом начале первого десятилетия XXI века Visual Basic являлся самым популярным языком среди профессиональных разработчиков (Feiman and Driver, 2002).

## 4.2. Конвенции программирования

**Перекрестная ссылка** Подробнее о силе конвенций см. разделы 11.3–11.5.

В высококачественном приложении должна быть очевидна связь между концептуальной целостностью архитектуры и ее низкоуровневой реализацией. Реализация должна соответствовать высокоуровневой архитектуре и обладать внутренней согласованностью. В этом и заключается смысл принципов конструирования, определяющих конвенции именования переменных, классов, методов, а также форматирования кода и оформления комментариев.

При разработке сложной программы архитектурные принципы вносят в программу структурный баланс, а принципы конструирования — низкоуровневую гармонию, при наличии которой каждый класс воспринимается как неотъемлемая часть общего плана. Любая крупная программа требует применения контролирующей структуры, унифицирующей аспекты языка программирования. Красота крупной структуры частично заключается в том, как в ее отдельных компонентах выражены особенности архитектуры. Без унификации ваша программа будет смесью

небрежных вариаций стиля, заставляющих прилагать дополнительные усилия только для того, чтобы понять различия в стиле кодирования, которых вполне можно было избежать. Одно из условий успешного программирования — устранение ненужных вариаций, позволяющее сосредоточиться на действительно необходимых вариациях. См. об этом подраздел «Главный технический императив разработки ПО: управление сложностью» раздела 5.2.

Что, если у вас есть отличный план создания картины, но одну ее часть вы решите писать в классическом стиле, другую в импрессионистском, а третью в кубистском? Как бы упорно вы ни следовали своему грандиозному плану, картина не будет концептуально целостной. Она будет похожа на коллаж. Программа тоже должна обладать низкоуровневой целостностью.

Перед началом конструирования сформулируйте конвенции программирования. Детали конвенций кодирования относятся к такому низкому уровню, что после написания программы их почти невозможно изменить. В оставшейся части книги я еще не раз затрону конвенции кодирования.

## 4.3. Волны развития технологий

Я видел, как взошла звезда ПК, в то время как звезда мэйнфреймов опустилась за горизонт. Я видел, как консольные программы были вытеснены программами с GUI. Я также видел, как традиционные программы уступили главную роль Web-приложениям. Могу предположить, что, когда вы будете читать эту книгу, будут бурно развиваться некоторые новые технологии, а Web-программирование в его современном (2004) виде начнет отходить на второй план. В соответствии с этими технологическими циклами, или волнами, изменяются и методики программирования.

В зрелых технологических средах — таких как среда Web-программирования в середине 2000-х — нам доступны все достоинства богатой инфраструктуры разработки ПО. Такие среды предоставляют широкий выбор языков программирования, мощные средства поиска ошибок, эффективные инструменты отладки и надежные автоматизированные средства оптимизации производительности приложений. Компиляторы почти не содержат ошибок. Инструменты хорошо описаны в документации производителей, в книгах и статьях сторонних фирм и на многочисленных Web-сайтах. Инструменты интегрированы, благодаря чему вы можете разрабатывать UI, модули работы с БД, составления отчетов и бизнес-логики в одной среде. Решения проблем можно легко найти в ответах на «часто задаваемые вопросы». Кроме того, доступны разнообразные услуги консультантов и программы тренинга.

В ранних средах — таких как Web-программирование в середине 1990-х — ситуация противоположная. Языков программирования мало, при этом они часто полны ошибок и плохо документированы. Вместо написания нового кода программисты тратят массу времени только на то, чтобы разобраться в особенностях языка. Бесчисленные часы уходят на борьбу с ошибками в языках, ОС и других инструментах. Инструменты программирования часто примитивны. Отладчиков может не быть вообще, а об оптимизаторах компиляторов программистам приходится

лишь мечтать. Производители часто выпускают новые версии компиляторов, при этом каждая новая версия отказывается поддерживать значительные части вашего кода. Инструменты не интегрированы, из-за чего UI, модули работы с БД, составления отчетов и бизнес-логики приходится разрабатывать при помощи разных средств. Из-за плохой совместимости инструментов и частого появления новых компиляторов и библиотек программисты тратят много усилий только на поддержание работоспособности имеющейся инфраструктуры. При возникновении проблем в Интернете можно найти кое-какую документацию, но она не отличается достоверностью и полнотой.

Вам может показаться, что я рекомендую избегать программирования в ранних средах, но это не так. В ранних средах были разработаны программы, давшие начало некоторым из самых инновационных приложений, такие как Turbo Pascal, Lotus 123, Microsoft Word и браузер Mosaic. Я просто хочу сказать, что от стадии развития технологии зависит то, как будет протекать ваша работа. В зрелой среде вы можете посвящать большую часть дня постепенной реализации новой функциональности. Работая в ранней среде, исходите из того, что вам придется тратить много времени на выяснение недокументированных возможностей выбранного языка программирования, отладку ошибок, которые в итоге окажутся дефектами библиотек, проверку того, что написанный код будет работать с новой версией библиотеки какого-нибудь производителя и т. д.

При работе в примитивной среде методики программирования, описанные в этой книге, могут оказаться еще более полезными, чем в зрелых средах. Как сказал Дэвид Грайс (Gries, 1981), подход к программированию не должен определяться используемыми инструментами. В связи с этим он проводит различие между программированием *на* языке (programming *in* language) и программированием *с использованием* языка (programming *into* language). Разработчики, программирующие «на» языке, ограничивают свое мышление конструкциями, непосредственно поддерживаемых языком. Если предоставляемые языком средства примитивны, мысли программистов будут столь же примитивными.

Разработчики, программирующие «с использованием» языка, сначала решают, какие мысли они хотят выразить, после чего определяют, как выразить их при помощи конкретного языка.

## Пример программирования *с использованием* языка

Разрабатывая программу на Visual Basic, который тогда находился на раннем этапе развития, я с огорчением обнаружил, что язык не поддерживает встроенных способов разделения бизнес-логики, кода GUI и кода работы с БД. Я знал, что, если буду невнимателен, со временем некоторые из моих «форм» Visual Basic включат в себя код бизнес-логики, другие — код доступа к БД, а остальные не будут содержать ни того, ни другого — в итоге я не смогу вспомнить, какая форма за что отвечает. Я только что завершил работу над проектом C++, в котором разделение кода было выполнено плохо, и не хотел еще раз наступать на те же грабли.

Поэтому я принял конвенцию, в соответствии с которой файлам .frm (файлам формы) дозволялось только извлекать данные из БД и сохранять их обратно, но не передавать эти данные другим частям программы. Все формы поддерживали

метод IsFormCompleted(), который сообщал вызвавшему его методу, сохранила ли активная форма свои данные. IsFormCompleted() был единственным открытым методом, который могли иметь формы. Код форм также не мог включать никакой бизнес-логики. Весь остальной код, в том числе проверяющий корректность вводимых в форму данных, должен был содержаться в ассоциированном файле .bas.

Visual Basic не поощрял такого подхода. Он поощрял программистов включать в файл .frm максимальный объем кода, и это отнюдь не облегчало реализацию взаимодействия межу файлами .frm и .bas.

Принятая мной конвенция была очень проста, но по мере развития проекта я обнаружил, что она помогла мне избежать многих случаев, в которых мне пришлось бы писать неестественный код. Так, мне пришлось бы загружать формы, но держать их скрытыми, чтобы можно было вызвать реализованные в них методы проверки корректности данных, или мне пришлось бы копировать код форм в другие места программы и сопровождать этот параллельный код. Кроме того, конвенция IsFormCompleted() позволила все упростить. Все формы работали одинаково, поэтому я мог не предполагать семантику IsFormCompleted() — вызовы этого метода всегда имели одинаковый смысл.

Visual Basic не поддерживал такой подход непосредственно, но простая конвенция программирования — программирование *с использованием* языка — позволила мне реализовать отсутствующую в то время структуру языка и помогла упростить проект до приемлемого уровня.

Понимание различия между программированием на языке и программированием с использованием языка — важнейшее условие понимания этой книги. Большинство важных принципов программирования зависит не от конкретных языков, а от способа их использования. Если язык не поддерживает нужные конструкции или имеет другие недостатки, попробуйте их компенсировать. Создайте свои конвенции кодирования, стандарты, библиотеки классов и другие средства.

## 4.4. Выбор основных методик конструирования

При подготовке к конструированию следует решить, какие из доступных эффективных методик вы будете использовать. Некоторые проекты предусматривают парное программирование и предварительное создание тестов, тогда как другие — индивидуальное программирование и проведение формальных инспекций. Обе комбинации методик могут быть удачными, но при их выборе следует учитывать специфические особенности проекта.

Специфические методики конструирования, которые должны быть осознанно приняты или отвергнуты, указаны ниже. В оставшейся части книги эти методики будут описаны подробнее.

### Контрольный список: основные методики конструирования

http://cc2e.com/0496

**Кодирование**

- ❑ Решили ли вы, какая часть проекта приложения будет разработана предварительно, а какая во время написания кода?
- ❑ Выбрали ли вы конвенции именования программных элементов, оформления комментариев и форматирования кода?
- ❑ Выбрали ли вы специфические методики кодирования, определяемые архитектурой приложения? Определили ли вы, как будут обрабатываться ошибки, как будут решаться проблемы, связанные с безопасностью, какие конвенции будут использоваться при разработке интерфейсов классов, каким стандартам должен будет отвечать повторно используемый код, сколько внимания нужно будет уделять быстродействию приложения при кодировании и т. д.?
- ❑ Определили ли вы стадию развития используемой технологии и адаптировали ли к ней свой подход? Если это необходимо, определились ли вы с тем, как будете программировать с использованием языка, вместо того чтобы ограничиваться программированием на нем?

**Работа в группе**

- ❑ Определили ли вы процедуру интеграции? Иначе говоря, какие специфические действия программист должен будет выполнить перед включением своего кода в исходный код всего проекта?
- ❑ Будут ли программисты программировать парами, индивидуально или эти подходы будут скомбинированы?

**Гарантия качества**

- ❑ Должны ли будут программисты разработать тесты для своего кода до написания самого кода?
- ❑ Должны ли будут программисты разработать блочные тесты для своего кода?
- ❑ Должны ли будут программисты перед включением своего кода в исходный код всего проекта проанализировать его в отладчике?
- ❑ Должны ли будут программисты выполнить интеграционное тестирование своего кода до его включения в исходный код проекта?
- ❑ Будут ли программисты выполнять взаимные обзоры или инспекцию кода?

**Перекрестная ссылка** Гарантия качества рассматривается в главе 20.

**Инструменты**

- ❑ Выбрали ли вы инструмент управления версиями?
- ❑ Выбрали ли вы язык, версию языка и версию компилятора?
- ❑ Выбрали ли вы платформу программирования (такую как J2EE или Microsoft .NET) или явно решили не использовать ее?
- ❑ Приняли ли вы решение о том, можно ли будет использовать нестандартные возможности языка?
- ❑ Определили ли вы другие средства, которые будете применять: редактор, инструмент рефакторинга, платформу для тестирования, модуль проверки синтаксиса и т. д.? Приобрели ли вы их?

**Перекрестная ссылка** Об инструментах программирования см. главу 30.

## Ключевые моменты

- Каждый язык программирования имеет достоинства и недостатки. Вы должны знать отдельные достоинства и недостатки используемого языка.
- Определите конвенции программирования до начала программирования. Позднее адаптировать к ним код станет почти невозможно.
- Методик конструирования слишком много, чтобы использовать все в одном проекте. Тщательно выбирайте методики, наиболее подходящие для вашего проекта.
- Спросите себя, являются ли используемые вами методики программирования ответом на выбранный язык программирования или их выбор был определен языком. Помните, что программировать следует с использованием языка, а не на языке.
- Эффективность конкретных подходов и даже возможность их применения зависит от стадии развития соответствующей технологии. Определите стадию развития используемой технологии и адаптируйте к ней свои планы и ожидания.

# Часть II

# ВЫСОКОКАЧЕСТВЕННЫЙ КОД

- **Глава 5.** Проектирование при конструировании
- **Глава 6.** Классы
- **Глава 7.** Высококачественные методы
- **Глава 8.** Защитное программирование
- **Глава 9.** Процесс программирования с псевдокодом

ГЛАВА 5

# Проектирование при конструировании

http://cc2e.com/0578

**Содержание**



**Связанные темы**

- Разработка архитектуры ПО: раздел 3.5
- Классы: глава 6
- Характеристики высококачественных методов: глава 7
- Защитное программирование: глава 8
- Рефакторинг: глава 24
- Зависимость конструирования от объема программы: глава 27

Некоторые программисты могут заявить, что проектирование не связано с конструированием, но при работе над небольшими проектами конструирование часто включает другие процессы, в том числе проектирование. В некоторых более крупных проектах формальная архитектура может давать ответы только на во-просы системного уровня, при этом значительная часть проектирования может быть намеренно оставлена на этап конструирования. В других крупных проектах проектирование может быть проведено в таком объеме, что кодирование становится почти механическим, однако это случается редко — официально или нет, программисты обычно сами проектируют некоторые фрагменты программы.

**Перекрестная ссылка** Об уровнях формальности, требуемой при работе над крупными и небольшими проектами, см. главу 27.

В случае небольших неформальных проектов значительная часть проектирования выполняется за клавиатурой. «Проектирование» может выражаться в простом написании интерфейса класса на псевдокоде до разработки его деталей. Оно может выражаться в рисовании диаграмм отношений

нескольких классов перед написанием их кода. Оно может выражаться в обсуждении оптимального шаблона проектирования вместе с коллегой. Какую бы форму проектирование ни принимало, от тщательного его выполнения выигрывают проекты любого масштаба, и, рассматривая проектирование как явный процесс, вы извлечете из него максимальную выгоду.

Проектирование — очень обширная тема, поэтому в данной главе мы рассмотрим только несколько ее аспектов. Эффективность проектирования классов или методов во многом определяется архитектурой системы, поэтому убедитесь, что вы выполнили предварительные условия, связанные с разработкой архитектуры (см. раздел 3.5). Еще больший объем проектирования выполняется на уровне отдельных классов и методов, что мы обсудим в главах 6 и 7.

Если вы уже хорошо знакомы с проектированием ПО, можете только бегло просмотреть основные моменты раздела 5.1, посвященного проблемам проектирования, и раздела 5.3, в котором обсуждаются основные эвристические принципы проектирования.

## 5.1. Проблемы, связанные с проектированием ПО

Под «проектированием ПО» понимают разработку или изобретение схемы преобразования спецификации приложения в готовое приложение. Проектирование — это тот процесс, который связывает выработку требований с кодированием и отладкой. Структура удачного высокоуровневого проекта приложения может успешно охватывать целый ряд более низкоуровневых проектов. Хорошее проектирование полезно при работе над небольшими приложениями и просто необходимо при работе над крупными.

**Перекрестная ссылка** О различии между эвристическим и детерминированным процессами см. главу 2.

Однако с проектированием связано множество проблем — их-то мы и обсудим.

### Проектирование — «грязная» проблема

Образ разработчика, проектирующего программу рациональным безошибочным способом на основе ясно сформулированных требований, совершенно нереалистичен. Никакая система так никогда не разрабатывалась и, наверное, не будет разрабатываться. Даже примеры разработки небольших программ, встречающиеся в учебниках, нереалистичны. Авторы перепроверяют и улучшают их до тех пор, пока не продемонстрируют нам то, что они хотели бы получить, а не то, что получается на самом деле.

*Дэвид Парнас и Пол Клементс (David Parnas and Paul Clements)*

Хорст Риттел и Мелвин Веббер определили «грязную» проблему как проблему, которую можно ясно определить только путем полного или частичного решения (Rittel and Webber, 1973). По сути данный парадокс подразумевает, что проблему нужно «решить» один раз, чтобы получить ее ясное определение, а затем еще раз для создания работоспособного решения. Этот процесс уже несколько десятилетий неразрывно связан с разработкой ПО (Peters and Tripp, 1976).

Одним драматическим примером подобной грязной проблемы является проектирование первого варианта моста Tacoma Narrows. В то время главным соображением при проектировании мостов было обеспечение прочности, адекватной планируемой нагрузке. В случае моста Tacoma Narrows оказалось, что ветер вызывает непредвиденные волнообразные гармонические колебания моста из стороны в

сторону. В один ветреный день 1940 г. колебания неконтролируемо усилились, и часть моста обрушилась (рис. 5-1).

Это наглядный пример грязной проблемы: до разрушения моста инженеры не знали, что аэродинамика играет такую большую роль. Только построив мост (решив проблему), они смогли обнаружить дополнительный аспект проблемы, что позволило им возвести новый мост, действующий и поныне.

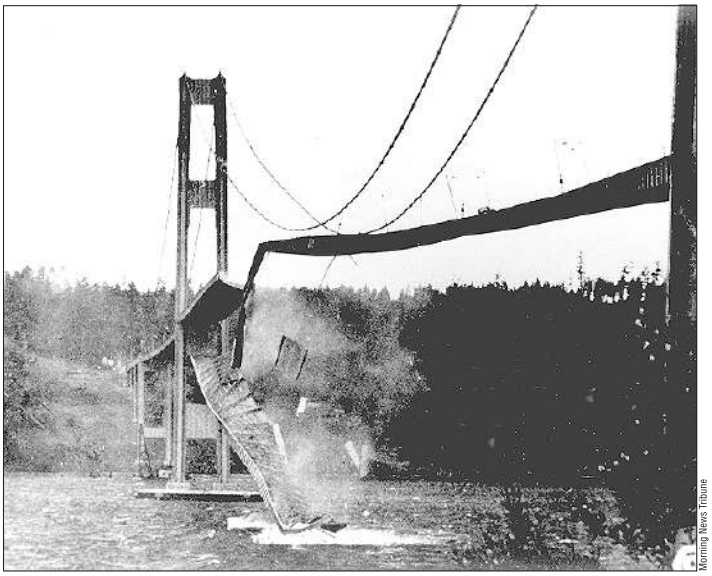



***Рис. 5-1.*** *Мост Tacoma Narrows — пример грязной проблемы*

Одно из главных отличий программ, которые вы разрабатывали в институте, от программ, которые разрабатываете теперь, став профессиональным программистом, в том, что проблемы проектирования, решаемые институтскими программами, редко бывают грязными, если вообще бывают таковыми. В институте задания по программированию составлены так, чтобы вы по кратчайшему пути двигались от начала решения к его результату. Преподавателя, который дает студентам задания и свободно изменяет их по завершении проектирования и даже перед сдачей готовых программ, вероятно, облили бы дегтем и вываляли в перьях. Однако в мире профессионального программирования такие изменения происходят ежедневно.

## Проектирование — неряшливый процесс (даже если оно приводит к аккуратному результату)

Завершенный проект приложения должен выглядеть хорошо организованным и ясным, но процесс разработки этого проекта далеко не так аккуратен, как конечный результат.

Проектирование неряшливо потому, что вы выполняете много неверных действий и попадаете во множество тупиков, т. е. совершаете массу ошибок. В действительности ошибки являются сутью проектирования: дешевле допустить ошибки и исправить проект программы, чем найти их после кодирования и исправлять готовый код. Проектирование неряшливо потому, что удачное решение часто лишь чуть-чуть отличается от неудачного.

**Дополнительные сведения** См. обсуждение этой точки зрения в статье «A Rational Design Process: How and Why to Fake It» (Parnas and Clements, 1986).

Проектирование неряшливо еще и потому, что трудно узнать, когда проект «достаточно хорош». Какого уровня детализации достаточно? Какую часть проектирования выполнить с использованием формальной нотации, а какую — прямо за клавиатурой? Когда проектирование считать завершенным? Улучшать проект программы можно постоянно, поэтому чаще всего на последний вопрос отвечают: «Когда у вас вышло время».

**Перекрестная ссылка** Лучший ответ на этот вопрос см. в подразделе «Какую степень проектирования можно считать достаточной?» раздела 5.4.

## Проектирование связано с определением компромиссов и приоритетов

В идеальном мире все системы обладали бы бесконечным быстродействием, не предъявляли никаких требований к подсистеме хранения данных, давали нулевую нагрузку на сеть, никогда не содержали никаких ошибок и создавались без всяких затрат. Однако в реальном мире один из важнейших аспектов работы проектировщика — анализ конкурирующих характеристик проекта и достижение баланса между ними. Если быстрота отклика системы важнее, чем минимизация времени разработки, проектировщик выберет один вариант. Если во главе угла быстрота разработки, оптимальным может оказаться другой вариант проекта.

## Проектирование подразумевает ограничение возможностей

Проектирование предполагает не только обеспечение возможностей, но и их *ограничение*. Если бы люди, создавая физические структуры, обладали бесконечным объемом времени и ресурсов, мы увидели бы на улицах невероятно странные здания с диковинными башенками и сотнями комнат на отдельных этажах. При отсутствии целенаправленно заданных ограничений ПО может оказаться именно таким. Ограниченные объемы ресурсов при конструировании зданий требуют упрощения решения, что в итоге приводит к его улучшению. Проектирование ПО в этом смысле ничем не отличается.

## Проектирование — недетерминированный процесс

Если вы попросите трех человек спроектировать одну и ту же программу, они вполне могут разработать три совершенно разных, но вполне приемлемых проекта. Как правило, спроектировать компьютерную программу можно десятками разных способов.

## Проектирование — эвристический процесс

Так как проектирование не детерминировано, методы проектирования чаще всего являются эвристическими методами, т. е. «практическими правилами» или «способами, которые могут сработать», а не воспроизводимыми процессами, которые всегда приводят к предсказуемым результатам. Проектирование — метод проб и ошибок. Инструменты или методы проектирования, оказавшиеся эффективными в одном случае, в другой ситуации могут оказаться куда менее эффективными. Универсальных методик проектирования не существует.

## Проектирование — постепенный процесс

http://cc2e.com/0539

Можно довольно удачно обобщить названные аспекты проектирования, сказав, что проектирование — «постепенный» процесс. Проекты приложений не возникают в умах разработчиков сразу в готовом виде. Они развиваются и улучшаются в ходе обзоров, неформальных обсуждений, написания кода и выполнения его ревизий.

**Дополнительные сведения** ПО — не единственный тип структур, изменяющихся с течением времени. Физические структуры также развиваются; см. об этом книгу «How Buildings Learn» (Brand, 1995).

Практически во всех случаях проект несколько меняется во время первоначальной разработки системы и еще больше — при ее модернизации. Степень, в которой изменение выгодно или приемлемо, зависит от особенностей создаваемого ПО.

# 5.2. Основные концепции проектирования

Успешное проектирование ПО требует понимания нескольких важных концепций. Здесь мы обсудим роль сложности при проектировании, желательные характеристики проектов и уровни проектирования.

## Главный Технический Императив Разработки ПО: управление сложностью

**Перекрестная ссылка** О влиянии сложности на другие аспекты программирования см. раздел 34.1.

Чтобы лучше понять важность управления сложностью, обратимся к известной работе Фреда Брукса «No Silver Bullets: Essence and Accidents of Software Engineering» (Brooks, 1987).

### Существенные и несущественные проблемы

Брукс утверждает, что сложность разработки ПО объясняется *существенными* и *несущественными* проблемами. Используя два этих термина, Брукс опирается на философскую традицию, уходящую корнями к Аристотелю. В философии существенными называют свойства, которыми объект должен обладать, чтобы быть именно этим объектом. Автомобиль должен иметь двигатель, колеса и двери — если объект не обладает каким-нибудь из этих существенных свойств, это не автомобиль.

Несущественными (акцидентными) свойствами называют свойства, которыми объект обладает в силу случайности, — свойства, не влияющие на его суть. Так, автомобиль может иметь четырехцилиндровый двигатель с турбонаддувом, восьмицилиндровый или любой другой и все же являться автомобилем. Тип двигателя

и колес, число дверей — все это несущественные свойства. Можете также думать о них как о *второстепенных*, *произвольных*, *необязательных* и *случайных*.

Брукс замечает, что главные несущественные проблемы разработки ПО уже давно решены. Например, несущественные проблемы, связанные с неудобным синтаксисом языков программирования, постепенно утратили свою значимость по мере эволюции языков. Несущественные проблемы, связанные с неинтерактивностью компьютеров, исчезли, когда на смену ОС, работающим в пакетном режиме, пришли системы с разделением времени. Среды интегрированной разработки избавили программистов от проблем, обусловленных плохим взаимодействием инструментов.

**Перекрестная ссылка** В ранних средах несущественные проблемы проявляются сильнее, чем в зрелых (см. раздел 4.3).

В то же время Брукс утверждает, что решение оставшихся *существенных* проблем разработки ПО будет более медленным. Это объясняется тем, что разработка программ по своей сути требует анализа всех деталей крайне сложного набора взаимосвязанных концепций. Причиной существенных проблем является необходимость анализа сложного неорганизованного реального мира, точного и полного определения зависимостей и исключений, проектирования абсолютно, но никак не приблизительно верных решений и т. д. Даже если б мы смогли придумать язык программирования, основанный на той же терминологии, что и требующая решения проблема реального мира, программирование все равно осталось бы сложным из-за необходимости точного определения принципов функционирования мира. По мере того как разработчики ПО берутся за решение все более серьезных проблем реального мира, им приходится анализировать все более сложные взаимодействия между сущностями, что в свою очередь приводит к повышению существенной сложности программных решений.

Источник всех этих существенных проблем — сложность как несущественная, так и существенная.

### Важность управления сложностью

Программные проекты редко терпят крах по техническим причинам. Чаще всего провал объясняется неадекватной выработкой требований, неудачным планированием или неэффективным управлением. Если же провал обусловлен все-таки преимущественно технической причиной, очень часто ею оказывается неконтролируемая сложность. Иначе говоря, приложение стало таким сложным, что разработчики перестали по-настоящему понимать, что же оно делает. Если работа над проектом достигает момента, после которого уже никто не может полностью понять, как изменение одного фрагмента программы повлияет на другие фрагменты, прогресс прекращается.

> Есть два способа разработки проекта приложения: сделать его настолько простым, чтобы было *очевидно*, что в нем нет недостатков, или сделать его таким сложным, чтобы в нем не было *очевидных* недостатков.
>
> *Ч. Э. Р. Хоар (C. A. R. Hoare)*

Управление сложностью — самый важный технический аспект разработки ПО. По-моему, *управление сложностью* настолько важно, что оно должно быть Главным Техническим Императивом Разработки ПО.

Сложность — не новинка в мире разработки ПО. Один из пионеров информатики Эдсгер Дейкстра обращал внимание на то, что компьютерные технологии —

единственная отрасль, заставляющая человеческий разум охватывать диапазон, простирающийся от отдельных битов до нескольких сотен мегабайт информации, что соответствует отношению 1 к $10^9$, или разнице в девять порядков (Dijkstra, 1989). Такое гигантское отношение просто ошеломляет. Дейкстра выразил это так: «По сравнению с числом семантических уровней средняя математическая теория кажется почти плоской. Создавая потребность в глубоких концептуальных иерархиях, компьютерные технологии бросают нам абсолютно новый интеллектуальный вызов, не имеющий прецедентов в истории». Разумеется, за прошедшее с 1989 г. время сложность ПО только выросла, и сегодня отношение Дейкстры вполне может характеризоваться 15 порядками.

Одним из симптомов того, что вы погрязли в чрезмерной сложности, является упрямое применение метода, нерелевантность которого очевидна по крайней мере любому внешнему наблюдателю. При этом вы уподобляетесь человеку, который при поломке автомобиля в силу своей некомпетентности не находит ничего лучшего, чем заменить воду в радиаторе и выбросить окурки из пепельниц.

*Ф. Дж. Плоджер (P. J. Plauger)*

Дейкстра пишет, что ни один человек не обладает интеллектом, способным вместить все детали современной компьютерной программы (Dijkstra, 1972), поэтому нам — разработчикам ПО — не следует пытаться охватить всю программу сразу. Вместо этого мы должны попытаться организовать программы так, чтобы можно было безопасно работать с их отдельными фрагментами по очереди. Целью этого является минимизация объема программы, о котором нужно думать в конкретный момент времени. Можете считать это своеобразным умственным жонглированием: чем больше умственных шаров программа заставляет поддерживать в воздухе, тем выше вероятность того, что вы уроните один из них и допустите ошибку при проектировании или кодировании.

На уровне архитектуры ПО сложность проблемы можно снизить, разделив систему на подсистемы. Несколько несложных фрагментов информации понять проще, чем один сложный. В разбиении сложной проблемы на простые фрагменты и заключается цель всех методик проектирования ПО. Чем более независимы подсистемы, тем безопаснее сосредоточиться на одном аспекте сложности в конкретный момент времени. Грамотно определенные объекты разделяют аспекты проблемы так, чтобы вы могли решать их по очереди. Пакеты обеспечивают такое же преимущество на более высоком уровне агрегации.

Стремление к краткости методов программы помогает снизить нагрузку на интеллект. Этому же способствует написание программы в терминах проблемной области, а не низкоуровневых деталей реализации, а также работа на самом высоком уровне абстракции.

Суть сказанного в том, что программисты, компенсирующие изначальные ограничения человеческого ума, пишут более понятный и содержащий меньшее число ошибок код.

### Как бороться со сложностью?

Чаще всего причинами неэффективности являются:

- сложное решение простой проблемы;
- простое, но неверное решение сложной проблемы;
- неадекватное сложное решение сложной проблемы.

Как указал Дейкстра, сложность современного ПО обусловлена самой его природой, поэтому, как бы вы ни старались, вы все равно столкнетесь со сложностью, присущей самой проблеме реального мира. Исходя из этого, можно предложить двойственный подход к управлению сложностью:

- старайтесь свести к минимуму объем существенной сложности, с которым придется работать в каждый конкретный момент времени;
- сдерживайте необязательный рост несущественной сложности.

Как только вы поймете, что все остальные технические цели разработки ПО вторичны по отношению к управлению сложностью, многие принципы проектирования окажутся простыми.

## Желательные характеристики проекта

Высококачественные проекты программ имеют несколько общих характеристик. Если вы сумеете достичь всех этих целей, ваш проект на самом деле будет очень хорош. Некоторые цели противоречат другим, но это и есть одна из задач проектирования — объединение конкурирующих целей в удачном наборе компромиссов. Некоторые аспекты качества проекта — надежность, производительность и т. д. — описывают и качество программы, тогда как другие являются внутренними характеристиками проекта.

> Работая над проблемой, я никогда не думаю о красоте. Я думаю только о решении проблемы. Но если полученное решение некрасиво, я знаю, что оно неверно.
>
> *Р. Бакминстер Фуллер (R. Buckminster Fuller)*

Вот список таких внутренних характеристик проекта.

> **Перекрестная ссылка** Эти характеристики связаны с общими атрибутами качества ПО (см. раздел 20.1).

***Минимальная сложность*** В силу только что описанных причин главной целью проектирования должна быть минимизация сложности. Избегайте создания «хитроумных» проектов: как правило, их трудно понять. Вместо этого создавайте «простые» и «понятные» проекты. Если при работе над отдельным фрагментом программы проект не позволяет безопасно игнорировать большинство остальных фрагментов, он неудачен.

***Простота сопровождения*** Проектируя приложение, не забывайте о программистах, которые будут его сопровождать. Постоянно представляйте себе вопросы, которые будут возникать у них при взгляде на создаваемый вами код. Думайте о таких программистах как о своей аудитории и проектируйте систему так, чтобы ее работа была очевидной.

***Слабое сопряжение*** Слабое сопряжение (loose coupling) предполагает сведение к минимуму числа соединений между разными частями программы. Для проектирования классов с минимальным числом взаимосвязей используйте принципы адекватной абстракции интерфейсов, инкапсуляцию и сокрытие информации. Это позволит максимально облегчить интеграцию, тестирование и сопровождение программы.

***Расширяемость*** Расширяемостью системы называют свойство, позволяющее улучшать систему, не нарушая ее основной структуры. Изменение одного фрагмента системы не должно влиять на ее другие фрагменты. Внесение наиболее вероятных изменений должно требовать наименьших усилий.

***Возможность повторного использования*** Проектируйте систему так, чтобы ее фрагменты можно было повторно использовать в других системах.

***Высокий коэффициент объединения по входу*** При высоком коэффициенте объединения по входу (fan-in) к конкретному классу обращается большое число других классов. Это значит, что система предусматривает интенсивное использование вспомогательных низкоуровневых классов.

***Низкий или средний коэффициент разветвления по выходу*** Это означает, что конкретный класс обращается к малому или среднему числу других классов. Высокий коэффициент разветвления по выходу (fan-out) (более семи) говорит о том, что класс использует большое число других классов и, возможно, слишком сложен. Ученые обнаружили, что низкий коэффициент разветвления по выходу выгоден как в случае вызова методов из метода, так и в случае вызова методов из класса (Card and Glass, 1990; Basili, Briand, and Melo, 1996).

***Портируемость*** Проектируйте систему так, чтобы ее можно было легко адаптировать к другой среде.

***Минимальная, но полная функциональность*** Этот аспект подразумевает отсутствие в системе лишних частей (Wirth, 1995; McConnell, 1997). Вольтер говорил, что книга закончена не тогда, когда в нее больше нечего добавить, а когда из нее ничего нельзя выбросить. При разработке ПО это верно вдвойне, потому что дополнительный код необходимо разработать, проанализировать, протестировать, а также пересматривать при изменении других фрагментов программы. Кроме того, в будущих версиях приложения придется поддерживать обратную совместимость с дополнительным кодом. Опасайтесь вопроса: «Эту функцию реализовать легко — почему бы этого не сделать?»

***Стратификация*** Под стратификацией понимают разделение уровней декомпозиции, позволяющее изучить систему на любом отдельном уровне и получить при этом согласованное представление. Проектируйте систему так, чтобы ее можно было изучать на отдельных уровнях, игнорируя другие уровни.

**Перекрестная ссылка** О работе со старыми системами см. раздел 24.5.

Например, если вы создаете современную систему, которая должна использовать большой объем старого, плохо спроектированного кода, напишите уровень, отвечающий за взаимодействие со старым кодом. Спроектируйте этот уровень так, чтобы он скрывал плохое качество старого кода, предоставляя более новым уровням согласованный набор сервисов. Пусть остальные части системы работают с этими классами вместо старого кода. Такой подход сулит два преимущества: 1) он изолирует плохой код и 2) если вы когда-нибудь решите выбросить старый код или выполнить его рефакторинг, вам не придется изменять новый код за исключением промежуточного уровня.

**Перекрестная ссылка** Об особенно полезном типе стратификации — применении шаблонов проектирования — см. подраздел «Старайтесь использовать популярные шаблоны проектирования» раздела 5.3.

***Соответствие стандартным методикам*** Чем экзотичнее система, тем сложнее будет другим программистам понять ее. Попытайтесь придать всей системе привычный для разработчиков облик, применяя стандартные популярные подходы.

## Уровни проектирования

Проектирование программной системы требует нескольких уровней детальности. Некоторые методы проектирования используются на всех уровнях, а другие только на одном-двух (рис. 5-2).

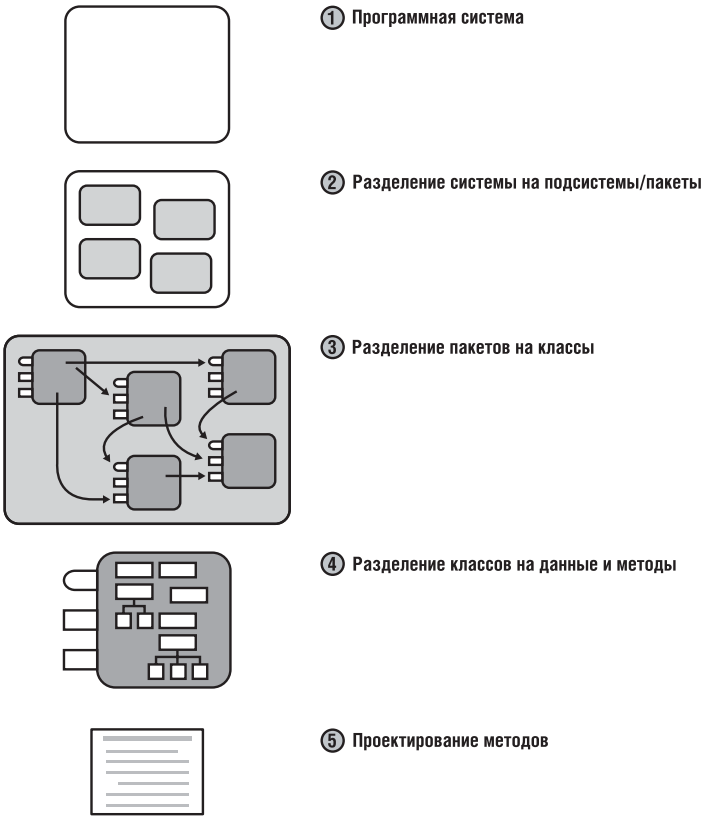



***Рис. 5-2.*** *Уровни проектирования программы. Систему (1) следует разделить на подсистемы (2), подсистемы — на классы (3), а классы — на методы и данные (4); методы также необходимо спроектировать (5)*

### Уровень 1: программная система

Первому уровню проектирования соответствует вся система. Некоторые программисты с системного уровня сразу переходят к проектированию классов, но обычно целесообразно обдумать более высокоуровневые комбинации классов, такие как подсистемы или пакеты.

> Иными словами — и это неизменный принцип, на котором основан всегалактический успех всей корпорации, — фундаментальные изъяны конструкции ее товаров камуфлируются их внешними изъянами.
>
> *Дуглас Адамс*
> (*Douglas Adams*)

### Уровень 2: разделение системы на подсистемы или пакеты

Главный результат проектирования на этом уровне — определение основных подсистем. Подсистемы могут быть

довольно крупными, такими как модуль работы с базами данных, модули GUI, бизнес-правил или создания отчетов, интерпретатор команд и т. д. Суть проектирования на данном уровне заключается в разделении программы на основные подсистемы и определении взаимодействий между подсистемами. Обычно этот уровень нужен при работе над любыми проектами, требующими более нескольких недель. При проектировании отдельных подсистем можно применять разные подходы: выбирайте тот, который кажется вам оптимальным в каждом конкретном случае. На рис. 5-2 данный уровень проектирования обозначен цифрой 2.

Особенно важный аспект этого уровня — определение правил взаимодействия подсистем. Если все подсистемы могут взаимодействовать, выгода их разделения исчезает. Подчеркивайте суть подсистем, ограничивая их взаимодействие между собой.

Допустим, вы определили систему из шести подсистем (рис. 5-3). При отсутствии каких-либо ограничений в силу второго закона термодинамики энтропия системы должна увеличиться. Один из способов увеличения энтропии является абсолютно свободное взаимодействие между подсистемами (рис. 5-4).

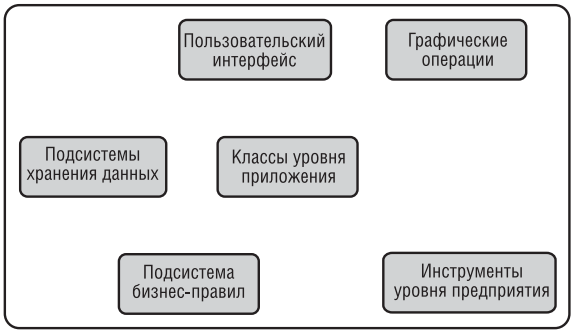



***Рис. 5-3.*** *Пример системы, включающей шесть подсистем*

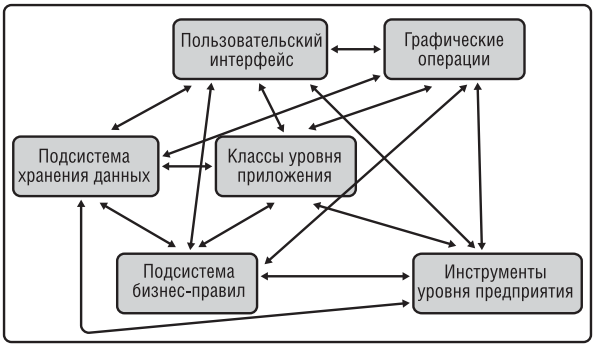



***Рис. 5-4.*** *Возможный результат отсутствия правил, ограничивающих взаимодействие подсистем*

Как видите, в итоге все подсистемы начинают напрямую взаимодействовать, что поднимает несколько важных вопросов:

- в скольких разных частях системы нужно хоть немного разбираться разработчику, желающему изменить какой-то аспект подсистемы графических операций?

- что будет, если вы попытаетесь задействовать данный модуль бизнес-правил в другой системе?
- что будет, если вы захотите включить в систему новый пользовательский интерфейс (например, интерфейс командной строки, удобный для проведения тестирования)?
- что произойдет, если вы захотите перенести модуль хранения данных на удаленный компьютер?

Стрелки между подсистемами можно рассматривать как шланги с водой. Если вам захочется «выдернуть» одну из подсистем, к ней наверняка будут подключены несколько шлангов. Чем больше шлангов вам нужно будет отсоединить и подключить заново, тем сильнее вы промокнете. Архитектура системы должна быть такой, чтобы замена подсистем требовала как можно меньше возни со шлангами.

При должной предусмотрительности все эти вопросы можно решить, проделав немного дополнительной работы. Реализуйте коммуникацию между подсистемами на основе принципа «необходимого знания», и пусть оно будет действительно необходимым. Помните: проще сначала ограничить взаимодействие, а затем сделать его более свободным, чем пытаться изолировать подсистемы после написания нескольких сотен вызовов между ними. На рис. 5-5 показано, как несколько правил коммуникации могут изменить систему, изображенную на рис. 5-4.

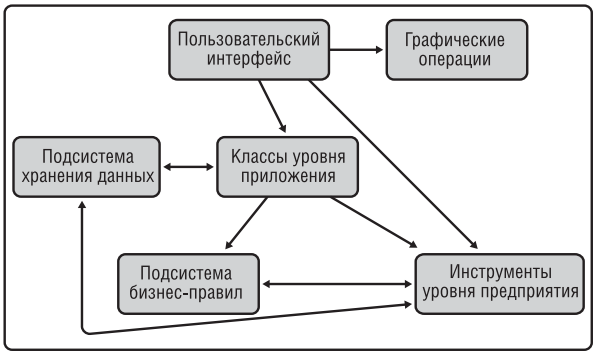



***Рис. 5-5.*** *Определив несколько правил коммуникации, можно существенно упростить взаимодействие подсистем*

Чтобы соединения подсистем были понятными и легкими в сопровождении, старайтесь поддерживать простоту отношений между подсистемами. Самым простым отношением является то, при котором одна подсистема вызывает методы другой. Более сложное отношение имеет место, когда одна подсистема содержит классы другой. Самое сложное отношение — наследование классов одной подсистемы от классов другой.

Придерживайтесь одного разумного правила: диаграмма системного уровня вроде той, что показана на рис. 5-5, должна быть ациклическим графом. Иначе говоря, программа не должна содержать циклических отношений, при которых класс A использует класс B, класс B использует класс C, а класс C — класс A.

При работе над крупными программами и программными комплексами проектирование на уровне подсистем просто необходимо. Если вам кажется, что ваша

программа достаточно мала, чтобы проектирование на уровне подсистем можно было пропустить, хотя бы примите это решение осознанно.

### Часто используемые подсистемы

Некоторые типы подсистем снова и снова используются в разных системах. Ниже приведены те, что встречаются чаще всего.

**Перекрестная ссылка** Об упрощении бизнес-логики путем ее выражения в форме таблиц см. главу 18.

***Подсистема бизнес-правил*** Бизнес-правилами называют законы, директивы, политики и процедуры, реализуемые в компьютерной системе. Например, в случае системы расчета заработной платы бизнес-правилами могли бы быть директивы налогового управления, определяющие разнообразные виды налогов. Дополнительным источником правил могло бы быть соглашение с профсоюзом, регламентирующее оплату сверхурочной работы, отпуска и т. д. При создании программы для агентства по страхованию автомобилей правила могут быть основаны на соответствующих государственных законах.

***Подсистема пользовательского интерфейса*** Изоляция компонентов пользовательского интерфейса в отдельной подсистеме позволяет изменять его, не влияя на остальную программу. Как правило, подсистема пользовательского интерфейса включает несколько подчиненных подсистем или классов, отвечающих за GUI, интерфейс командной строки, работу с меню, управление окнами, справочную систему и т. д.

***Подсистема доступа к БД*** Вы может скрыть детали реализации доступа к БД, чтобы большая часть программы не нуждалась в знании «грязных» подробностей операций над низкоуровневыми структурами и могла работать с данными в терминах бизнес-проблемы. Подсистемы, скрывающие детали реализации, обеспечивают важный уровень абстракции, снижающий сложность программы. Они концентрируют операции над БД в одном месте и снижают вероятность ошибок при работе с данными, а также позволяют легко изменять структуру БД без изменения большей части программы.

***Подсистема изоляции зависимостей от ОС*** Зависимости от ОС следует изолировать в подсистеме по той же причине, что и зависимости от оборудования. Если, например, вы разрабатываете программу для Microsoft Windows, зачем ограничивать себя средой Windows? Изолируйте вызовы Windows в специализированной интерфейсной подсистеме, и если вам позднее захочется перенести программу на платформу Mac OS или Linux, то придется изменить только эту подсистему. Интерфейсная подсистема может быть слишком крупной, чтобы вы могли реализовать ее самостоятельно, однако такие подсистемы уже разработаны и включены в несколько коммерческих библиотек.

### Уровень 3: разделение подсистем на классы

Этот уровень проектирования предполагает определение всех классов системы. Например, подсистема доступа к БД может быть далее разделена на классы доступа к данным и классы хранения данных, а также метаданные БД. На рис. 5-2 показано, как разделить на классы одну из подсистем уровня 2; конечно, три других подсистемы также следует разделить на классы.

**Дополнительные сведения** Проектирование БД хорошо рассмотрено в книге «Agile Database Techniques» (Ambler, 2003).

Кроме того, на этом уровне следует определить детали взаимодействия каждого класса с остальными элементами системы, особенно интерфейс класса. В целом сутью проектирования на данном уровне является декомпозиция подсистем до такого уровня детальности, который позволит реализовать части подсистем в форме отдельных классов.

Разделение подсистем на классы обычно требуется во всех проектах, на реализацию которых уйдет более нескольких дней. В крупных проектах необходимы и второй, и третий уровни проектирования. При работе над совсем небольшим проектом второй уровень проектирования можно пропустить.

**Перекрестная ссылка** Характеристики высококачественных классов см. в главе 6.

#### *Классы и объекты*

Один из важнейших аспектов объектно-ориентированного проектирования — различие между объектами и классами. Объект — это любая конкретная динамическая сущность, имеющая конкретные значения и атрибуты и существующая в период выполнения программы. Класс — это статическая сущность, с которой вы имеете дело, просматривая листинг программы. Например, вы можете объявить класс *Person* (человек), имеющий такие атрибуты, как фамилия, возраст, пол и т. д. В период выполнения вы будете работать с объектами *nancy*, *hank*, *diane*, *tony* и т. д. — иначе говоря, со специфическими экземплярами класса. Если вы знакомы с терминологией БД, различие между классом и объектом аналогично различию между «схемой» и «экземпляром». Класс можно рассматривать как форму для выпечки булочек, а объекты — как сами булочки. В этой книге термины «класс» и «объект» используются неформально и более или менее взаимозаменяемо.

### Уровень 4: разделение классов на методы

Данный уровень проектирования заключается в разделении каждого класса на методы. Некоторые методы уже будут определены на уровне 3, при проектировании интерфейсов классов. На уровне 4 вы детально определите закрытые методы классов. При этом многие методы окажутся простыми, тогда как другие будут включать иерархии методов и потребуют дополнительного проектирования.

Полное определение методов класса часто позволяет лучше понять его интерфейс, что может подтолкнуть к соответствующему изменению интерфейса, т. е. к возвращению на уровень 3.

Четвертый уровень декомпозиции и проектирования часто оставляется отдельным программистам и необходим в любом проекте, требующем более нескольких часов. Формально выполнять этот этап не обязательно, но хотя бы про себя выполнить его нужно.

### Уровень 5: проектирование методов

На этом уровне проектирование заключается в детальном определении функциональности отдельных методов, за что обычно отвечают отдельные программисты, работающие над конкретными методами. Данный уровень может включать такие действия, как написание псевдокода, поиск алгоритмов в книгах, размышление над оптимальной организацией фрагментов метода и написание кода. Этот

**Перекрестная ссылка** О создании высококачественных методов см. главы 7 и 8.

уровень проектирования выполняется во всех случаях, но не всегда осознанно и качественно. На рис. 5-2 он отмечен цифрой 5.

## 5.3. Компоненты проектирования: эвристические принципы

Разработчики ПО любят четкие и ясные правила: «Сделай A, B и C, и это обязательно приведет к X, Y и Z». Мы испытываем гордость, когда находим тайные действия, приводящие к желаемым результатам, и сердимся, если команды работают не так, как описано. Стремление к детерминированному поведению прекрасно согласуется с детальным программированием, при котором строгое внимание к деталям может определить успех или провал программы. Однако проектирование ПО — совсем другая история.

Так как проектирование не является детерминированным, главным аспектом проектирования качественного ПО становится умелое применение набора эффективных эвристических принципов. Ниже мы рассмотрим ряд таких принципов — подходов, способных привести к удачным решениям. Можете считать эвристические принципы правилами выполнения проб при использовании метода проб и ошибок. Несомненно, некоторые из них вам уже известны. Каждый из эвристических принципов будет описан в контексте Главного Технического Императива Разработки ПО — управления сложностью.

### Определите объекты реального мира

Прежде всего следует узнать, не что система выполняет, а над ЧЕМ она это выполняет!

*Бертран Мейер (Bertrand Meyer)*

Первый, и самый популярный, подход к проектированию — «общепринятый» объектно-ориентированный подход — основан на определении объектов реального мира и искусственных объектов.

При проектировании с использованием объектов определите:

- объекты и их атрибуты (методы и данные);
- действия, которые могут быть выполнены над каждым объектом;
- действия, которые каждый объект может выполнять над другими объектами;
- части каждого объекта, видимые другим объектам, т. е. открытые и закрытые части;
- открытый интерфейс каждого объекта.

**Перекрестная ссылка** О проектировании с использованием классов см. главу 6.

Эти часто повторяющиеся действия не обязательно выполнять в указанном порядке. Помните о важности итерации.

***Определите объекты и их атрибуты*** В основе создания программ обычно лежат сущности реального мира. Например, система расчета повременной оплаты может быть основана на таких сущностях, как сотрудники, клиенты, карты учета времени и счета (рис. 5-6).

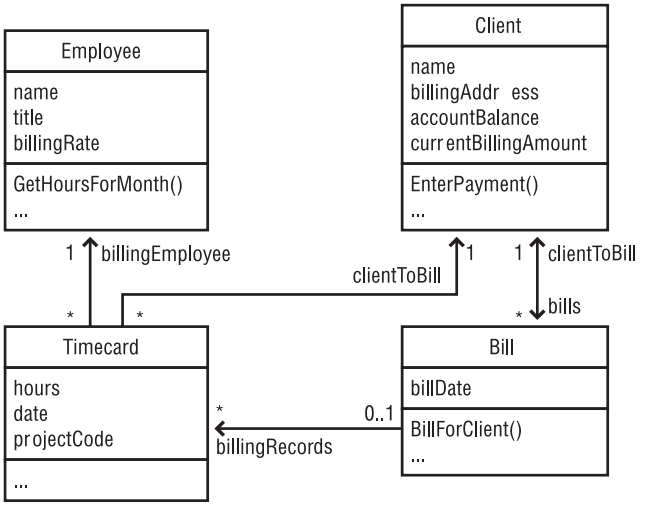



***Рис. 5-6.*** *Эта система расчета оплаты состоит из четырех основных объектов (пример упрощен)*

Определить атрибуты объектов не сложнее, чем сами объекты. Каждый объект имеет характеристики, релевантные для компьютерной программы. Скажем, в системе расчета повременной оплаты объект «сотрудник» обладал бы такими атрибутами, как имя/фамилия, должность и уровень оплаты. С объектом «счет» были бы связаны такие атрибуты, как сумма, имя/фамилия клиента, дата и т. д.

Объектами системы GUI были бы разнообразные окна, кнопки, шрифты и инструменты рисования. При дальнейшем изучении проблемной области вы можете прийти к выводу, что установление однозначного соответствия между объектами программы и объектами реального мира — не самый лучший способ определения объектов, но для начала он тоже неплох.

***Определите действия, которые могут быть выполнены над каждым объектом*** Объекты могут поддерживать самые разные операции. В нашей системе расчета оплаты объект «сотрудник» мог бы поддерживать изменение должности или уровня оплаты, объект «клиент» — изменение реквизитов счета и т. д.

***Определите действия, которые каждый объект может выполнять над другими объектами*** Суть этого этапа ясна из его названия. Двумя универсальными действиями, которые объекты могут выполнять друг над другом, являются включение (containment) и наследование. Какие объекты могут *включать* другие (какие?) объекты? Какие объекты могут быть *унаследованными от* других (каких?) объектов? На рис. 5-6 объект «карта учета времени» может включать объект «сотрудник» и объект «клиент», а объект «счет» может включать карты учета времени. Кроме того, счет может сообщать, что клиент оплатил услуги, а клиент — оплачивать указанную в счете сумму. Более сложная система включала бы дополнительные взаимодействия.

***Определите части каждого объекта, видимые другим объектам*** Один из главных аспектов проектирования — определение частей объекта, которые следует сделать открытыми, и частей, которые следует держать закрытыми. Этого решения требуют и данные, и методы.

**Перекрестная ссылка** О классах и сокрытии информации см. подраздел «Скрывайте секреты (к вопросу о сокрытии информации)» раздела 5.3.

***Определите интерфейс каждого объекта*** Для каждого объекта надо определить формальный синтаксический интерфейс на уровне языка программирования. Данные и методы, которые объект предоставляет в распоряжение остальным объектам, называются «открытым интерфейсом». Части объекта, доступные производным от него объектам, называются «защищенным интерфейсом» объекта. Проектируя программу, обдумайте интерфейсы обоих типов.

Завершая проектирование высокоуровневой объектно-ориентированной организации системы, вы будете использовать два вида итерации: высокоуровневую, направленную на улучшение организации классов, и итерацию на уровне каждого из определенных классов, направленную на детализацию проекта каждого класса.

## Определите согласованные абстракции

Абстракция позволяет задействовать концепцию, игнорируя ее некоторые детали и работая с разными деталями на разных уровнях. Имея дело с составным объектом, вы имеете дело с абстракцией. Если вы рассматриваете объект как «дом», а не как комбинацию стекла, древесины и гвоздей, вы прибегаете к абстракции. Если вы рассматриваете множество домов как «город», вы прибегаете к другой абстракции.

Базовые классы представляют собой абстракции, позволяющие концентрироваться на общих атрибутах производных классов и игнорировать детали конкретных классов при работе с базовым классом. Удачный интерфейс класса — это абстракция, позволяющая сосредоточиться на интерфейсе, не беспокоясь о внутренних механизмах работы класса. Интерфейс грамотно спроектированного метода обеспечивает такую же выгоду на более низком уровне детальности, а интерфейс грамотно спроектированного пакета или подсистемы — на более высоком.

С точки зрения сложности, главное достоинство абстракции в том, что она позволяет игнорировать нерелевантные детали. Большинство объектов реального мира уже является абстракциями некоторого рода. Как я только что сказал, дом — это абстракция окон, дверей, обшивки, электропроводки, водопроводных труб, изоляционных материалов и конкретного способа их организации. Дверь же — это абстракция особого вида организации прямоугольного фрагмента некоторого материала, петель и ручки. А дверную ручку можно считать абстракцией конкретного способа упорядочения медных, никелевых или стальных деталей.

Мы используем абстракции на каждом шагу. Если б, открывая или закрывая дверь, вы должны были иметь дело с отдельными волокнами древесины, молекулами лака и стали, вы вряд ли смогли бы войти в дом или выйти из него. Абстракция — один из главных способов борьбы со сложностью реального мира (рис. 5-7).

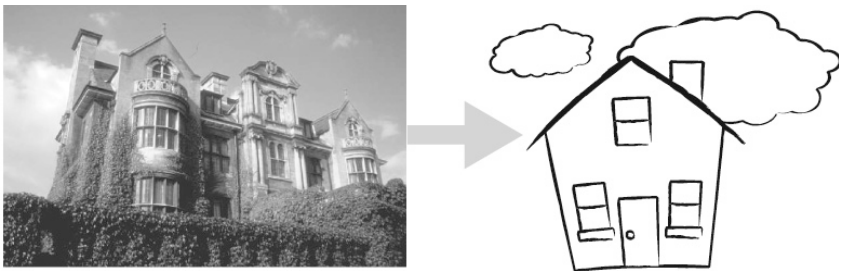


***Рис. 5-7.*** *Абстракция позволяет представить сложную концепцию в более простой форме*

Разработчики ПО иногда создают системы на уровне волокон древесины и молекул лака и стали, из-за чего такие системы становятся слишком сложными и плохо поддаются осмыслению. Если программисты не создают более общие абстракции, разработка системы может завершиться неудачей.

**Перекрестная ссылка** Об абстракции в контексте проектирования классов см. подраздел «Хорошая абстракция» раздела 6.2.

Благоразумные программисты создают абстракции на уровне интерфейсов методов, интерфейсов классов и интерфейсов пакетов (иначе говоря, на уровне дверной ручки, уровне двери и на уровне дома), что способствует более быстрому и безопасному программированию.

## Инкапсулируйте детали реализации

Когда абстракция нас покидает, на помощь приходит инкапсуляция. Абстракция говорит: «Вы можете рассмотреть объект с общей точки зрения». Инкапсуляция добавляет: «Более того, вы не можете рассмотреть объект с иной точки зрения».

Продолжим нашу аналогию: инкапсуляция позволяет вам смотреть на дом, но не дает подойти достаточно близко, чтобы узнать, из чего сделана дверь. Инкапсуляция позволяет вам знать о существовании двери, о том, открыта она или заперта, но при этом вы не можете узнать, из чего она сделана (из дерева, стекловолокна, стали или другого материала), и уж никак не сможете рассмотреть отдельные волокна древесины.

Инкапсуляция помогает управлять сложностью, блокируя доступ к ней (рис. 5-8). В подразделе «Хорошая инкапсуляция» раздела 6.2 инкапсуляция рассматривается подробнее в контексте проектирования классов.

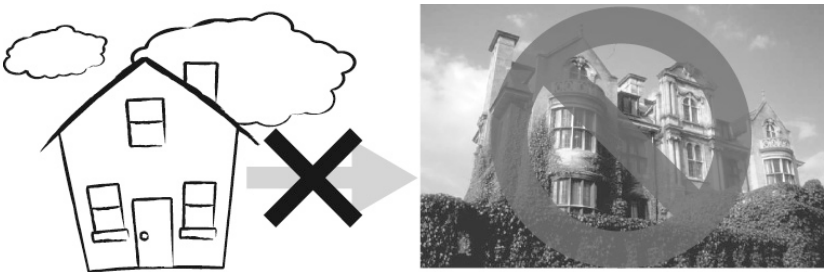


***Рис. 5-8.*** *Инкапсуляция не только представляет сложную концепцию в более простой форме, но и не позволяет взглянуть на какие бы то ни было детали сложной концепции. Что видите, то и получите — и не более того!*

## Используйте наследование, если оно упрощает проектирование

При проектировании ПО часто выясняется, что одни объекты аналогичны другим за исключением нескольких различий. Так, при создании системы расчета зарплаты нужно учесть, что одни сотрудники работают полный день, а другие — неполный. В этом случае наборы данных, ассоциированные с сотрудниками обеих категорий, будут различаться лишь несколькими аспектами. Объектно-ориентированный подход позволяет создать общий тип «сотрудник» и определить сотрудников, работающих полный день, как сотрудников общего типа за исключением нескольких различий. Если операция над объектом «сотрудник» не зависит от его категории, она выполняется так, как если бы объект был сотрудником общего типа. Если же операция зависит от типа сотрудника, она выполняется разными способами.

Определение сходств и различий между такими объектами называется «наследованием», потому что отдельные типы сотрудников, работающих полный и неполный день, наследуют свойства общего типа «сотрудник».

Польза наследования в том, что оно дополняет идею абстракции. Абстракция позволяет представить объекты с разным уровнем детальности. Если помните, на одном уровне мы рассматривали дверь как набор определенных типов молекул, на втором — как набор волокон древесины, а на третьем — как что-то, что защищает нас от воров. Древесина имеет определенные свойства — скажем, вы можете распилить ее пилой или склеить столярным клеем, — при этом и плинтусы, и подоконники имеют общие свойства древесины, но вместе с тем и некоторые специфические свойства.

Наследование упрощает программирование, позволяя создать универсальные методы для выполнения всего, что основано на общих свойствах дверей, и затем написать специфические методы для выполнения специфических операций над конкретными типами дверей. Некоторые операции, такие как *Open()* или *Close()*, будут универсальными для всех дверей: внутренних, входных, стеклянных, стальных — каких угодно. Поддержка языком операций вроде *Open()* или *Close()* при отсутствии информации о конкретном типе двери вплоть до периода выполнения называется *полиморфизмом*. Объектно-ориентированные языки, такие как C++, Java и более поздние версии Microsoft Visual Basic, поддерживают и наследование, и полиморфизм.

Наследование — одно из самых мощных средств объектно-ориентированного программирования. При правильном применении оно может принести большую пользу, однако в обратном случае и ущерб будет немалым. Подробнее см. подраздел «Наследование (отношение «является»)» раздела 6.3.

## Скрывайте секреты (к вопросу о сокрытии информации)

Сокрытие информации — один из основных принципов и структурного, и объектно-ориентированного проектирования. В первом случае сокрытие информации лежит в основе идеи «черных ящиков». Во втором оно дает начало концепциям инкапсуляции и модульности и связано с концепцией абстракции. Сокрытие информации — одна из самых конструктивных идей в мире разработки ПО, и сейчас мы рассмотрим ее подробнее.

Впервые сокрытие информации было представлено на суд общественности в 1972 г. Дэвидом Парнасом (David Parnas) в статье «On the Criteria to Be Used in Decomposing Systems Into Modules (О критериях, используемых при декомпозиции систем на модули)». С сокрытием информации тесно связана идея «секретов» — аспектов проектирования и реализации, которые разработчик ПО решает скрыть в каком-то месте от остальной части программы.

В юбилейном 20-летнем издании книги «Мифический человеко-месяц» Фред Брукс пришел к выводу, что критика сокрытия информации была одной из ошибок, допущенных им в первом издании книги. «Парнас был прав в отношении сокрытия информации, а я ошибался», — признал он (Brooks, 1995). Барри Бом сообщил, что сокрытие информации — мощный метод избавления от повторной работы, и указал, что оно особенно эффективно в инкрементных средах с высоким уровнем изменений (Boehm, 1987).

В контексте Главного Технического Императива Разработки ПО сокрытие информации оказывается особенно мощным эвристическим принципом, так как все его аспекты и даже само название подчеркивают *сокрытие сложности*.

### Секреты и право на личную жизнь

При сокрытии информации каждый класс (пакет, метод) характеризуется аспектами проектирования или конструирования, которые он скрывает от остальных классов. Секретом может быть источник вероятных изменений, формат файла, реализация типа данных или область, изоляция которой требуется для сведения к минимуму вреда от возможных ошибок. Класс должен скрывать эту информацию и защищать свое право на «личную жизнь». Небольшие изменения системы могут влиять на несколько методов класса, но не должны распространяться за его интерфейс.

> Интерфейсы классов должны быть полными и минимальными.
>
> *Скотт Мейерс (Scott Meyers)*

Один из важнейших аспектов проектирования класса — принятие решения о том, какие свойства сделать доступными вне класса, а какие оставить секретными. Класс может включать 25 методов, предоставляя доступ только к пяти из них и используя остальные 20 внутренне. Класс может использовать несколько типов данных, не раскрывая сведений о них. Этот аспект проектирования классов называют «видимостью», так как он определяет, какие свойства класса «видимы» или «доступны» извне.

Интерфейс класса должен сообщать как можно меньше о внутренней работе класса. В этом смысле класс во многом похож на айсберг, большая часть которого скрыта под водой (рис. 5-9).

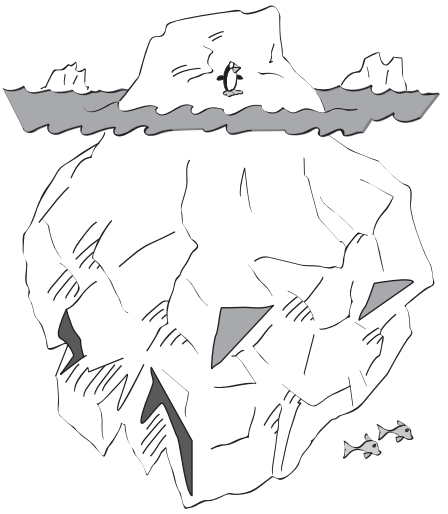


***Рис. 5-9.*** *Хороший интерфейс класса похож на верхушку айсберга: большую часть класса он оставляет скрытой*

Как и любой другой аспект проектирования, разработка интерфейса класса — итеративный процесс. Если приемлемый интерфейс класса не удается создать с первого раза, сделайте еще несколько попыток, пока он не стабилизируется. Если интерфейс не стабилизируется, попробуйте другой подход.

### Пример сокрытия информации

Допустим, вы пишете программу, каждый объект которой должен иметь уникальный идентификатор, хранящийся в переменной-члене *id*. Один подход к проектированию может заключаться в применении целочисленных идентификаторов и хранении максимального на данный момент идентификатора в глобальной переменной *g_maxId*. При создании новых объектов вы можете — скажем, в конструкторе каждого объекта — просто выполнять команду *id = ++g_maxId*, что гарантирует уникальность идентификаторов, и требует абсолютно минимального кода при создании каждого объекта. Разве это может привести к каким-нибудь неприятностям?

Может. Что, если вы захотите зарезервировать диапазоны идентификаторов для определенных целей? Что, если для повышения защищенности программы вы захотите назначать идентификаторы в другом порядке? А если вы захотите повторно задействовать идентификаторы уничтоженных объектов? Или включить в программу диагностический тест, проверяющий, не превысило ли число идентификаторов допустимый предел? Если, назначая идентификаторы, вы распространите команды *id = ++g_maxId* по всей программе, вам придется изменить каждую из них. Кроме того, этот подход небезопасен в многопоточной среде.

Способ генерации новых идентификаторов является тем аспектом проектирования, который следует скрыть. Применив команду *++g_maxId*, вы раскроете сведения о том, что новый идентификатор создается просто путем увеличения переменной *g_maxId*. Если же вместо этого вы используете команды *id = NewId()*, вы скроете

информацию о способе создания новых идентификаторов. Сам метод *NewId()* может состоять из единственной строки *return ( ++g_maxId )* или ее эквивалента, однако, если вы позднее решите зарезервировать определенные диапазоны идентификаторов для специфических целей или повторно использовать старые идентификаторы, вам придется изменить только метод *NewId()*, но не десятки команд *id = NewId()*. Какими бы сложными ни были изменения метода *NewId()*, они не повлияют ни на какую другую часть программы.

Допустим теперь, что вам понадобилось изменить тип идентификатора с целочисленного на строковый. Если по всей программе у вас разбросаны объявления вроде *int id*, метод *NewId()* не поможет. В этом случае вам тоже придется просмотреть всю программу и внести десятки или сотни изменений.

Итак, тип идентификатора — это тоже секрет, который следует скрыть. Показывая, что идентификаторы — целые числа, вы поощряете программистов выполнять над ними такие операции, как >, < и =. Программируя на C++, вы могли бы не объявлять идентификаторы как *int*, а назначить им при помощи директивы *typedef* пользовательский тип *IdType* соответствующий тому же *int*. Или же вы могли бы создать простой класс *IdType*. Повторю еще раз: сокрытие аспектов проектирования позволяет значительно уменьшить объем кода, затрагиваемого изменениями.

Сокрытие информации полезно на всех уровнях проектирования: от применения именованных констант вместо литералов до создания типов данных и проектирования классов, методов и подсистем.

### Две категории секретов

Связанные с сокрытием информации секреты относятся к двум общим категориям:

- секреты, которые скрывают сложность, позволяя программистам забыть о ней при работе над остальными частями программы;
- секреты, которые скрывают источники изменений с целью локализации результатов возможных изменений.

В число источников сложности входят сложные типы данных, файловые структуры, булевы тесты, запутанные алгоритмы и т. д. Источники изменений будут описаны немного позднее.

### Барьеры, препятствующие сокрытию информации

В некоторых случаях скрыть информацию невозможно, однако большинство барьеров, препятствующих сокрытию информации, является умственными и обусловлены привыканием к другим методикам.

**Дополнительные сведения** Отдельные фрагменты этого раздела взяты из статьи «Designing Software for Ease of Extension and Contraction» (Parnas, 1979).

***Избыточное распространение информации*** Зачастую сокрытию информации препятствует избыточное распространение информации по системе. Так, жесткое кодирование литерала *100* во многих местах программы децентрализует ссылки на него. Лучше скрыть эту информацию в одном месте — скажем, при помощи константы *MAX_EMPLOYEES*, для изменения значения которой придется изменить только одну строку кода.

Еще один пример избыточного распространения информации — распределение по системе кода взаимодействия с пользователями. При этом в случае изменения способа взаимодействия — например, при замене графического интерфейса на интерфейс командной строки — придется изменить почти весь код. Лучше сконцентрировать взаимодействие с пользователями в одном классе (пакете, подсистеме), который можно было бы изменить, не влияя на всю систему.

**Перекрестная ссылка** О доступе к глобальным данным при помощи интерфейсов классов см. подраздел «Используйте методы доступа вместо глобальных данных» раздела 13.3.

В качестве другого примера приведу повсеместное использование глобального элемента данных, такого как массив данных о сотрудниках, поддерживающий до 1000 элементов. Если программа будет обращаться к глобальным данным напрямую, информация о реализации элемента данных — скажем, то, что это массив, способный включать до 1000 элементов, — распространится по всей программе. Если программа будет обращаться к данным только через методы доступа, детали реализации будут известны только этим методам.

***Круговая зависимость*** Более тонким барьером, мешающим сокрытию информации, является круговая зависимость, когда, например, метод класса A вызывает метод класса B, а метод класса B — метод класса A.

Избегайте таких зависимостей: они осложняют тестирование системы, не позволяя протестировать ни один из классов, пока не будет реализована хотя бы часть второго класса.

***Ошибочное представление о данных класса как о глобальных данных*** Если вы добросовестный программист, возможна еще одна преграда на пути к эффективному сокрытию информации: вы можете рассматривать данные класса как глобальные данные, избегая их из-за соответствующих проблем. Всем известно, что дорога в ад программирования вымощена глобальными переменными, однако использовать данные класса гораздо безопаснее.

Глобальные данные имеют два главных недостатка: методы, обращающиеся к глобальным данным, не знают о том, что другие методы тоже обращаются к этим данным, или же методы знают об этом, но не знают, что именно другие методы делают с глобальными данными. Данные класса этих недостатков не имеют. Непосредственный доступ к данным класса ограничен несколькими методами этого же класса, которые знают и о том, что другие методы также работают с данными, и о том, что это за методы.

Конечно, это предполагает, что система включает грамотно спроектированные небольшие классы. Если программа использует огромные классы, включающие десятки методов, различие между данными класса и глобальными данными стирается, и данные класса приобретают многие недостатки, характерные для глобальных данных.

**Перекрестная ссылка** О повышении производительности на уровне кода см. главы 25 и 26.

***Кажущееся снижение производительности*** Наконец, отказ от сокрытия информации может объясняться стремлением избежать снижения производительности и на уровне архитектуры, и на уровне кода. В обоих случаях волноваться не о чем. При проектировании архитектуры сокрытие информации не конфликтует с производительностью. Помня и о сокрытии информации, и о производительности, вы сможете достичь обеих целей.

Производительность на уровне кода вызывает еще больше беспокойств. Разработчикам кажется, что опосредованный доступ к данным снизит производительность программы в период выполнения из-за дополнительных затрат на создание объектов, вызовы методов и т. д. Эти волнения преждевременны. Пока вы не оцените производительность системы и не найдете узкие места, лучшим способом подготовки к повышению производительности на уровне кода является модульное проектирование. Позже, определив в коде «горячие точки», вы оптимизируете отдельные классы и методы, не затрагивая остальную часть системы.

### Важность сокрытия информации

Сокрытие информации относится к тем немногим теоретическим подходам, польза которых уже долгое время неоспоримо подтверждается на практике (Boehm, 1987a). Было обнаружено, что крупные программы, использующие сокрытие информации, вчетверо легче модифицировать, чем программы, его не использующие (Korson and Vaishnavi, 1986). Более того, сокрытие информации — один из основных принципов и структурного, и объектно-ориентированного проектирования.

Сокрытие информации обладает уникальной эвристической силой, уникальной способностью подталкивать разработчиков к эффективным проектным решениям. Традиционное объектно-ориентированное проектирование предоставляет мощные эвристические средства моделирования мира в терминах объектов, но объектный подход не помог бы вам догадаться, что идентификатор следует объявить как *IdType*, а не как *int*. Разработчик, использующий объектно-ориентированный подход, спросил бы: «Рассматривать ли идентификатор как объект?» В зависимости от принятых в проекте стандартов кодирования утвердительный ответ мог бы означать, что программист должен написать конструктор, деструктор, операторы копирования и присваивания, закомментировать все это и сохранить в системе управления конфигурацией. Но скорее всего программист решил бы: «Нет, не стоит создавать целый класс ради какого-то идентификатора. Использую просто *int*».

Смотрите: эффективный вариант проектирования — простое сокрытие типа данных идентификатора — даже не был рассмотрен! Если бы вместо этого разработчик спросил: «Не скрыть ли информацию об идентификаторе?» — он, возможно, решил бы объявить собственный тип *IdType* как синоним *int*. Различие между объектно-ориентированным проектированием и сокрытием информации в этом примере не сводится к простому несоответствию явных правил и предписаний. Принятое в соответствии с принципом сокрытия информации решение прекрасно согласуется с объектно-ориентированным подходом. Вместо этого различие относится к области эвристики: размышление над сокрытием информации может указать на такие варианты проектирования, которые при использовании объектно-ориентированного подхода остались бы незамеченными.

Сокрытие информации может пригодиться при проектировании открытого интерфейса класса. Теория и практика проектирования классов во многом расходятся, и многие разработчики, решая, чту включить в открытый интерфейс класса, думают прежде всего об удобном интерфейсе, а это обычно приводит к раскрытию почти всей информации об устройстве класса. Опыт подсказывает мне,

что некоторые программисты скорее раскрыли бы все закрытые данные класса, чем написали 10 дополнительных строк для защиты его секретов.

Вопрос «Что этот класс должен скрывать?» обнажает самую суть проблемы проектирования интерфейса. Если функцию или данные можно включить в открытый интерфейс класса, не раскрыв его секретов, сделайте это. В противном случае воздержитесь от такого решения.

Размышление о том, чту скрыть, способствует принятию удачных решений на всех уровнях проектирования. Оно подталкивает к применению именованных констант вместо чисел на уровне конструирования, помогает выбирать удачные имена методов классов и их параметров и указывает на грамотные варианты декомпозиции и реализации взаимодействия классов и подсистем на уровне системы.

Почаще задавайте себе вопрос «Что мне скрыть?», и вы удивитесь, сколько проблем проектирования растает на ваших глазах

## Определите области вероятных изменений

**Дополнительные сведения** Подход, описанный в этом разделе, взят из статьи «Designing Software for Ease of Extension and Contraction» (Parnas, 1979).

Исследования показали, что всем лучшим проектировщикам свойственно предвосхищать изменения (Glass, 1995). Обеспечение легкости адаптации программы к возможным изменениям относится к самым сложным аспектам проектирования. Его цель заключается в изоляции нестабильных областей, позволяющей ограничить следствия изменений одним методом, классом или пакетом. Вот как проходит подготовка к изменениям.

1. **Определите элементы, изменение которых кажется вероятным.** Если вы выработали адекватные требования, они включают список потенциальных изменений и оценки вероятности каждого из них. В этом случае определить вероятные изменения легко. Если требования не описывают потенциальные изменения, ниже вы найдете список областей, которые меняются чаще всего независимо от типа проекта.
2. **Отделите элементы, изменение которых кажется вероятным.** Создайте отдельный класс для каждого нестабильного компонента, определенного в п. 1, или разработайте классы, включающие несколько нестабильных компонентов, изменение которых скорее всего будет одновременным.
3. **Изолируйте элементы, изменение которых кажется вероятным.** Спроектируйте интерфейсы между классами так, чтобы они не зависели от потенциальных изменений. Спроектируйте интерфейсы так, чтобы изменения ограничивались только внутренними частями классов. Изменение класса должно оставаться незаметным для любых других классов. Интерфейс класса должен защищать его секреты.

Ниже описано несколько областей, изменяющихся чаще всего.

**Перекрестная ссылка** Один из самых эффективных способов предвосхищения изменений — табличное управление (см. главу 18).

***Бизнес-правила*** Необходимость изменения ПО часто объясняется изменениями бизнес-правил. Оно и понятно: конгресс может изменить систему налогообложения, профсоюзы — пересмотреть условия контрактов и т. д. Если вы соблюдаете принцип сокрытия информации, логика, основанная на этих правилах, не будет распространена на всю

программу. Она будет скрыта в одном темном уголке системы, пока не придет время ее изменить.

***Зависимости от оборудования*** Примерами модулей, зависимых от оборудования, могут служить интерфейсы между программой и разными типами мониторов, принтеров, клавиатур, дисководов, звуковых плат и сетевых устройств. Изолируйте зависимости от оборудования в отдельной подсистеме или отдельном классе. Это облегчает адаптацию программы к новой аппаратной среде, а также помогает разрабатывать ПО для нестабильных версий устройств. Вы можете разработать ПО, моделирующее взаимодействие с конкретным устройством, и создать подсистему аппаратного интерфейса, использующую эту модель, пока устройство нестабильно или недоступно. Когда устройство будет готово к работе, подсистему интерфейса можно будет отключить от модели и подключить к устройству.

***Ввод-вывод*** На чуть более высоком в сравнении с аппаратными интерфейсами уровне проектирования частой областью изменений является ввод-вывод. Если ваше приложение создает собственные файлы данных, его усложнение вполне может потребовать изменения формата файлов. Аспекты формата ввода-вывода данных, относящиеся к пользовательскому уровню, такие как позиционирование и число полей на странице, их последовательность и т. д., изменяются не менее часто. В общем, анализ всех внешних интерфейсов на предмет возможных изменений — благоразумная идея.

***Нестандартные возможности языка*** Большинство версий языков поддерживает нестандартные расширения, облегчающие работу программистов. Расширения — палка о двух концах, потому что в другой среде — будь то другая аппаратная платформа, реализация языка другим производителем или новая версия языка, выпущенная тем же производителем, — они могут оказаться недоступны.

Если вы применяете нестандартные расширения языка, скройте работу с ними в отдельном классе, чтобы его можно было заменить при адаптации приложения к другой среде. Аналогично, используя библиотечные методы, доступные не во всех средах, скройте их за интерфейсом, поддерживающим все нужные среды.

***Сложные аспекты проектирования и конструирования*** Скрывайте сложные аспекты проектирования и конструирования, потому что их частенько приходится реализовывать заново. Отделите их и минимизируйте влияние, которое может оказать их неудачное проектирование или конструирование на остальные части системы.

***Переменные статуса*** Переменные статуса характеризуют состояние программы и изменяются чаще, чем большинство других видов данных. Так, разработчики, определившие переменную статуса ошибки как булеву переменную, вполне могут позднее прийти к выводу, что для этого лучше было бы использовать перечисление со значениями *ErrorType_None*, *ErrorType_Warning* и *ErrorType_Fatal*:

Использование переменных статуса можно сделать более гибким и понятным минимум двумя способами.

- В качестве переменных статуса примените не булевы переменные, а перечисления. Диапазон поддерживаемых переменными статуса состояний часто приходится расширять, что в случае перечисления требует лишь перекомпиляции

программы, а не масштабной ревизии всех фрагментов кода, выполняющих проверку переменной.

- Вместо непосредственной проверки переменной используйте методы доступа. Так вы сохраните возможность реализации более сложного механизма определения состояния. Например, если вы захотите проверять комбинацию переменной статуса ошибки и переменной текущего функционального состояния, вам будет легко реализовать это, если проверка будет скрыта в методе, и гораздо сложнее, если механизм проверки будет жестко закодирован во многих местах программы.

***Размеры структур данных*** Объявляя массив из *100* элементов, вы раскрываете информацию, которую никто знать не должен. Защищайте право на личную жизнь! Сокрытие информации не всегда требует создания целого класса. Иногда для этого достаточно именованной константы: например, *MAX_EMPLOYEES* позволяет скрыть число *100*.

### Предвосхищение изменений разного масштаба

**Перекрестная ссылка** Рассматриваемый в этом разделе подход к предвосхищению изменений не связан с заблаговременным проектированием или кодированием (см. подраздел «Программа содержит код, который может когда-нибудь понадобиться» раздела 24.2).

Обдумывая потенциальные изменения системы, проектируйте ее так, чтобы влияние изменений было обратно пропорционально их вероятности. Если вероятность изменения высока, убедитесь, что систему будет легко адаптировать к нему. С большим влиянием на несколько классов системы можно смириться лишь в случае крайне маловероятных изменений. Грамотные проектировщики также принимают во внимание цену предвосхищения изменений. Если изменение маловероятно, но его легко предугадать, рассмотрите его внимательнее, чем более вероятное изменение, которое трудно спланировать.

**Дополнительные сведения** Это обсуждение основано на подходе, описанном в статье «On the design and development of program families» (Parnas, 1976).

Один хороший метод определения областей вероятных изменений подразумевает, что вы должны сначала определить минимальное подмножество фрагментов программы, необходимых пользователям. Это подмножество составляет ядро системы, и его изменения маловероятны. Затем вы определяете минимальные инкрементные приращения системы. Они могут быть совсем небольшими, даже тривиальными. Вместе с функциональными изменениями рассматривайте также качественные изменения программы: обеспечение безопасности в многопоточной среде, поддержку механизмов локализации и т. д. Эти области потенциальных улучшений являются потенциальными изменениями системы; спроектируйте эти области, используя принципы сокрытия информации. Определив ядро в самом начале, вы поймете, какие компоненты системы на самом деле являются дополнениями, и сможете с этого момента экстраполировать и скрывать аспекты возможных изменений программы.

## Поддерживайте сопряжение слабым

Сопряжение характеризует силу связи класса или метода с другими классами или методами. Наша цель — создать классы и методы, имеющие немногочисленные, непосредственные, явные и гибкие отношения с другими классами, что еще на-

зывают «слабым сопряжением» (loose coupling)». В контекстах классов и методов концепция сопряжения одна и та же, так что при обсуждении сопряжения буду называть методы и классы «модулями».

Сопряжение модулей должно быть достаточно слабым, чтобы одни модули могли с легкостью использовать другие. Например, железнодорожные вагоны соединяются с помощью крюков, которые при столкновении двух вагонов защелкиваются. Представьте, как бы все усложнилось, если бы вагоны нужно было соединять при помощи болтов, набора тросов или если бы вы могли соединить между собой только определенные типы вагонов. Механизм соединения вагонов эффективен потому, что он максимально прост. Соединения между программными модулями также должны быть как можно проще.

Старайтесь создавать модули, слабо зависящие от других модулей. Отношения модулей должны напоминать отношения деловых партнеров, а не сиамских близнецов. Скажем, метод *sin()* (синус) сопряжен слабо, так как нужную информацию он получает в форме одного значения — угла в градусах. Метод *InitVars* ( *var1, var2, var3, ..., varN* ) сопряжен жестче, поскольку многие детали его работы становятся известными вызывающему модулю по передаваемым значениям. Два класса, зависящих от того, как каждый из них использует одну глобальную переменную, сопряжены еще жестче.

### Критерии оценки сопряжения

Ниже описаны критерии, позволяющие оценить сопряжение модулей.

***Объем*** Объем связи характеризует число соединений между модулями. Чем их меньше, тем лучше, поскольку модуль, имеющий более компактный интерфейс, легче связать с другими модулями. Метод, принимающий один параметр, слабее сопряжен с вызывающими его модулями, чем метод, принимающий шесть параметров. Класс, имеющий четыре грамотно определенных открытых метода, слабее сопряжен с модулями, которые его используют, чем класс, предоставляющий 37 открытых методов.

***Видимость*** Видимостью называют заметность связи между двумя модулями. Программирование не служба в ЦРУ — никто не похвалит вас за удачную маскировку. Оно больше похоже на рекламу: вам следует делать связи между модулями как можно более крикливыми. Передача данных посредством списка параметров формирует очевидную связь, и это удачный вариант. Передача информации другому модулю в глобальных данных является замаскированной и потому неудачной связью. Описание связи, осуществляемой через глобальные данные, в документации делает ее более явной и является чуть более удачным подходом.

***Гибкость*** Гибкость характеризует легкость изменения связи между модулями. Идеальная связь должна быть как можно гибче. Гибкость частично определяется другими аспектами связанности, но в то же время отличается от них. Положим, у вас есть метод *LookupVacationBenefit()*, определяющий длительность отпуска сотрудника на основании даты его приема на работу и должности. Допустим далее, что в другом модуле у вас есть объект *employee* (сотрудник), содержащий, помимо всего прочего, информацию о должности и дате приема на работу, и что этот модуль передает объект *employee* в метод *LookupVacationBenefit()*.

С точки зрения других критериев, эти два модуля кажутся слабо сопряженными: связь двух модулей посредством объекта *employee* очевидна и является единственной. Теперь предположим, что вам нужно использовать модуль *LookupVacationBenefit()* из третьего модуля, владеющего информацией о дате приема сотрудника на работу и его должности, но хранит ее не в объекте *employee*. В этот момент модуль *LookupVacationBenefit()* начинает вести себя гораздо менее дружелюбно, не желая связываться с новым модулем.

Чтобы третий модуль мог обратиться к модулю *LookupVacationBenefit()*, он должен знать о существовании класса *Employee*. Он мог бы подделать объект *employee*, используя лишь два поля, но тогда он должен был бы знать внутренние детали работы метода *LookupVacationBenefit()*: ему была бы необходима уверенность в том, что метод *LookupVacationBenefit()* использует только два этих поля. Такое решение было бы небрежным и безобразным. Второй вариант мог бы заключаться в таком изменении метода *LookupVacationBenefit()*, чтобы вместо объекта *employee* он принимал должность сотрудника и дату его приема на работу. В обоих случаях первоначальный модуль оказывается на самом деле гораздо менее гибким, чем казалось сначала.

Возможен и счастливый конец этой истории: недружелюбный модуль сможет завести друзей, если пожелает быть гибким — если вместо объекта *employee* он согласится принимать должность и дату приема сотрудника на работу.

Короче, чем проще вызывать модуль из других модулей, тем слабее он сопряжен, и это хорошо, потому что такой модуль более гибок и прост в сопровождении. Создавая структуру программы, делите ее на блоки с учетом их взаимосвязанности. Если бы программа была куском дерева, его следовало бы расщепить параллельно волокнам.

### Виды сопряжения

Самые распространенные виды сопряжения описаны ниже.

***Простое сопряжение посредством данных-параметров*** Два модуля сопряжены таким способом, если между ними передаются только элементарные типы данных, причем передаются через списки параметров. Этот вид сопряжения нормален и приемлем.

***Простое сопряжение посредством объекта*** Модуль сопряжен с объектом этим способом, если он создает экземпляр данного объекта. С этим видом сопряжения также все в порядке.

***Сопряжение посредством объекта-параметра*** Два модуля сопряжены друг с другом объектом-параметром, если Объект 1 требует, чтобы Объект 2 передал ему Объект 3. Этот вид сопряжения жестче, чем тот вид, при котором Объект 1 требует от Объекта 2 только примитивных типов данных, потому что Объект 2 должен обладать информацией об Объекте 3.

***Семантическое сопряжение*** Самый коварный тип сопряжения имеет место тогда, когда один модуль использует не какой-то синтаксический элемент другого модуля, а некоторые семантические знания о внутренней работе этого модуля. Некоторые примеры такого вида сопряжения описаны ниже.

- Модуль 1 передает в Модуль 2 управляющий флаг, определяющий дальнейшую работу Модуля 2. Этот подход подразумевает, что Модуль 1 должен сделать предположения о внутренней работе Модуля 2, а именно о том, что Модуль 2 собирается делать с управляющим флагом. Если Модуль 2 определяет для управляющего флага специфический тип данных (перечисление или объект), этот вид сопряжения, вероятно, будет вполне приемлем.
- Модуль 2 использует глобальные данные после их изменения Модулем 1. При этом Модуль 2 предполагает, что Модуль 1 был вызван в нужное время и изменил данные так, как нужно Модулю 2.
- Интерфейс Модуля 1 утверждает, что метод *Module1.Initialize()* должен быть вызван до метода *Module1.Routine()*. Модуль 2 знает, что *Module1.Routine()* както вызывает метод *Module1.Initialize()*, поэтому он просто создает экземпляр Модуля 1 и вызывает *Module1.Routine()* без предварительного вызова метода *Module1.Initialize()*.
- Модуль 1 передает Объект в Модуль 2. Модуль 1 знает, что Модуль 2 использует только три метода Объекта из семи, поэтому он инициализирует Объект лишь частично, только теми данными, что нужны этим трем методам.
- Модуль 1 передает в Модуль 2 Базовый Объект. Модуль 2 знает, что на самом деле Модуль 1 передал ему Производный Объект, поэтому он приводит тип Базового Объекта к типу Производного Объекта и вызывает методы, специфические для Производного Объекта.

Семантическое сопряжение опасно тем, что изменение кода в используемом модуле может так нарушить работу использующего модуля, что компилятор этого не определит. Обычно это приводит к очень тонким проблемам, которые никто не соотносит с изменениями используемого модуля, что превращает отладку в сизифов труд.

Суть слабого сопряжения в том, что грамотно спроектированный модуль предоставляет дополнительный уровень абстракции: разработав его, вы можете принимать его как данное. Это снижает общую сложность программы и позволяет сосредоточиваться в каждый момент времени только на одном аспекте. Если для использования модуля нужно учитывать сразу несколько аспектов: механизм внутренней работы, изменения глобальных данных, неясную функциональность, — сила абстракции исчезает, и модуль перестает облегчать управление сложностью.

Классы и методы — главные интеллектуальные инструменты снижения сложности. Если они не упрощают вашу работу, они не исполняют свои обязанности.

## Старайтесь использовать популярные шаблоны проектирования

http://cc2e.com/0585

Шаблоны проектирования — это готовые шаблоны, позволяющие решать частые проблемы разработки. Конечно, есть проблемы, требующие совершенно новых решений, но большинство уже встречалось разработчикам, поэтому их можно решить, применяя проверенные подходы, или шаблоны. В число популярных шаблонов проектирования входят Адаптер, Мост, Декоратор, Фасад, Фабричный метод, Наблюдатель,

Одиночка, Стратегия и Шаблонный метод. О шаблонах проектирования см. книгу «Design Patterns» Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса (Gamma, Helm, Johnson, and Vlissides, 1995).

Шаблоны имеют ряд достоинств, не характерных для полностью самостоятельного проектирования программы.

***Шаблоны снижают сложность, предоставляя готовые абстракции*** Если вы скажете: «В этом фрагменте для создания экземпляров производных классов применяется шаблон "Фабричный метод"», — другие программисты поймут, что ваш код включает богатый набор взаимодействий и протоколов программирования, специфических для названного шаблона.

Шаблон «Фабричный метод» позволяет создавать экземпляры любого класса, производного от указанного базового класса, причем отдельные производные классы отслеживаются только самим «Фабричным методом». Обсуждение шаблона «Фабричный метод» см. в разделе «Replace Constructor with Factory Method» (Замена конструктора на «Фабричный метод») книги «Refactoring» (Fowler, 1999).

Если вы будете использовать шаблоны, другие программисты легко поймут вы-бранный вами подход к проектированию без подробного обсуждения кода.

***Шаблоны снижают число ошибок, стандартизируя детали популярных решений*** Проблемы проектирования содержат нюансы, которые полностью проявляются только после решения проблемы один или два раза (или три, или четыре, или...). Шаблоны — это стандартизованные способы решения частых проблем, заключающие мудрость, накопленную за годы попыток решения этих проблем, и исправления неудачных попыток.

Так что, с концептуальной точки зрения, применение шаблона проектирования похоже на использование библиотеки кода вместо написания собственного кода. Многие программисты рано или поздно решают создать собственный вариант алгоритма быстрой сортировки, но каковы шансы, что его первая версия окажется безошибочной? Так же и в проектировании: многие проблемы довольно похожи на уже решенные задачи, и при столкновении с ними изобретать велосипед ни к чему.

***Шаблоны имеют эвристическую ценность, указывая на возможные варианты проектирования*** Проектировщик, знакомый с популярными шаблонами, может с легкостью перебрать список шаблонов и спросить себя: «Какие из них соответствуют моей проблеме проектирования?» Перебрать набор известных вариантов гораздо проще, чем создавать собственное решение с нуля. Кроме того, код, основанный на популярном шаблоне, будет понятнее, чем код, полностью разработанный самостоятельно.

***Шаблоны упрощают взаимодействие между разработчиками, позволяя им общаться на более высоком уровне*** Шаблоны проектирования не только помогают управлять сложностью, но и способны ускорить обсуждение проектов, позволяя разработчикам размышлять и делиться мыслями на более высоком уровне. Если вы скажете: «Не могу решить, какой шаблон следует использовать в данной ситуации: "Создатель" или "Фабричный метод"», — вы в нескольких словах сообщите очень подробную информацию — конечно, если и вам, и вашему собеседнику известны эти шаблоны. Представьте, насколько больше времени потребовалось

бы для обсуждения деталей кода шаблонов «Создатель» и «Фабричный метод» и сравнения этих двух подходов.

Если вы еще не сталкивались с шаблонами проектирования, изучите табл. 5-1, где описаны некоторые из самых популярных шаблонов.

**Табл. 5-1. Популярные шаблоны проектирования**

| Шаблон | Описание |
|---|---|
| Абстрактная фабрика (Abstract Factory) | Поддерживает создание наборов родственных объектов путем определения вида набора, но не вида каждого отдельного объекта. |
| Адаптер (Adapter) | Преобразует интерфейс класса в другой интерфейс. |
| Мост (Bridge) | Создает интерфейс и реализацию, так что их можно изменять независимо друг от друга. |
| Компоновщик (Composite) | Состоит из объекта, содержащего дополнительные объекты такого же типа, позволяя клиентскому коду взаимодействовать с объектом верхнего уровня и не заботиться о детальных объектах. |
| Декоратор (Decorator) | Динамически назначает объекту виды ответственности без создания отдельных подклассов для каждой возможной конфигурации видов ответственности. |
| Фасад (Facade) | Предоставляет согласованный интерфейс к коду, который в противном случае не предоставлял бы согласованного интерфейса. |
| Фабричный метод (Factory Method) | Создает экземпляры классов, производных от конкретного базового класса, причем отдельные производные классы отслеживаются только «Фабричным методом». |
| Итератор (Iterator) | Этот серверный объект предоставляет доступ к каждому элементу набора в последовательном порядке. |
| Наблюдатель (Observer) | Поддерживает синхронизацию нескольких объектов, при которой объект уведомляет набор связанных объектов об изменениях любого члена набора. |
| Одиночка (Singleton) | Предоставляет глобальный доступ к классу, который может иметь один и только один экземпляр. |
| Стратегия (Strategy) | Определяет набор динамически взаимозаменяемых алгоритмов или видов поведения. |
| Шаблонный метод (Template Method) | Определяет структуру алгоритма, оставляя некоторые детали реализации подклассам. |

Если раньше вы не встречались с шаблонами проектирования, при взгляде на табл. 5-1 у вас может возникнуть мысль: «Почти все эти идеи мне уже знакомы». Этим во многом и объясняется ценность шаблонов проектирования. Они известны большинству опытных программистов, а присвоение шаблонам запоминающихся названий позволяет быстро и эффективно делиться мыслями.

С шаблонами связаны две ловушки. Первая — насильственная адаптация кода к какому-нибудь шаблону. Иногда легкое изменение кода в соответствии с известным шаблоном может сделать код более понятным. Но если адаптация кода к стандартному шаблону требует слишком крупного изменения, это может привести к усложнению программы.

Вторая — применение шаблона, продиктованное не целесообразностью, а желанием испытать шаблон в деле.

Вообще применение шаблонов проектирования — это эффективный инструмент управления сложностью. Некоторые хорошие книги по этой теме указаны в конце главы.

## Другие эвристические принципы

В предыдущих разделах были рассмотрены основные эвристические принципы проектирования ПО. Ниже описаны менее полезные, однако заслуживающие упоминания эвристические принципы.

### Стремитесь к максимальной связности

Понятие связности (cohesion) возникло в области структурного проектирования и обычно обсуждается в том же контексте, что и сопряжение (coupling). Связность характеризует то, насколько хорошо все методы класса или все фрагменты метода соответствуют главной цели, — иначе говоря, насколько сфокусирован класс. Классы, состоящие из очень похожих по функциональности блоков, обладают высокой степенью связности, и наша эвристическая цель состоит в том, чтобы целостность была как можно выше. Связность — полезный инструмент управления сложностью, потому что чем лучше код класса соответствует главной цели, тем проще запомнить все, что код выполняет.

Стремление к связности на уровне методов давно считается полезным эвристическим принципом. На уровне классов эвристический принцип связности во многом выражен в более общем эвристическом принципе адекватного определения абстракций, что уже обсуждалось в этой главе и будет еще обсуждаться в главе 6. Абстрагирование полезно и на уровне методов, но в этом случае принципы абстрагирования и связности более равноправны.

### Формируйте иерархии

Иерархия — это многоуровневая структура организации информации, при которой наиболее общая или абстрактная репрезентация концепции соответствует вершине, а более детальные специализированные репрезентации — более низким уровням. При разработке ПО иерархии обнаруживаются, например, в наборах классов и в последовательностях вызовов методов (уровень 4 на рис. 5-2).

Формирование иерархий уже более 2000 лет является важным средством управления сложными наборами информации. Так, Аристотель использовал иерархию для организации царства животных. Люди часто организуют сложную информацию (такую как эта книга) при помощи иерархических схем. Ученые обнаружили, что люди в целом находят иерархии естественным способом организации сложной информации. Рисуя сложный объект (скажем, дом), люди рисуют его иерархически. Сначала они рисуют очертания дома, затем окна и двери, а после этого — еще более подробные детали. Они не рисуют дом по отдельным кирпичам, доскам или гвоздям (Simon, 1996).

Иерархии помогают в достижении Главного Технического Императива Разработки ПО, позволяя сосредоточиться только на том уровне детальности, который

заботит вас в конкретный момент. Иерархия не устраняет детали — она просто выталкивает их на другой уровень, чтобы вы могли думать о них, когда захотите, а не все время.

### Формализуйте контракты классов

На более детальном уровне полезную информацию можно получить, рассматривая интерфейс каждого класса как контракт с остальными частями программы. Обычно контракт имеет форму «Если вы обещаете предоставить данные x, y и z и гарантируете, что они будут иметь характеристики a, b и c, я обязуюсь выполнить операции 1, 2 и 3 с ограничениями 8, 9 и 10». Обещания клиентов классу обычно называются *предусловиями* (preconditions), а обязательства класса перед клиентами — *постусловиями* (postconditions).

**Перекрестная ссылка** О контрактах см. подраздел «Используйте утверждения для документирования и проверки предусловий и постусловий» раздела 8.2.

Контракты помогают управлять сложностью, потому что хотя бы теоретически объект может свободно игнорировать любое поведение, не описанное в контракте. На практике этот вопрос куда сложнее.

### Грамотно назначайте сферы ответственности

Еще один эвристический принцип — обдумывание сфер ответственности, которые следует назначить объектам. Рассмотрение сферы ответственности объекта аналогично вопросу о том, какую информацию он должен скрывать, но мне кажется, что первый способ может привести к более общим ответам, чем и объясняется уникальность этого эвристического принципа.

### Проектируйте систему для тестирования

На некоторые интересные идеи можно натолкнуться, спросив, как будет выглядеть система, если спроектировать ее для обеспечения максимальной легкости тестирования. Отделять ли пользовательский интерфейс от остальной части программы, чтобы протестировать его независимо? Организовывать ли каждую подсистему так, чтобы минимизировать ее зависимость от других подсистем? Проектирование для тестирования часто приводит к разработке более формализованных интерфейсов классов, что обычно выгодно.

### Избегайте неудач

Профессор гражданского строительства Генри Петроски в интересной книге «Design Paradigms: Case Histories of Error and Judgment in Engineering» (Petroski, 1994), посвященной истории неудач в отрасли проектирования мостов, утверждает, что многие известные мосты рушились из-за чрезмерного внимания к прошлым успехам и неадекватного рассмотрения возможных причин аварий. Он делает вывод, что аварий вроде крушения моста Tacoma Narrows можно было бы избежать, если б инженеры тщательно рассматривали возможные причины аварий, а не просто копировали другие успешные проекты.

Крупные бреши в защите многих известных систем, обнаруженные в прошедшие годы, заставляют подумать о том, как применить идеи Петроски в области проектирования ПО.

### Тщательно выбирайте время связывания

**Перекрестная ссылка** О времени связывания см. раздел 10.6.

Временем связывания (binding time) называют тот момент, когда переменной присваивается конкретное значение. Раннее связывание обычно упрощает код, но и снижает его гибкость. Иногда к полезным идеям проектирования можно прийти, спросив себя: «Что, если связать эти значения раньше? Что, если связать их позже? Что, если инициализировать эту таблицу в этом месте кода? Что, если получить значение этой переменной от пользователя в период выполнения программы?»

### Создайте центральные точки управления

Ф. Дж. Плоджер говорит, что главным его принципом является «Принцип Одного Верного Места: в программе должно быть Одно Верное Место для поиска нетривиального фрагмента кода и Одно Верное Место для внесения вероятных изменений» (Plauger, 1993). Управление может быть централизовано в классах, методах, макросах препроцессора, файлах, включаемых директивой *#include*, — даже именованная константа может быть центральной точкой управления.

Этот принцип также способствует снижению сложности: если какой-то программный элемент встречается в минимальном числе фрагментов, его изменение окажется проще и безопаснее.

Если сомневаетесь, используйте грубую силу.

*Батлер Лэмпсон* (*Butler Lampson*)

### Подумайте об использовании грубой силы

Грубая сила — один из мощнейших эвристических инструментов. Не стоит ее недооценивать. Работоспособное решение проблемы методом грубой силы лучше, чем элегантное, но не работающее решение. Создавать элегантные решения зачастую долго и сложно. Так, описывая историю разработки алгоритмов поиска, Дональд Кнут указал, что, хотя первое описание алгоритма двоичного поиска было опубликовано в 1946 г., алгоритм, правильно обрабатывающий списки всех размеров, был разработан только спустя 16 лет (Knuth, 1998). Двоичный поиск элегантнее, но и основанный на грубой силе последовательный поиск часто приемлем.

### Рисуйте диаграммы

Диаграммы — еще один мощный эвристический инструмент. Все знают, что лучше один раз увидеть, чем сто раз услышать. Диаграммы позволяют представить проблему на более высоком уровне абстракции, и никакие описания их не заменят. Помните: иногда проблему следует рассматривать на детальном уровне, а иногда целесообразно иметь дело с более общими аспектами.

### Поддерживайте модульность проекта системы

Этот принцип подразумевает, что каждый метод или класс должен быть похож на «черный ящик»: вы знаете, что в него поступает и что из него выходит, но не знаете, что происходит внутри. Черный ящик имеет такой простой интерфейс и такую ясную функциональность, что для любых конкретных входных данных можно точно предсказать соответствующие выходные данные.

Концепция модульности связана с сокрытием информации, инкапсуляцией и другими эвристическими принципами проектирования. И все же размышление о том, как собрать систему из набора черных ящиков иногда приводит к догадкам, к которым сокрытие информации и инкапсуляция привести не могут, так что принцип модульности заслужил право занять место в вашем арсенале полезных идей.

## Резюме эвристических принципов проектирования

Ниже приведен список основных эвристических принципов проектирования:

- определите объекты реального мира;
- определите согласованные абстракции;
- инкапсулируйте детали реализации;
- используйте наследование, когда это возможно;
- скрывайте секреты (помните про сокрытие информации);
- определите области вероятных изменений;
- поддерживайте сопряжение слабым;
- старайтесь использовать популярные шаблоны проектирования.

> Больше беспокоит то, что программист вполне может выполнить ту же задачу двумя или тремя способами: иногда неосознанно, но довольно часто просто ради изменения или же создания элегантной вариации.
>
> *А. Р. Браун и У. А. Сэмпсон (A. R. Brown and W. A. Sampson)*

Следующие эвристические принципы также иногда бывают полезны:

- стремитесь к максимальной связности;
- формируйте иерархии;
- формализуйте контракты классов;
- грамотно назначайте сферы ответственности;
- проектируйте систему для тестирования;
- избегайте неудач;
- тщательно выбирайте время связывания;
- создайте центральные точки управления;
- подумайте об использовании грубой силы;
- рисуйте диаграммы;
- поддерживайте модульность проекта системы.

## Советы по использованию эвристических принципов

Подходы к проектированию ПО могут быть основаны на подходах, применяемых в других областях. Одной из первых книг, посвященных использованию эвристики при решении проблем, является «How to Solve It (Как решать задачу)» Д. Полья (Polya, 1957). Обобщенный подход Полья к решению проблем концентрируется на решении математических задач. Он резюмирован на рис. 5-10 (шрифт оригинала сохранен).

http://cc2e.com/0592

**1. Понимание постановки задачи.** Нужно ясно *понять* задачу.

*Что неизвестно? Что дано? В чем состоит условие?* Возможно ли удовлетворить условию? Достаточно ли условие для определения неизвестного? Или недостаточно?

Сделайте чертеж. Введите подходящие обозначения. Разделите условие на части. Постарайтесь записать их.

**2. Составление плана решения.** Нужно найти связь между данными и неизвестными. Если не удается сразу обнаружить эту связь, возможно, полезно будет рассмотреть вспомогательные задачи. В конечном счете необходимо прийти к *плану* решения.

Не встречалась ли вам раньше эта задача? Хотя бы в несколько иной форме? *Известна ли вам какая-нибудь родственная задача?* Не знаете ли теоремы, которая могла бы оказаться полезной?

*Рассмотрите неизвестное!* И постарайтесь вспомнить знакомую задачу с тем же или подобным неизвестным. *Вот задача, родственная данной и уже решенная*. Нельзя ли воспользоваться ею? Нельзя ли применить ее результат? Нельзя ли использовать метод ее решения? Не следует ли ввести какой-нибудь вспомогательный элемент, чтобы стало возможно воспользоваться прежней задачей?

Нельзя ли иначе сформулировать задачу? Еще иначе? Вернитесь к определениям.

Если не удается решить данную задачу, попытайтесь сначала решить сходную. Нельзя ли придумать более доступную сходную задачу? Более общую? Более частную? Аналогичную задачу? Нельзя ли решить часть задачи? Сохраните только часть условия, отбросив остальную часть: насколько определенным окажется тогда неизвестное, как оно сможет меняться? Нельзя ли извлечь что-то полезное из данных? Нельзя ли придумать другие данные, из которых можно было бы определить неизвестное? Нельзя ли изменить неизвестное, или данные, или, если необходимо, и то и другое так, чтобы новое неизвестное и новые данные оказались ближе друг к другу?

Все ли данные вами использованы? Все ли условия? Приняты ли вами во внимание все существующие понятия, содержащиеся в задаче?

**3. Осуществление плана.** *Нужно осуществить* план решения.

Осуществляя план решения, *контролируйте каждый свой шаг*. Ясно ли вам, что предпринятый вами шаг правилен? Сумеете ли доказать, что он правилен?

**4. Взгляд назад.** Нужно *изучить* найденное решение. Нельзя ли *проверить результат*? Нельзя ли проверить ход решения? Нельзя ли получить тот же результат иначе? Нельзя ли усмотреть его с одного взгляда? Нельзя ли в какой-нибудь другой задаче использовать полученный результат или метод решения?

***Рис. 5-10.*** *Д. Полья разработал подход к решению математических задач, который полезен и при решении проблем, связанных с проектированием ПО (Polya 1957)*

Одним из самых ценных советов, которые можно дать по поводу проектирования ПО, является использование разных подходов. Если проект, разработанный с помощью UML, неудачен, выразите его на обычном языке. Напишите небольшую тестовую программу. Попробуйте совершенно другой подход. Подумайте об использовании грубой силы. Продолжайте рисовать эскизы и наброски карандашом, и мозг последует за рукой. Если ничего не выходит, отложите решение проблемы.

Прежде чем возвращаться к работе над ней, прогуляйтесь, подумайте о чем-то другом. Довольно часто это приводит к более быстрому получению нужного результата, чем простое упорство.

Никто не заставляет вас разработать весь проект за раз. Если натолкнетесь на препятствие, подумайте, обладаете ли вы информацией, достаточной для решения всех специфических проблем? Зачем через силу разрабатывать оставшиеся 20% проекта, если впоследствии они прекрасно станут на свое место? Зачем принимать неудачные решения, основанные на недостаточной информации, если позже можно будет принять более подходящие решения? Некоторые разработчики чувствуют дискомфорт, если не могут создать полный проект программы к окончанию этапа проектирования, но после создания нескольких удачных программ без заблаговременного решения всех вопросов на этапе проектирования такая ситуация начинает казаться вполне естественной (Zahniser, 1992; Beck, 2000).

# 5.4. Методики проектирования

Предыдущий раздел был посвящен эвристическим принципам, связанным с атрибутами проектов. Иначе говоря, эти принципы характеризуют то, каким должен быть завершенный проект приложения. В этом разделе мы рассмотрим эвристические *методики проектирования* — действия, которые часто приводят к хорошим результатам.

## Используйте итерацию

Возможно, у вас были случаи, когда вы так много узнали во время написания программы, что желали бы написать ее заново, опираясь на полученные знания. Этот же феномен наблюдается и при проектировании, но этап проектирования короче, тогда как влияние, оказываемое им на последующие этапы, выражено сильнее, поэтому вы вполне можете выполнить этап проектирования несколько раз.

Проектирование — итеративный процесс. Выйдя из точки А и достигнув точки Б, не останавливайтесь, а вернитесь в точку А.

Изучая возможные варианты проектирования и пробуя разные подходы, вы будете рассматривать и высокоуровневые, и низкоуровневые аспекты. Общая картина, которую вы получаете при работе над высокоуровневыми вопросами, поможет вам лучше понять низкоуровневые детали. Детали, которые вы узнаете при работе над низкоуровневыми вопросами, помогут вам создать прочный фундамент для принятия высокоуровневых решений. Некоторые конфликты между высокоуровневыми и низкоуровневыми соображениями — вполне здоровое явление; это напряжение способствует созданию структуры, более стабильной, чем структура, полностью созданная «сверху вниз» или «снизу вверх».

Многим программистам — и вообще многим людям — трудно переключаться между высокоуровневыми и низкоуровневыми точками зрения, но эта способность — важное условие эффективного проектирования. Занимательные упражнения, позволяющие развить гибкость ума, можно найти в книге «Conceptual Blockbusting» (Adams, 2001), описанной в разделе «Дополнительные ресурсы» в конце главы.

**Перекрестная ссылка** Безопасный способ попробовать разные варианты кода предоставляет рефакторинг (глава 24).

Если первая попытка создания проекта кажется вполне удачной, не останавливайтесь! Вторая попытка почти всегда оказывается лучше первой, и при каждой попытке вы будете узнавать что-то такое, что поможет вам улучшить общий проект. Говорят, что, когда Томаса Эдисона, который пытался создать нить лампочки и испробовал на тот момент уже тысячу разных материалов, спросили, не жалеет ли он о том, что зря потратил время, так ничего и не обнаружив, Эдисон ответил: «Ни в коей мере. Я обнаружил тысячу вариантов, которые не работают». Во многих случаях, решив проблему при помощи одного подхода, вы получите знания, которые позволят решить проблему иным, более эффективным способом.

## Разделяй и властвуй

Как указал Эдсгер Дейкстра, никто не обладает умом, способным вместить все детали сложной программы. То же можно сказать и о проектировании. Разделите программу на разные области и спроектируйте их по отдельности. Если, работая над одной из областей, вы попадете в тупик, вспомните про итерацию!

Инкрементное улучшение — мощное средство управления сложностью. Вспомните, как Полья советовал решать математические задачи: поймите задачу, составьте план решения, осуществите план и *оглянитесь назад*, чтобы лучше понять, что и как вы сделали (Polya, 1957).

## Нисходящий и восходящий подходы к проектированию

Слова «нисходящий» и «восходящий» могут казаться устаревшими, но они предоставляют много ценной информации об объектно-ориентированных способах проектирования. Нисходящее (top-down) проектирование начинается на высоком уровне абстракции. Например, вы сначала определяете базовые классы или другие неспецифические элементы проекта. По ходу работы вы повышаете уровень детальности и определяете производные классы, сотрудничающие классы и другие детали.

Восходящее (bottom-up) проектирование начинается со специфики и постепенно переходит ко все большей общности. Как правило, оно начинается с определения конкретных объектов, на основе которых затем разрабатываются более общие объединения объектов и базовые классы.

Некоторые разработчики утверждают, что лучше всего начинать с общего и двигаться по направлению к частному, а другие — что общие принципы проектирования нельзя определить, не обдумав важных деталей. Аргументы обеих сторон описаны ниже.

### Аргументы в пользу нисходящего проектирования

Нисходящее проектирование основано на том факте, что человеческий мозг в каждый конкретный момент времени может работать только с определенным объемом информации. Если вы начинаете проектирование с общих классов, выполняя их декомпозицию на более специализированные классы шаг за шагом, мозгу не приходится иметь дело со слишком большим числом деталей сразу.

Процесс «разделяй и властвуй» итеративен в двух аспектах. Во-первых, потому, что, выполнив один этап декомпозиции, разработчики обычно не останавливаются, а выполняют еще несколько этапов. Во-вторых, потому, что дело обычно не ограничивается одной попыткой декомпозиции. Вы выполняете декомпозицию одним способом. На разных этапах декомпозиции у вас будут разные варианты разделения подсистем, формирования дерева наследования и группирования объектов. Вы делаете выбор и смотрите, что происходит. Затем вы возвращаетесь, выполняете декомпозицию иначе и смотрите, что работает лучше. После нескольких попыток вы получите хорошее представление о том, чту будет работать и почему.

До каких пор продолжать декомпозицию программы? До тех, пока вам не покажется, что вместо декомпозиции следующего уровня его было бы проще закодировать. До тех, пока очевидность и простота проекта не станут вас в каком-то смысле раздражать. В этот момент все готово. Если что-то неясно, продолжите декомпозицию. Если сейчас решение кажется вам хоть чуть-чуть хитрым, для любого, кто будет работать над ним позднее, оно станет головоломкой.

### Аргументы в пользу восходящего проектирования

Иногда нисходящий подход настолько абстрактен, что его трудно начать. Если вам нужно работать с чем-то более реальным, попробуйте восходящий подход к проектированию. Спросите себя: «Какие функции эта система должна выполнять?» Несомненно, вы сможете ответить на этот вопрос. Вы можете определить несколько низкоуровневых аспектов ответственности, которые можно назначить конкретным классам. Так, вы можете знать, что система должна форматировать конкретный отчет, вычислять данные для отчета, центрировать его заголовки, отображать на экране, печатать на принтере и т. д. Определив несколько низкоуровневых аспектов ответственности, вы скорее всего почувствуете себя достаточно подготовленным, чтобы еще раз взглянуть на вершину.

В других случаях основные атрибуты проекта могут быть продиктованы низкоуровневыми факторами, такими как особенности взаимодействия с оборудованием.

Вот некоторые рекомендации, о которых следует помнить при выполнении восходящей композиции:

- спросите себя, какие функции должна выполнять система;
- опираясь на этот вопрос, определите конкретные объекты и их сферы ответственности;
- определите общие объекты и сгруппируйте их, организовав в подсистемы или пакеты, с помощью композиции или наследования (выберите самый подходящий вариант);
- поднимитесь на следующий уровень или вернитесь на вершину и попробуйте еще раз начать нисходящее проектирование.

### Никакого конфликта нет

Главное различие между нисходящей и восходящей стратегиями в том, что одна является стратегией декомпозиции, а вторая — композиции. В первом случае вы начинаете работу с общей проблемы, разбивая ее на управляемые фрагменты, во втором вы начинаете с управляемых фрагментов, составляя из них общее реше-

ние. Оба подхода имеют достоинства и недостатки, которые следует рассмотреть в контексте конкретной проблемы.

Сила нисходящего подхода — в простоте. Люди (особенно программисты) прекрасно умеют делить что-то крупное на меньшие компоненты.

Еще одно достоинство в том, что нисходящее проектирование позволяет отложить работу над деталями конструирования. Изменения аспектов конструирования (таких как структура файлов или формат отчетов) часто сказываются на всей системе, поэтому лучше заранее знать, что эти детали следует скрыть в классах на нижних уровнях иерархии.

Одно достоинство восходящего подхода объясняется тем, что он обычно приводит к раннему определению вспомогательной функциональности, что способствует созданию компактного, хорошо факторизованного проекта системы. Если похожая система уже создавалась, восходящий подход позволяет начать проектирование новой системы с рассмотрения и повторного использования фрагментов старой системы.

Восходящая композиция имеет и недостатки: во-первых, ее трудно использовать без применения других подходов. Большинство людей находят разбиение крупной концепции на меньшие части более легким, чем объединение небольших концепций в более крупную. Это напоминает старую проблему, связанную с конструкторами: модель кажется готовой, но почему в коробке остались детали? К счастью, никто не заставляет проектировать программы, применяя исключительно восходящий подход.

Во-вторых, при восходящем проектировании иногда оказывается, что из исходных фрагментов создать программу невозможно. Никто не сможет собрать самолет из кирпичей. Чтобы узнать, какие фрагменты понадобятся на нижних уровнях, иногда нужно сначала получить общее представление о системе.

Подведем итог: нисходящее проектирование обычно начинается с простого, но иногда низкоуровневые сложности прорываются на вершину, и это может приводить к усложнению системы, которого можно было избежать. Восходящее проектирование начинается со сложных аспектов, но определение этой сложности на ранних этапах позволяет лучше спроектировать высокоуровневые классы.... если к этому моменту сложность не потопит всю систему!

В конечном счете это не конкурирующие стратегии — они дополняют друг друга. Проектирование — эвристический процесс, а значит, универсальных решений не существует. Проектирование содержит элементы метода проб и ошибок. Пробуйте разные подходы, пока не найдете тот, что вас устроит.

## Экспериментальное прототипирование

http://cc2e.com/0599

Иногда адекватность конкретного проекта невозможно оценить, не имея дополнительных сведений о деталях реализации. Вы можете не знать, приемлема ли конкретная организация базы данных, пока не узнаете, будет ли она удовлетворять конкретным требованиям к производительности. Вы можете не знать, приемлем ли проект

конкретной подсистемы, пока не будут выбраны конкретные библиотеки GUI. Это примеры существенной «грязи» при проектировании ПО: вы не можете полностью определить проблему проектирования, пока не решите ее хоть частично.

Хорошо известен недорогой способ получить ответы на эти вопросы — экспериментальное прототипирование. В слово «прототипирование» люди вкладывают разный смысл (McConnell, 1996). В данном контексте оно означает написание абсолютно минимального объема подлежащего выбрасыванию кода, нужного для ответа на отдельный вопрос проектирования.

Если разработчики недисциплинированно относятся к написанию *абсолютно минимального* объема кода, нужного для ответа на вопрос, прототипирование работает плохо. Допустим, вопрос проектирования таков: «Может ли выбранная нами организация базы данных поддерживать нужный объем транзакций?» Для ответа не нужно писать полноценный код, который можно было бы использовать в готовой системе. Вы можете даже не знать специфику базы данных. Вам лишь нужна информация, достаточная для аппроксимации проблемной области: число таблиц, число элементов в таблицах и т. д. Далее вы можете написать простой прототипный код, использующий таблицы и столбцы с именами вроде *Table1*, *Table2* и *Column1*, *Column2*, заполнить таблицы фиктивными данными и протестировать производительность.

Прототипирование также работает плохо, если задача недостаточно *конкретна*. Вопрос «Будет ли эта организация базы данных работать?» недостаточно хорошо определяет направление прототипирования. В то же время вопрос «Будет ли эта организация базы данных поддерживать 1000 транзакций в секунду при условиях X, Y и Z?» предоставляет более прочную основу для прототипирования.

Наконец, еще один фактор риска возникает, если разработчики не рассматривают код как *подлежащий выбрасыванию*. Я обнаружил, что люди не могут написать абсолютно минимальный объем кода, нужный для ответа на вопрос, если думают, что код в конечном счете войдет в итоговую версию системы. Из-за этого они вместо прототипирования занимаются реализацией системы. Настроившись на то, что, как только ответ на вопрос будет получен, код будет выброшен, вы сведете этот риск к минимуму. Избежать этой проблемы можно, если создавать прототипы и основную программу, используя разные технологии. Вы можете создать прототип проекта Java на языке Python или смоделировать пользовательский интерфейс в Microsoft PowerPoint. Если вы все-таки создаете прототипы, используя ту же технологию, пусть имена прототипичных классов и пакетов начинаются с префикса *prototype*. Это хотя бы заставит программиста дважды подумать, прежде чем он решит расширять прототипный код (Stephens, 2003).

При дисциплинированном применении прототипирование — эффективный способ борьбы с «грязнотой» проектирования. В противном случае оно делает проектирование еще более грязным.

## Совместное проектирование

**Перекрестная ссылка** О совместной разработке ПО см. главу 21

Обычно при проектировании две головы лучше, чем одна, организованы они формально или неформально. Сотрудничество может принимать любую из следующих форм:

- вы подходите к столу коллеги и просите его обсудить с вами некоторые идеи;
- вы с коллегой идете в конференц-зал и рисуете на доске варианты проекта;
- вы с коллегой садитесь перед клавиатурой и выполняете детальное проектирование на выбранном языке программирования, т. е. вы можете использовать парное программирование (см. главу 21);
- вы назначаете собрание для обсуждения своих идей с одним или несколькими коллегами;
- вы назначаете формальную инспекцию, включающую все аспекты, описанные в главе 21;
- никто не может провести обзор вашей работы, поэтому вы выполняете некоторый объем работы, сохраняете ее и возвращаетесь к ней через неделю — вы забудете достаточно, чтобы самостоятельно провести довольно хороший обзор своей же работы;
- вы обращаетесь за помощью к людям, не работающим в вашей компании: отправляете вопросы в специализированный форум или группу новостей.

Если целью является гарантия качества, тогда по причинам, описанным в главе 21, я рекомендую наиболее структурированную методику обзора — формальные инспекции. Но если цель состоит в содействии творчеству и увеличении числа предлагаемых вариантов проекта, а не в простом нахождении ошибок, лучше применять менее структурированные подходы. После выбора определенного варианта проекта может оказаться уместным переход к более формальным инспекциям, что определяется конкретной ситуацией.

## Какую степень проектирования считать достаточной?

Мы пытаемся решить проблему, максимально ускоряя процесс проектирования, чтобы в конце работы над системой у нас осталось достаточно времени для нахождения ошибок, допущенных из-за слишком быстрого проектирования.

*Гленфорд Майерс*
(*Glenford Myers*)

Иногда перед началом кодирования создается только самый общий набросок архитектуры. В других случаях группы создают проекты с таким уровнем подробностей, что кодирование становится практически механическим занятием. Насколько детально выполнять проектирование до начала кодирования?

Родственный вопрос заключается в том, насколько формальным делать проект. Нужны ли вам формальные, тщательно выполненные диаграммы проекта системы или цифровых снимков нескольких рисунков на доске будет достаточно?

Принятие решения о том, какую часть проектирования выполнять до начала полномасштабного кодирования и насколько формально документировать проект системы, трудно назвать точной наукой. При этом следует учитывать опыт группы, ожидаемый срок службы системы, желательный уровень надежности, масштаб проекта и число членов группы. Влияние этих факторов на подход к проектированию отражено в табл. 5-2.

**Табл. 5-2. Необходимые уровни формальности и детальности проекта**

| Фактор | Уровень детальности проекта, нужный перед началом конструирования | Уровень формальности документации |
|---|---|---|
| Члены группы проектирования/конструирования имеют большой опыт работы в прикладной области. | Низкий | Низкий |
| Члены группы проектирования/конструирования имеют большой опыт, но плохо знакомы с прикладной областью. | Средний | Средний |
| Члены группы проектирования/конструирования неопытны. | Средний — высокий | Низкий — средний |
| Для группы проектирования/конструирования характерен средний — высокий уровень текучести. | Средний | — |
| От приложения будет зависеть безопасность людей. | Высокий | Высокий |
| Приложение предназначено для решения ответственных задач. | Средний | Средний — высокий |
| Проект небольшой. | Низкий | Низкий |
| Проект крупный. | Средний | Средний |
| Предполагается, что ПО будет использоваться недолго (недели или месяцы). | Низкий | Низкий |
| Предполагается, что ПО будет использоваться длительное время (месяцы или годы). | Средний | Средний |

При создании конкретной системы могут иметь место сразу несколько факторов, подталкивающих к разным решениям: скажем, опытная группа может разрабатывать ПО, от которого будет зависеть безопасность людей. Тогда вам, вероятно, следует отдать предпочтение более высокой детальности и формальности проекта. Вообще в подобной ситуации нужно оценить роль каждого фактора и решить, какой из них важнее.

Если разработчики выполняют какую-то часть проектирования индивидуально, то при снижении проектирования до уровня задачи, которая уже была решена ранее, или до уровня простого изменения или расширения такой задачи, проектирование, вероятно, можно прекратить и начать кодирование.

Если я не могу решить, насколько детальным должен быть проект программы перед началом кодирования, я предпочитаю разрабатывать его более детально. Самые крупные ошибки проектирования возникали, когда я думал, что выполнил проектирование в достаточной степени, но позднее оказывалось, что я заблуждался и не учитывал дополнительные проблемы. Иначе говоря, самые серьезные проблемы проектирования обычно были связаны не с теми областями, которые я считал сложными и спроектировал неудачно, а с теми, которые я считал легкими и не спроектировал вообще. Мне редко встречаются проекты, страдающие от чрезмерного проектирования.

Я никогда не встречал человека, желающего читать 17 000 страниц документации, а если бы встретил, то убил бы его, чтобы он не портил генофонд.

*Джозеф Костелло (Joseph Costello)*

С другой стороны, иногда я имею дело с проектами, страдающими от слишком объемной проектной *документации*. Здесь вступает в игру своеобразный закон Грешема[1], и «механические действия начинают вытеснять творчество» (Simon, 1965). Чрезмерное внимание к созданию проектной документации — хорошее подтверждение этого закона. Я бы предпочел, чтобы разработчики тратили 80% усилий на разработку и анализ различных вариантов проектирования и 20% — на создание менее изысканной документации, а не наоборот: 20% — на создание посредственных решений и 80% — на совершенствование документации не совсем удачных проектов.

## Регистрация процесса проектирования

http://cc2e.com/0506

Традиционным способом регистрации проекта является его описание в формальной проектной документации. Однако есть масса других способов, эффективных при использовании неформального подхода, при создании небольших систем или когда нужна «облегченная» методика регистрации проекта:

Плохие новости заключаются в том, что мы, как нам кажется, никогда не найдем философского камня. Мы никогда не найдем процесса, позволяющий проектировать ПО абсолютно рациональным образом. Но есть и хорошая новость: мы можем его подделать.

*Дэвид Парнас и Пол Клементс (David Parnas and Paul Clements)*

***Включайте проектную документацию прямо в код*** Документируйте основные аспекты проектирования в комментариях — как правило, в заголовках файлов или классов. Дополнив этот подход использованием утилиты извлечения документации (такой как JavaDoc), вы позволите программистам, работающим над конкретными фрагментами кода, с легкостью обращаться к соответствующей проектной документации, и повысите вероятность того, что они будут поддерживать ее актуальность.

***Регистрируйте протоколы обсуждения проекта и принятые решения при помощи Wiki*** Сохраняйте письменные протоколы обсуждения проекта в системе Wiki (набор Web-страниц, которые может редактировать каждый член группы при помощи Web-браузера). Это позволяет автоматизировать регистрацию нужных данных, хотя и требует дополнительных затрат на набор текста. Кроме того, в Wiki можно хранить полезные цифровые фотографии, ссылки на Web-сайты, содержащие обоснования принятых решений, документы и другие материалы. Этот метод особенно полезен, если члены группы отдалены друг от друга.

***Пишите резюме дискуссий в форме электронной почты*** Обсудив проект системы, поручите кому-нибудь записать резюме беседы — особенно принятые решения — и отправить его каждому члену группы. Сохраняйте копии писем в папке, доступной всем участникам проекта.

---

[1] Закон Грешема (Gresham's Law) — монетарный принцип, названный в честь лорда Томаса Грешема, управляющего английским монетным двором при королеве Елизавете. Суть закона Грешема в том, что «худшие» деньги (некачественные или с пониженным содержанием благородного металла) вытесняют «лучшие» деньги из обращения, т. е. более «дешевые» деньги вытесняют более «дорогие». — *Прим. перев.*

***Используйте цифровой фотоаппарат*** Одним частым барьером, препятствующим документированию проекта, является утомительность рисования проектов традиционным способом. Однако способы документирования не ограничиваются вариантами «регистрации проекта с использованием красиво отформатированной формальной нотации» и «полного отсутствия проектной документации».

Фотографируйте диаграммы, которые разработчики рисуют на доске, и включайте их в традиционные документы: это гораздо проще, чем рисовать схемы вручную, но не менее эффективно.

***Храните плакаты со схемами проекта*** Нет закона, утверждающего, что для документирования проекта нужно использовать стандартные листы бумаги почтового размера. Если вы рисуете диаграммы проектов на больших листах, можете просто сохранить их в удобном месте или, что еще лучше, развесить на стенах, чтобы разработчики могли обращаться к ним и обновлять по мере надобности.

***Используйте карточки CRC (Class, Responsibility, Collaborator — класс, ответственность, сотрудничество)*** Еще один простой вариант документирования проекта — использовать карточки. Напишите на каждой карточке имя класса, аспекты его ответственности и имена классов, с которыми он сотрудничает. Продолжайте работать с карточками, пока не будете удовлетворены результатом. В этот момент вы можете просто сохранить карточки на будущее. Этот способ почти не требует расходов, не пугает своей сложностью и поощряет взаимодействие членов группы (Beck, 1991).

http://cc2e.com/0513

***Создавайте диаграммы UML с уместным уровнем детальности*** Одним из популярных способов создания диаграмм проектов является язык UML (Unified Modeling Language; унифицированный язык моделирования), стандартизацией которого занимается организация Object Management Group (Fowler, 2004). Пример UML-диаграммы классов вы уже видели на рис. 5-6. UML предоставляет богатый набор формализованных репрезентаций для проектирования сущностей и их отношений. Вы можете использовать неформальные версии UML для анализа и обсуждения подходов к проектированию. Начните с минимальных набросков и добавляйте детали, только выбрав конкретный вариант проекта. Так как UML стандартизирован, он позволяет эффективнее обмениваться идеями и может ускорить процесс рассмотрения вариантов проектов при работе в группе.

Описанные способы работают и в различных комбинациях, так что можете свободно смешивать их, приспосабливая к конкретным проектам и даже разным областям одного проекта.

## 5.5. Комментарии по поводу популярных методологий

История проектирования ПО отмечена бурными спорами фанатичных сторонников конфликтующих подходов к проектированию. Когда в начале 1990-х вышло в свет первое издание этой книги, фанатики проектирования утверждали, что перед началом кодирования нужно расставить все точки над «i» на этапе проектирования. Эта рекомендация не имела никакого смысла.

Люди, описывающие проектирование ПО как дисциплинированный процесс, тратят много энергии, заставляя всех нас почувствовать себя виноватыми. Мы никогда не станем достаточно структурированными или объектно-ориентированными для достижения нирваны при жизни. Все мы расплачиваемся за первородный грех — изучение Basic в особо впечатлительном возрасте. Но я готов спорить, что большинство из нас проектируют программы лучше, чем кажется пуристам.

*Ф. Дж. Плоджер*
(*P. J. Plauger*)

Сейчас, в середине первого десятилетия XXI века, некоторые «проповедники» утверждают, что проектирование вообще не требуется. «Крупномасштабное Предварительное Проектирование — это плохо, — говорят они. — Лучше вообще не выполнять проектирование перед началом кодирования!»

За десять лет маятник сместился от «проектирования всего» до «проектирования ничего». Но альтернативой Крупномасштабному Предварительному Проектированию является не отсутствие предварительного проектирования, а Небольшое Предварительное Проектирование или Достаточное Предварительное Проектирование.

Какой объем проектирования достаточен? Никто не может точно ответить на этот вопрос. Тем не менее можно с полной уверенностью утверждать, что два варианта обязательно окажутся неудачными: проектирование всех деталей до единой и полное отсутствие проектирования. Крайние точки зрения всегда ошибочны!

Как сказал Ф. Дж. Плоджер, «чем догматичнее вы будете в отношении методики проектирования, тем меньше реальных проблем решите» (Plauger, 1993). Рассматривайте проектирование как грязный, неряшливый эвристический процесс. Не останавливайтесь на первом проекте, который пришел вам в голову. Сотрудничайте. Стремитесь к простоте. Создавайте прототипы, если нужно. Не забывайте про итерацию. Вы будете довольны своими проектами.

# Дополнительные ресурсы

http://cc2e.com/0520

Проектирование ПО описанно во множестве работ. Проблема в том, чтобы определить, какие из них наиболее полезны. Позволю себе дать вам некоторые советы.

## Общие вопросы проектирования ПО

Weisfeld, Matt. *The Object-Oriented Thought Process*, 2d ed. — SAMS, 2004 — понятное введение в объектно-ориентированное программирование. Если вы уже знакомы с объектно-ориентированным программированием, возможно, вам следует поискать более серьезную книгу, но если вы новичок в этой области, эта книга познакомит вас с фундаментальными объектно-ориентированными концепциями, такими как объекты, классы, интерфейсы, наследование, полиморфизм, перегрузка, абстрактные классы, агрегация и ассоциация, конструкторы/деструкторы, исключения и т. д.

Riel, Arthur J. *Object-Oriented Design Heuristics*. — Reading, MA: Addison-Wesley, 1996. В этой книге основное внимание уделяется проектированию на уровне классов. Еще она легко читается.

Plauger, P. J. *Programming on Purpose: Essays on Software Design*. — Englewood Cliffs, NJ: PTR Prentice Hall, 1993. В этой книге я нашел столько хороших советов по проектированию ПО, сколько во всех остальных прочитанных мной книгах вместе

взятых. Плоджер прекрасно разбирается во многих подходах к проектированию, он прагматичен, и он отличный писатель.

Meyer, Bertrand. *Object-Oriented Software Construction*, 2d ed. — New York, NY: Prentice Hall PTR, 1997. Мейер приводит убедительные доводы в защиту чистого объектно-ориентированного программирования.

Raymond, Eric S. *The Art of UNIX Programming*. — Boston, MA: Addison-Wesley, 2004. Эта книга — хорошо обоснованный взгляд на проектирование ПО сквозь призму UNIX. В разделе 1.6 приведено лаконичное объяснение 17 ключевых принципов проектирования ПО для UNIX.

Larman, Craig. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2d ed. — Englewood Cliffs, NJ: Prentice Hall, 2001. Это популярное введение в объектно-ориентированное проектирование в контексте метода Unified Process. Кроме того, здесь обсуждается объектно-ориентированный анализ.

## Теория проектирования ПО

Parnas, David L., and Paul C. Clements. «A Rational Design Process: How and Why to Fake It». — *IEEE Transactions on Software Engineering SE-12*, no. 2 (February 1986): 251–57. В этой классической статье описывается разрыв между реальным и желательным процессами проектирования программ. Суть статьи в том, что в действительности никто никогда не следует рациональному упорядоченному процессу проектирования, но стремление к этому приводит в итоге к созданию лучших проектов.

Работы, в которых было бы приведено исчерпывающее обсуждение сокрытия информации, мне неизвестны. В большинстве учебников по разработке ПО оно обсуждается кратко, часто в контексте объектно-ориентированных подходов. Наверное, до сих пор лучшими материалами по сокрытию информации являются три статьи, принадлежащие перу Парнаса.

Parnas, David L. «On the Criteria to Be Used in Decomposing Systems into Modules». — *Communications of the ACM 5*, no. 12 (December 1972): 1053-58.

Parnas, David L. «Designing Software for Ease of Extension and Contraction». — *IEEE Transactions on Software Engineering SE-5*, no. 2 (March 1979): 128-38.

Parnas, David L., Paul C. Clements, and D. M. Weiss. «The Modular Structure of Complex Systems». — *IEEE Transactions on Software Engineering SE-11*, no. 3 (March 1985): 259-66.

## Шаблоны проектирования

Gamma, Erich, et al. *Design Patterns*. — Reading, MA: Addison-Wesley, 1995. Очень полезная книга о шаблонах проектирования, написанная «Бандой четырех»[1].

Shalloway, Alan, and James R. Trott. *Design Patterns Explained*. — Boston, MA: Addison-Wesley, 2002. Данная книга представляет собой несложное введение в шаблоны проектирования.

---

[1] «Бандой четырех (Gang of Four)» называют группу авторов, в которую входят Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес. — *Прим. перев.*

## Проектирование в общем

Adams, James L. *Conceptual Blockbusting: A Guide to Better Ideas*, 4th ed. — Cambridge, MA: Perseus Publishing, 2001. Нельзя сказать, что эта книга посвящена непосредственно проектированию ПО, но это не умаляет ее достоинств: она была написана как учебник по проектированию для студентов инженерного факультета Стэнфордского университета. Даже если вы никогда ничего не проектировали и не проектируете, в ней вы найдете увлекательное обсуждение творческого мышления и много упражнений, позволяющих развить мышление, для эффективного проектирования. Кроме того, данная книга включает список литературы, посвященной проектированию и творческому мышлению, с подробными аннотациями. Если вам нравится решать проблемы, вам понравится эта книга.

Polya, G. *How to Solve It: A New Aspect of Mathematical Method*, 2d ed. — Princeton, NJ: Princeton University Press, 1957. Это обсуждение эвристики и решения проблем концентрируется на математике, но актуально и для разработки ПО. Книга Полья стала первым трудом, посвященным применению эвристики для решения математических проблем. В ней проводится четкое различие между небрежной эвристикой, используемой для обнаружения решений, и более аккуратными методами, которые применяются для представления найденных решений. Читать ее нелегко, но если вы интересуетесь эвристикой, то в итоге все равно прочитаете ее, хотите вы того или нет. Полья ясно показывает, что решение проблем не является детерминированным процессом и что приверженность единственной методологии аналогично ходьбе в кандалах. Когда-то в Microsoft эту книгу выдавали всем новым программистам.

Michalewicz, Zbigniew, and David B. Fogel. *How to Solve It: Modern Heuristics*. — Berlin: Springer-Verlag, 2000. Это обновленный вариант книги Полья, который содержит некоторые нематематические примеры и менее требователен к читателю.

Simon, Herbert. *The Sciences of the Artificial*, 3d ed. — Cambridge, MA: MIT Press, 1996. В этой интересной книге проводится различие между науками, имеющими дело с естественным миром (биология, геология и т. д.), и науками, изучающими искусственный мир, созданный людьми (бизнес, архитектура и информатика). Затем в ней обсуждаются характеристики наук об искусственном, при этом особое внимание уделяется проектированию. Книга написана в академическом стиле, и ее следует прочитать всем, кто решил сделать карьеру в области разработки ПО или любой другой «искусственной» области.

Glass, Robert L. *Software Creativity*. — Englewood Cliffs, NJ: Prentice Hall PTR, 1995. Что в большей степени управляет процессом разработки ПО: теория или практика? Является ли он преимущественно творческим или преимущественно детерминированным? Какие интеллектуальные качества нужны разработчику ПО? В этой книге приведено интересное обсуждение природы разработки ПО со специфическим акцентом на проектировании.

Petroski, Henry. *Design Paradigms: Case Histories of Error and Judgment in Engineering*. — Cambridge: Cambridge University Press, 1994. Главная идея этой книги в том, что анализ прошлых неудач способствует успешному проектированию не в меньшей, а то и в большей степени, чем исследование прошлых успехов. В подтверждение своей позиции автор приводит многие факты из области гражданского строительства (особенно проектирования мостов).

## Стандарты

*IEEE Std 1016-1998, Recommended Practice for Software Design Descriptions.* Данный документ содержит стандарт IEEE-ANSI описания проектов ПО, определяющий, что следует включать в проектную документацию.

*IEEE Std 1471-2000. Recommended Practice for Architectural Description of Software Intensive Systems.* Los Alamitos, CA: IEEE Computer Society Press. Этот документ представляет собой руководство IEEE-ANSI по созданию спецификаций архитектуры ПО.

### Контрольный список: проектирование при конструировании

http://cc2e.com/0527

**Методики проектирования**

- ❑ Выполнили ли вы несколько итераций проектирования, выбрав самую лучшую попытку, а не просто первую?
- ❑ Попробовали ли вы выполнить декомпозицию системы несколькими способами с целью нахождения наилучшего варианта?
- ❑ Использовали ли вы для решения проблемы и нисходящий, и восходящий способы проектирования?
- ❑ Выполнили ли вы прототипирование сомнительных или плохо известных частей системы, создав абсолютный минимум подлежащего выбрасыванию кода, нужного для ответа на отдельные вопросы?
- ❑ Был ли выполнен формальный или неформальный обзор вашего проекта другими разработчиками?
- ❑ Довели ли вы проектирование до той точки, в которой реализация проекта кажется очевидной?
- ❑ Выполнили ли вы регистрацию проекта уместными способами, такими как Wiki, электронная почта, плакаты, цифровые фотографии, UML, карточки CRC или комментарии в самом коде?

**Цели проектирования**

- ❑ Адекватно ли проект решает проблемы, которые были определены и отложены на этапе разработки архитектуры?
- ❑ Разделен ли проект на уровни?
- ❑ Удовлетворены ли вы тем, как выполнена декомпозиция программы на подсистемы, пакеты и классы?
- ❑ Удовлетворены ли вы тем, как выполнена декомпозиция классов на методы?
- ❑ Достигнуто ли минимальное взаимодействие классов между собой?
- ❑ Спроектированы ли классы и подсистемы так, чтобы их можно было использовать в других системах?
- ❑ Будет ли программа легкой в сопровождении?
- ❑ Является ли проект полным, но минимальным? Все ли его части действительно необходимы?
- ❑ Подразумевает ли проект использование только стандартных методик? Смогли ли вы избежать применения экзотических, трудных для понимания элементов?
- ❑ Помогает ли проект в целом минимизировать и несущественную, и существенную сложность?

## Ключевые моменты

- Главным Техническим Императивом Разработки ПО является управление сложностью. Управлять сложностью будет гораздо легче, если при проектировании вы будете стремиться к простоте.
- Есть два общих способа достижения простоты: минимизация объема существенной сложности, с которой приходится иметь дело в любой конкретный момент времени, и подавление необязательного роста несущественной сложности.
- Проектирование — эвристический процесс. Слепое следование какой-либо единственной методологии подавляет творческое мышление и снижает качество ваших программ.
- Оптимальный процесс проектирования итеративен; чем больше вариантов проектирования вы попробуете, тем удачнее будет ваш окончательный проект.
- Одной из самых полезных концепций проектирования является сокрытие информации. Вопрос «Что мне скрыть?» устраняет много сложных проблем проектирования.
- Много полезной и интересной информации о проектировании можно найти в других книгах. Описанные в этой главе идеи — лишь вершина айсберга.

ГЛАВА 6

# Классы

### Содержание

http://cc2e.com/0665

- 6.1. Основы классов: абстрактные типы данных
- 6.2. Качественные интерфейсы классов
- 6.3. Вопросы проектирования и реализации
- 6.4. Разумные причины создания классов
- 6.5. Аспекты, специфические для языков
- 6.6. Следующий уровень: пакеты классов

### Связанные темы

- Проектирование при конструировании: глава 5
- Архитектура ПО: раздел 3.5
- Высококачественные методы: глава 7
- Процесс программирования с псевдокодом: глава 9
- Рефакторинг: глава 24

На заре компьютерной эпохи программисты думали о программировании в терминах операторов. В 1970–80-е о программах стали думать в терминах методов. В XXI веке мы рассматриваем программирование в терминах классов.

Класс — это набор данных и методов, имеющих общую, целостную, хорошо определенную сферу ответственности. Данные — необязательный компонент класса: класс может включать только методы, предоставляющие целостный набор услуг. Одним из главных условий эффективного программирования является максимизация части программы, которую можно игнорировать при работе над конкретными фрагментами кода. Классы — главное средство достижения этой цели.

Эта глава содержит экстракт советов по созданию высококачественных классов. Если вы только знакомитесь с концепциями объектно-ориентированного программирования, она может показаться слишком сложной. Если вы не прочитали главу 5, вернитесь к ней. Затем начните с раздела 6.1, который поможет понять остальные разделы главы. Если вы уже знакомы с основами классов, можете, просмотрев раздел 6.1, начать серьезное чтение с раздела 6.2, в котором обсуждаются

интерфейсы классов. В разделе «Дополнительные ресурсы» в конце главы вы найдете список вводных материалов по данной теме, книг более высокого уровня и ресурсов, специфических для языков программирования.

# 6.1. Основы классов: абстрактные типы данных

Абстрактный тип данных (АТД) — это набор, включающий данные и выполняемые над ними операции. Операции описывают данные для остальной части программы и позволяют их изменять. Слово «данные» используется в выражении «абстрактный тип данных» довольно условно. АТД может быть графическое окно со всеми влияющими на него операциями, файл с файловыми операциями, таблица страховых тарифов с соответствующими операциями и др.

**Перекрестная ссылка** Размышление в первую очередь об АТД и только во вторую о классах является примером программирования с использованием языка в отличие от программирования на языке (см. разделы 4.3 и 34.4).

Понимание концепции АТД необходимо для понимания объектно-ориентированного программирования. Не имея ясного представления об АТД, программисты создают классы, которые только называются «классами», будучи на самом деле лишь удобными контейнерами, содержащими наборы плохо согласующихся друг с другом данных и методов. Понимание АТД облегчает создание классов и их изменение с течением времени.

В книгах по программированию обсуждение АТД традиционно носит математический характер. Довольно часто можно встретить высказывания вроде: «АТД можно понимать как математическую модель с определенным для нее набором операций». И создается впечатление, что АТД подойдет разве что в качестве снотворного.

Такие сухие объяснения АТД никуда не годятся. АТД удивительны тем, что позволяют работать с сущностями реального мира, а не с низкоуровневыми сущностями реализации. Благодаря этому вместо вставки узла в связный список можно добавить ячейку в электронную таблицу, новый тип окна в список типов окон или очередной пассажирский автомобиль в программу, моделирующую поток движения. Возможность работать в проблемной области, а не в низкоуровневой области реализации программы очень удобна. Используйте ее!

## Пример необходимости АТД

Для начала приведем пример ситуации, в которой применение АТД было бы полезным. После этого мы сможем углубиться в подробности.

Допустим, вы пишете программу, управляющую выводом текста на экран с использованием разнообразных гарнитур шрифтов, их размеров и атрибутов (например, «полужирный» и «курсив»). За работу со шрифтами отвечает конкретная часть программы. При использовании АТД данные — названия гарнитур, размеры и атрибуты шрифтов — будут объединены в одну группу с обрабатывающими их методами. Набор данных и методов, служащих одной цели, — это и есть АТД.

Без АТД вам пришлось бы принять специализированный подход к работе со шрифтами. Скажем, для выбора шрифта размером 12 пт, которым могли бы соответствовать 16 пикселов, вы написали бы что-то вроде:

```
currentFont.size = 16
```

Создав набор библиотечных методов, код можно было бы сделать чуть понятнее:

```
currentFont.size = PointsToPixels( 12 )
```

Кроме того, атрибуту шрифта можно было бы присвоить более определенное имя, например:

```
currentFont.sizeInPixels = PointsToPixels( 12 )
```

Однако при этом вы не смогли бы включить в программу сразу два поля, определяющих размер шрифта: *currentFont.sizeInPixels* (размер шрифта в пикселах) и *currentFont.sizeInPoints* (размер шрифта в пунктах), — потому что тогда структура *currentFont* не смогла бы узнать, какое из них использовать. Кроме того, изменяя размеры шрифтов в нескольких местах, вы распространили бы похожие строки по всей программе.

Для выбора полужирного начертания вы могли бы написать код, использующий логическое ИЛИ и шестнадцатеричную константу *0x02*:

```
currentFont.attribute = currentFont.attribute or 0x02
```

Этот код можно немного улучшить, но лучшее, что вы получите, используя специализированный подход, будет похоже на:

```
currentFont.attribute = currentFont.attribute or BOLD
```

или на что-нибудь такое:

```
currentFont.bold = True
```

Как и в случае с размером шрифта, проблема здесь в том, что клиентский код должен контролировать элементы данных непосредственно, а это ограничивает число возможных способов применения структуры *currentFont*.

Такой подход к программированию способствует распространению похожих строк кода по всей программе.

## Преимущества использования АТД

Проблема не в том, что специализированный подход — плохая методика программирования. Просто вы можете заменить его на лучшую методику, преимущества которой описаны ниже.

***Возможность сокрытия деталей реализации*** Сокрытие информации о типах данных шрифта подразумевает, что при необходимости изменения типа данных вы сможете изменить его в одном месте, не влияя на всю программу. Например, если вы не скроете детали реализации в АТД, то при изменении одного вида представления полужирного шрифта на другой вам придется изменить каждый фрагмент кода, в котором задается полужирное начертание. Сокрытие информации защитит остальную часть программы и в тех случаях, если вы решите хранить данные во внешнем хранилище, а не в памяти или переписать все методы, выполняющие операции над шрифтами, на другом языке.

***Ограничение области изменений*** Если вы захотите разнообразить шрифты и реализовать для них дополнительные операции (такие как переключение на надстрочный шрифт, перечеркивание и т. д.), вы сможете изменить один фрагмент кода, и это не повлияет на остальную часть программы.

***Более высокая информативность интерфейса*** Код *currentFont.size = 16* неоднозначен, так как число *16* может определять размер шрифта и в пикселах, и в пунктах. Контекст об этом ничего не говорит. Объединение всех похожих операций в АТД позволяет определить весь интерфейс в терминах пунктов, в терминах пикселов или четко разделить оба варианта, помогая избежать путаницы.

***Легкость оптимизации кода*** Для повышения быстродействия операций над шрифтами вы сможете переписать несколько четко определенных методов, а не блуждать по всей программе.

***Легкость проверки кода*** Нудную проверку правильности команд вида *currentFont.attribute = currentFont.attribute or 0x02* вы сможете заменить более простой проверкой правильности вызовов *currentFont.SetBoldOn()*. В первом случае можно указать неверное имя структуры, неверное имя поля, неверную операцию (*and* вместо *or*) или неверное значение атрибута (*0x20* вместо *0x02*). В случае вызова *currentFont.SetBoldOn()* ошибкой может быть лишь указание неверного имени метода, так что заметить ее легче.

***Удобочитаемость и понятность кода*** Команду вида *currentFont.attribute or 0x02* можно улучшить, заменив *0x02* на *BOLD* (или что там представляет константа *0x02*), но даже после этого по удобочитаемости она не сравнится с вызовом метода *currentFont.SetBoldOn()*.

Вудфилд, Дансмор и Шен провели исследование, участники которого — аспиранты и студенты старших курсов факультета информатики — должны были ответить на вопросы о двух программах: одна была разделена на восемь методов в функциональном стиле, а вторая — на восемь методов АТД (Woodfield, Dunsmore, and Shen, 1981). Студенты, отвечавшие на вопросы о второй программе, получили на 30% более высокие оценки.

***Ограничение области использования данных*** В только что представленных примерах структуру *currentFont* нужно изменять непосредственно или передавать в каждый метод, работающий со шрифтами. При использовании АТД вам не пришлось бы ни передавать ее в методы, ни превращать в глобальные данные. АТД просто включал бы структуру, содержащую данные *currentFont*. Прямой доступ к этим данным имели бы лишь методы из состава АТД, но не какие бы то ни было другие методы.

***Возможность работы с сущностями реального мира, а не с низкоуровневыми деталями реализации*** АТД позволяет определить операции над шрифтами так, что большая часть программы будет сформулирована исключительно в терминах шрифтов, а не доступа к массивам, определений структур или значений *True* и *False*.

В нашем случае в АТД можно было бы включить методы:

```
currentFont.SetSizeInPoints( sizeInPoints )
currentFont.SetSizeInPixels( sizeInPixels )
```

```
currentFont.SetBoldOn()
currentFont.SetBoldOff()
currentFont.SetItalicOn()
currentFont.SetItalicOff()
currentFont.SetTypeFace( faceName )
```

Эти методы, вероятно, были бы короткими — пожалуй, они напоминали бы код, приведенный при обсуждении специализированного подхода к управлению шрифтами. Различие двух подходов в том, что, используя АТД, вы изолируете операции над шрифтами в наборе методов, который предоставляет остальным частям программы, работающим с шрифтами, улучшенный уровень абстракции и защищает остальной код от изменений операций над шрифтами.

## Другие примеры АТД

Допустим, вы разрабатываете приложение, управляющее системой охлаждения ядерного реактора. С системой охлаждения можно работать как с АТД, определив для нее такие операции:

```
coolingSystem.GetTemperature()
coolingSystem.SetCirculationRate( rate )
coolingSystem.OpenValve( valveNumber )
coolingSystem.CloseValve( valveNumber )
```

Конкретная реализация данных операций зависела бы от конкретной среды. Остальные фрагменты программы взаимодействовали бы с системой охлаждения при помощи этих методов и могли бы не беспокоиться о внутренних деталях реализации структур данных, их ограничениях, изменениях и т. д.

Вот дополнительные примеры абстрактных типов данных и операций, которые можно было бы для них определить:

| **Система регулирования скорости** | **Кофемолка** | **Топливный бак** |
|---|---|---|
| Задать скорость | Включить | Заполнить бак |
| Получить текущие параметры | Выключить | Слить топливо |
| Восстановить предыдущее значение скорости | Задать скорость | Получить емкость топливного бака |
| Отключить систему | Начать перемалывание кофе | Получить статус топлив ного бака |
| | Прекратить перемалывание кофе | |
| **Список** | | **Стек** |
| Инициализировать список | **Фонарь** | Инициализировать стек |
| Вставить элемент | Включить | Поместить элемент в стек |
| Удалить элемент | Выключить | Извлечь элемент из стека |
| Прочитать следующий элемент | | Прочитать верхний элемент стека |

(*см. след. стр.*)

| **Система справочной информации** | **Меню** | **Файл** |
|---|---|---|
| Добавить раздел | Создать новое меню | Открыть файл |
| Удалить раздел | Уничтожить меню | Прочитать файл |
| Задать текущий раздел | Добавить в меню новый элемент | Записать файл |
| Отобразить окно справочной системы | Удалить элемент меню | Установить указатель файла |
| Уничтожить окно справочной системы | Активировать элемент меню | Закрыть файл |
| Отобразить указатель информационных разделов | Деактивировать элемент меню | |
| Вернуться к предыдущему разделу | Отобразить меню | **Лифт** |
| | Скрыть меню | Переместиться на один этаж вверх |
| **Указатель** | Получить индекс выбранного элемента меню | Переместиться на один этаж вниз |
| Выделить блок памяти | | Переместиться на конкретный этаж |
| Освободить блок памяти | | Сообщить текущий номер этажа |
| Изменить объем выделенной памяти | | Вернуться на первый этаж |

Изучение этих примеров позволяет вывести принципы использования АТД, которые мы и обсудим.

***Представляйте в форме АТД распространенные низкоуровневые типы данных*** Обычно при обсуждении АТД речь идет о представлении в форме АТД популярных низкоуровневых типов данных. Как вы могли заметить по примерам, в форме АТД можно представить стек, список, очередь и почти любой другой популярный тип данных.

Спросите себя: «Что представляет этот стек, список или эта очередь?» Если стек представляет набор сотрудников, рассматривайте АТД как набор сотрудников, а не как стек. Если список соответствует набору счетов, рассматривайте его как набор счетов. Если очередь представляет ячейки электронной таблицы, обращайтесь с ней как с набором ячеек, а не обобщенных элементов. Используйте как можно более высокий уровень абстракции.

***Представляйте в форме АТД часто используемые объекты, такие как файлы*** Большинство языков включает несколько АТД, которые известны всем программистам, но не всегда воспринимаются как АТД. В качестве примера приведу операции над файлами. При записи данных на диск ОС избавляет вас от забот, связанных с позиционированием головки чтения/записи, выделением новых секторов на диске при заполнении старых и интерпретацией непонятных кодов

ошибок. ОС предоставляет первый уровень абстракции и соответствующие ему АТД. Высокоуровневые языки предоставляют второй уровень абстракции и АТД для этого уровня. Высокоуровневые языки скрывают от вас детали генерации вызовов ОС и работы с буферами данных. Они позволяют рассматривать область диска как «файл».

АТД можно разделить на уровни аналогичным образом. Хотите использовать АТД на уровне операций со структурами данных (таких как помещение элементов в стек и их извлечение) — прекрасно, но поверх него можно создать и другой уровень, соответствующий проблеме реального мира.

***Представляйте в форме АТД даже простые элементы*** Для оправдания использования АТД не обязательно иметь гигантский тип данных. Одним из АТД в нашем списке примеров был фонарь, поддерживающий только две операции: включение и выключение. Вам может показаться, что создавать для операций «включить» и «выключить» отдельные методы слишком расточительно, однако на самом деле АТД выгодно использовать даже в случае самых простых операций. Представление фонаря и его операций в форме АТД облегчает понимание и изменение кода, ограничивает потенциальные следствия изменений методов *TurnLightOn()* и *TurnLightOff()* и снижает число элементов данных, которые нужно передавать в методы.

***Обращайтесь к АТД так, чтобы это не зависело от среды, используемой для его хранения*** Допустим, ваша таблица страховых тарифов настолько велика, что ее нужно всегда хранить на диске. Вы могли бы представить ее как «*файл* тарифов (rate *file*)» и создать такие методы доступа, как *RateFile.Read()*. Однако, ссылаясь на таблицу как на файл, вы сообщаете о ней больше информации, чем следовало бы. Если вы когда-нибудь измените программу так, чтобы таблица хранилась в памяти, а не на диске, код, обращающийся к ней как к файлу, станет некорректным и начнет вызывать замешательство. Поэтому старайтесь присваивать классам и методам доступа имена, не зависящие от способа хранения данных, и обращайтесь не к конкретным сущностям, а к АТД, таким как таблица страховых тарифов. В нашем случае класс и метод доступа следовало бы назвать *rateTable.Read()* или просто *rates.Read()*.

## Работа с несколькими экземплярами данных при использовании АТД в средах, не являющихся объектно-ориентированными

Объектно-ориентированные языки автоматически поддерживают работу с несколькими экземплярами АТД. Если вы использовали исключительно объектно-ориентированные среды и вам не приходилось реализовывать поддержку работы с несколькими экземплярами данных, можете положиться на свою удачу! (И перейти к следующему разделу — «АТД и классы»).

Если вы программируете на C или другом языке, не являющемся объектно-ориентированным, поддержку работы с несколькими экземплярами данных нужно реализовать вручную. В целом это значит, что вы должны создать для АТД сервисы создания и уничтожения экземпляров данных и спроектировать другие сервисы АТД так, чтобы они могли работать с несколькими экземплярами.

АТД «шрифт» изначально предлагал такие сервисы:

```
currentFont.SetSize( sizeInPoints )
currentFont.SetBoldOn()
currentFont.SetBoldOff()
currentFont.SetItalicOn()
currentFont.SetItalicOff()
currentFont.SetTypeFace( faceName )
```

В среде, не являющейся объектно-ориентированной, эти методы не были бы связаны с классом и выглядели бы так:

```
SetCurrentFontSize( sizeInPoints )
SetCurrentFontBoldOn()
SetCurrentFontBoldOff()
SetCurrentFontItalicOn()
SetCurrentFontItalicOff()
SetCurrentFontTypeFace( faceName )
```

Если бы вы хотели работать с несколькими шрифтами одновременно, то должны были бы создать сервисы создания и удаления экземпляров шрифтов вроде этих:

```
CreateFont( fontId )
DeleteFont( fontId )
SetCurrentFont( fontId )
```

Идентификатор шрифта *fontId* позволяет следить за несколькими шрифтами по мере их создания и использования. Что касается других операций, то в этом случае вы можете выбирать один из трех вариантов реализации интерфейса АТД.

- Вариант 1: явно указывать экземпляр данных при каждом обращении к сервисам АТД. В этом случае «текущий шрифт (current font)» не требуется. В каждый метод, работающий со шрифтами, вы передаете *fontId*. Методы АТД *Font* следят за всеми данными шрифта, а клиентский код — лишь за идентификатором *fontId*. Этот вариант требует, чтобы каждый метод, работающий со шрифтами, принимал дополнительный параметр *fontId*.
- Вариант 2: явно предоставлять данные, используемые сервисами АТД. В данном случае вы объявляете нужные АТД данные в каждом методе, использующем сервис АТД. Иначе говоря, вы создаете тип данных *Font*, который передаете в каждый из сервисных методов АТД. Вы должны спроектировать сервисные методы АТД так, чтобы они использовали данные *Font*, передаваемые в них при каждом вызове. При этом клиентский код не нуждается в идентификаторе шрифта, потому что он следит за данными шрифтов сам. (Хотя данные типа *Font* доступны напрямую, к ним надо обращаться только через сервисные методы АТД. Это называется поддержанием структуры «в закрытом виде».)

Преимущество этого подхода в том, что сервисным методам АТД не приходится просматривать информацию о шрифте, опираясь на его идентификатор. Есть и недостаток: такой способ предоставляет доступ к данным шрифта остальным частям программы, из-за чего повышается вероятность того, что клиентский код будет использовать детали реализации АТД, которым следовало бы оставаться скрытыми внутри АТД.

- Вариант 3: использовать неявные экземпляры (с большой осторожностью). Вы должны создать новый сервис — скажем, *SetCurrentFont ( fontId )*, — при вызове которого заданный экземпляр шрифта делается текущим. После этого все остальные сервисы используют текущий шрифт, благодаря чему в них не нужно передавать параметр *fontId*. При разработке простых приложений такой подход может облегчить использование нескольких экземпляров данных. В сложных приложениях подобная зависимость от состояния в масштабе всей системы подразумевает, что вы должны следить за текущим экземпляром шрифта во всем коде, вызывающем методы *Font*; разумеется, сложность программы при этом повышается. Каким бы ни был размер приложения, всегда можно найти более удачные альтернативы данному подходу.

Внутри АТД вы можете реализовать работу с несколькими экземплярами данных как угодно, но вне его при использовании языка, не являющегося объектно-ориентированным, возможны только три указанных варианта.

### АТД и классы

Абстрактные типы данных лежат в основе концепции классов. В языках, поддерживающих классы, каждый АТД можно реализовать как отдельный класс. Однако обычно с классами связывают еще две концепции: наследование и полиморфизм. Можете рассматривать класс как АТД, поддерживающий наследование и полиморфизм.

## 6.2. Качественные интерфейсы классов

Первый и, наверное, самый важный этап разработки высококачественного класса — создание адекватного интерфейса. Это подразумевает, что интерфейс должен представлять хорошую абстракцию, скрывающую детали реализации класса.

### Хорошая абстракция

Как я говорил в подразделе «Определите согласованные абстракции» раздела 5.3, под абстракцией понимается представление сложной операции в упрощенной форме. Интерфейс класса — это абстракция реализации класса, скрытой за интерфейсом. Интерфейс класса должен предоставлять группу методов, четко согласующихся друг с другом.

Рассмотрим для примера класс «сотрудник». Он может содержать такие данные, как фамилия сотрудника, адрес, номер телефона и т. д., и предлагать методы инициализации и использования этих данных. Вот как мог бы выглядеть такой класс:

**Пример интерфейса, формирующего хорошую абстракцию (C++)**

```
class Employee {
public:
   // открытые конструкторы и деструкторы
   Employee();
   Employee(
      FullName name,
      String address,
      String workPhone,
```

**Перекрестная ссылка** Примеры кода в этой книге отформатированы с использованием конвенции, поддерживающей сходство стилей между несколькими языками. Об этой конвенции (и разных стилях кодирования) см. подраздел «Программирование с использованием нескольких языков» раздела 11.4.

```
      String homePhone,
      TaxId taxIdNumber,
      JobClassification jobClass
   );
   virtual ~Employee();

   // открытые методы
   FullName GetName() const;
   String GetAddress() const;
   String GetWorkPhone() const;
   String GetHomePhone() const;
   TaxId GetTaxIdNumber() const;
   JobClassification GetJobClassification() const;
   ...
private:
   ...
};
```

Внутри этот класс может иметь дополнительные методы и данные, поддерживающие работу этих сервисов, но пользователям класса знать о них не нужно. Представляемая интерфейсом этого класса абстракция великолепна, потому что все методы интерфейса служат единой согласованной цели.

Интерфейс, представляющий плохую абстракцию, содержал бы набор разнородных методов, например:

**Пример интерфейса, формирующего плохую абстракцию (C++)**

```
class Program {
public:
   ...
   // открытые методы
   void InitializeCommandStack();
   void PushCommand( Command command );
   Command PopCommand();
   void ShutdownCommandStack();
   void InitializeReportFormatting();
   void FormatReport( Report report );
   void PrintReport( Report report );
   void InitializeGlobalData();
   void ShutdownGlobalData();
   ...
private:
   ...
};
```

Похоже, этот класс содержит методы работы со стеком команд, форматирования отчетов, печати отчетов и инициализации глобальных данных. Трудно увидеть связь между стеком команд, обработкой отчетов и глобальными данными. Интерфейс такого класса не формирует согласованную абстракцию, и класс обладает плохой

связностью. В данном случае методы следует реорганизовать в более четкие классы, интерфейсы которых будут представлять более удачные абстракции.

Если бы эти методы были частью класса *Program*, для формирования согласованной абстракции их можно было бы изменить так:

**Пример интерфейса, формирующего более удачную абстракцию (C++)**

```
class Program {
public:
   ...
   // открытые методы
    void InitializeUserInterface();
    void ShutDownUserInterface();
    void InitializeReports();
    void ShutDownReports();
   ...
private:
   ...
};
```

В ходе очистки интерфейса одни его методы были перемещены в более подходящие классы, а другие были преобразованы в закрытые методы, используемые методом *InitializeUserInterface()* и другими методами.

Данный способ оценки абстракции класса основан на изучении открытых методов класса, т. е. его интерфейса. Однако из того, что класс в целом формирует хорошую абстракцию, вовсе не следует, что его отдельные методы также представляют удачные абстракции. Рекомендации по проектированию методов см. в разделе 7.2.

Чтобы ваши классы имели высококачественные абстрактные интерфейсы, соблюдайте при их проектировании следующие принципы.

***Выражайте в интерфейсе класса согласованный уровень абстракции***
Классы полезно рассматривать как механизмы реализации абстрактных типов данных, описанных в разделе 6.1. В идеале каждый класс должен быть реализацией только одного АТД. Если класс реализует более одного АТД или если вам не удается определить, реализацией какого АТД класс является, самое время реорганизовать класс в один или несколько хорошо определенных АТД.

Так, следующий класс имеет несогласованный интерфейс, потому что формируемый им уровень абстракции непостоянен:

**Пример интерфейса, включающего разные уровни абстракции (C++)**

```
class EmployeeCensus: public ListContainer {
public:
   ...
   // открытые методы
```

Абстракция, формируемая этими методами, относится к уровню «employee» (сотрудник).

```
    void AddEmployee( Employee employee );
    void RemoveEmployee( Employee employee );
```

Абстракция, формируемая этими методами, относится к уровню «list» (список).

```
    Employee NextItemInList();
    Employee FirstItem();
    Employee LastItem();
    ...
private:
    ...
};
```

Этот класс представляет два АТД: *Employee* и *ListContainer* (список-контейнер). Подобные смешанные абстракции часто возникают, когда программист реализует класс при помощи класса-контейнера или других библиотечных классов и не скрывает этот факт. Спросите себя, должна ли информация об использовании класса-контейнера быть частью абстракции. Обычно это является деталью реализации, которую следует скрыть от остальных частей программы, например так:

**Пример интерфейса, формирующего согласованную абстракцию (C++)**

```
class EmployeeCensus {
public:
    ...
    // открытые методы
```

Абстракция, формируемая всеми этими методами, теперь относится к уровню «employee».

```
    void AddEmployee( Employee employee );
    void RemoveEmployee( Employee employee );
    Employee NextEmployee();
    Employee FirstEmployee();
    Employee LastEmployee();
    ...
private:
```

Тот факт, что класс использует библиотеку *ListContainer*, теперь скрыт.

```
    ListContainer m_EmployeeList;
    ...
};
```

Программисты могут утверждать, что наследование от *ListContainer* удобно, потому что оно поддерживает полиморфизм, позволяя создать внешний метод поиска или сортировки, принимающий объект *ListContainer*. Но этот аргумент не проходит главный тест на уместность наследования: «Используется ли наследование только для моделирования отношения „является“?» Наследование класса *EmployeeCensus* (каталог личных дел сотрудников) от класса *ListContainer* означало бы, что *EmployeeCensus* «является» *ListContainer*, что, очевидно, неверно. Если абстракция объекта *EmployeeCensus* заключается в том, что он поддерживает поиск или сортировку,

эти возможности должны быть явными согласованными частями интерфейса класса.

Если представить открытые методы класса как люк, предотвращающий попадание воды в подводную лодку, несогласованные открытые методы — это щели. Вода не будет протекать через них так быстро, как через открытый люк, но позже лодка все же потонет. На практике при смешении уровней абстракции именно это и происходит. По мере изменений программы смешанные уровни абстракции делают ее все менее и менее понятной, пока в итоге код не станет совсем загадочным.

***Убедитесь, что вы понимаете, реализацией какой абстракции является класс*** Некоторые классы очень похожи, поэтому при разработке класса нужно понимать, какую абстракцию должен представлять его интерфейс. Однажды я работал над программой, которая должна была поддерживать редактирование информации в табличном формате. Сначала мы хотели использовать простой элемент управления «grid» (сетка), но доступные элементы управления этого типа не позволяли закрашивать ячейки ввода данных в другой цвет, поэтому мы выбрали элемент управления «spreadsheet» (электронная таблица), который такую возможность поддерживал.

Элемент управления «электронная таблица» был гораздо сложнее «сетки» и предоставлял около 150 методов в сравнении с 15 методами «сетки». Так как наша цель заключалась в использовании «сетки», а не «электронной таблицы», мы поручили одному программисту написать класс-оболочку, который скрывал бы тот факт, что мы подменили один элемент управления другим. Он поворчал по поводу ненужных затрат и бюрократии, ушел и вернулся через пару дней с классом-оболочкой, который честно предоставлял все 150 методов «электронной таблицы».

Но нам было нужно не это — нам требовался интерфейс «сетки», инкапсулирующий тот факт, что за кулисами мы использовали гораздо более сложную «электронную таблицу». Программисту следовало предоставить доступ только к 15 методам «сетки» и еще одному, шестнадцатому методу, поддерживающему закрашивание ячеек. Открыв доступ ко всем 150 методам, программист подверг нас риску того, что после нескольких изменений реализации класса нам в итоге придется поддерживать все 150 открытых методов. Он не смог обеспечить нужную нам инкапсуляцию и проделал гораздо больше работы, чем стоило.

В зависимости от конкретных обстоятельств оптимальной абстракцией может оказаться как «сетка», так и «электронная таблица». Если приходится выбирать между двумя похожими абстракциями, убедитесь, что выбор правилен.

***Предоставляйте методы вместе с противоположными им методами*** Большинство операций имеет соответствующие противоположные операции. Если одна из операций включает свет, вам, вероятно, понадобится и операция, его выключающая. Если одна операция добавляет элемент в список, элементы скорее всего нужно будет и удалять. Если одна операция активизирует элемент меню, вторая, наверное, должна будет его деактивизировать. При проектировании класса проверьте каждый открытый метод на предмет того, требуется ли вам его противоположность. Создавать противоположные методы, не имея на то причин, не следует, но проверить их целесообразность нужно.

***Убирайте постороннюю информацию в другие классы*** Иногда вы будете обнаруживать, что одни методы класса работают с одной половиной данных, а другие — с другой. Это значит, что вы имеете дело с двумя классами, скрывающимися под маской одного. Разделите их!

***По мере возможности делайте интерфейсы программными, а не семантическими*** Каждый интерфейс состоит из программной и семантической частей. Первая включает типы данных и другие атрибуты интерфейса, которые могут быть проверены компилятором. Вторая складывается из предположений об использовании интерфейса, которые компилятор проверить не может. Семантический интерфейс может включать такие соображения, как «Метод А должен быть вызван перед Методом В» или «Метод А вызовет ошибку, если переданный в него Элемент Данных 1 не будет перед этим инициализирован». Семантический интерфейс следует документировать в комментариях, но вообще интерфейсы должны как можно меньше зависеть от документации. Любой аспект интерфейса, который не может быть проверен компилятором, является потенциальным источником ошибок. Старайтесь преобразовывать семантические элементы интерфейса в программные, используя утверждения (assertions) или иными способами.

**Перекрестная ссылка** О поддержании качества кода при его изменении см. главу 24.

***Опасайтесь нарушения целостности интерфейса при изменении класса*** При модификации и расширении класса часто обнаруживается дополнительная нужная функциональность, которая не совсем хорошо соответствует интерфейсу первоначального класса, но плохо поддается реализации иным образом. Так, класс *Employee* может превратиться во что-нибудь вроде:

**Пример интерфейса, изуродованного при сопровождении программы (C++)**

```
class Employee {
public:
   ...
   // открытые методы
   FullName GetName() const;
   Address GetAddress() const;
   PhoneNumber GetWorkPhone() const;
   ...
   bool IsJobClassificationValid( JobClassification jobClass );
   bool IsZipCodeValid( Address address );
   bool IsPhoneNumberValid( PhoneNumber phoneNumber );

   SqlQuery GetQueryToCreateNewEmployee() const;
   SqlQuery GetQueryToModifyEmployee() const;
   SqlQuery GetQueryToRetrieveEmployee() const;
   ...
private:
   ...
};
```

То, что начиналось как ясная абстракция, превратилось в смесь почти несогласованных методов. Между сотрудниками и методами, проверяющими корректность

почтового индекса, номера телефона или ставки зарплаты (job classification), нет логической связи. Методы, предоставляющие доступ к деталям SQL-запросов, относятся к гораздо более низкому уровню абстракции, чем класс *Employee*, нарушая общую абстракцию класса.

***Не включайте в класс открытые члены, плохо согласующиеся с абстракцией интерфейса*** Добавляя новый метод в интерфейс класса, всегда спрашивайте себя: «Согласуется ли этот метод с абстракцией, формируемой существующим интерфейсом?» Если нет, найдите другой способ внесения изменения, позволяющий сохранить согласованность абстракции.

***Рассматривайте абстракцию и связность вместе*** Понятия абстракции и связности (cohesion) тесно связаны: интерфейс класса, представляющий хорошую абстракцию, обычно отличается высокой связностью. И наоборот: классы, имеющие высокую связность, обычно представляют хорошие абстракции, хотя эта связь выражена слабее.

Я обнаружил, что при повышенном внимании к абстракции, формируемой интерфейсом класса, проект класса получается более удачным, чем при концентрации на связности класса. Если вы видите, что класс имеет низкую связность и не знаете, как это исправить, спросите себя, представляет ли он согласованную абстракцию.

## Хорошая инкапсуляция

Как я уже говорил в разделе 5.3, инкапсуляция является более строгой концепцией, чем абстракция. Абстракция помогает управлять сложностью, предоставляя модели, позволяющие игнорировать детали реализации. Инкапсуляция не позволяет узнать детали реализации, даже если вы этого захотите.

**Перекрестная ссылка** Об инкапсуляции см. подраздел «Инкапсулируйте детали реализации» раздела 5.3.

Две этих концепции связаны: без инкапсуляции абстракция обычно разрушается. По своему опыту могу сказать, что вы или имеете и абстракцию, и инкапсуляцию, или не имеете ни того, ни другого. Промежуточных вариантов нет.

***Минимизируйте доступность классов и их членов*** Минимизация доступности — одно из нескольких правил, поддерживающих инкапсуляцию. Если вы не можете понять, каким делать конкретный метод: открытым, закрытым или защищенным, — некоторые авторы советуют выбирать самый строгий уровень защиты, который работает (Meyers, 1998; Bloch, 2001). По-моему, это прекрасное правило, но мне кажется, что еще важнее спросить себя: «Какой вариант лучше всего сохраняет целостность абстракции интерфейса?» Если предоставление доступа к методу согласуется с абстракцией, сделайте его открытым. Если вы не уверены, скрыть больше обычно предпочтительнее, чем скрыть меньше.

> Самым важным отличием хорошо спроектированного модуля от плохо спроектированного является степень, в которой модуль скрывает свои внутренние данные и другие детали реализации от других модулей.
>
> *Джошуа Блох* (*Joshua Bloch*)

***Не делайте данные-члены открытыми*** Предоставление доступа к данным-членам нарушает инкапсуляцию и ограничивает контроль над абстракцией. Как

указывает Артур Риэль, класс *Point* (точка), который предоставляет доступ к данным:

```
float x;
float y;
float z;
```

нарушает инкапсуляцию, потому что клиентский код может свободно делать с данными *Point* что угодно, при этом сам класс может даже не узнать об их изменении (Riel, 1996). В то же время класс *Point*, включающий члены:

```
float GetX();
float GetY();
float GetZ();
void SetX( float x );
void SetY( float y );
void SetZ( float z );
```

поддерживает прекрасную инкапсуляцию. Вы не имеете понятия о том, реализованы ли данные как *float x*, *y* и *z*, хранит ли класс *Point* эти элементы как *double*, преобразуя их в *float*, или же он хранит их на Луне и получает через спутник.

***Не включайте в интерфейс класса закрытые детали реализации*** Истинная инкапсуляция не позволяла бы узнать детали реализации вообще. Они были бы скрыты и в прямом, и в переносном смыслах. Однако популярные языки — в том числе C++ — требуют, чтобы программисты раскрывали детали реализации в интерфейсе класса, например:

**Пример обнародования деталей реализации класса (C++)**

```
class Employee {
public:
   ...
   Employee(
      FullName name,
      String address,
      String workPhone,
      String homePhone,
      TaxId taxIdNumber,
      JobClassification jobClass
   );
   ...
   FullName GetName() const;
   String GetAddress() const;
   ...
private:

Обнародованные детали реализации.

   String m_Name;
   String m_Address;
   int m_jobClass;
   ...
};
```

Включение объявлений закрытых членов в заголовочный файл класса может показаться не таким уж и серьезным нарушением, но оно поощряет других программистов изучать детали реализации. В нашем случае предполагается, что использовать адреса в клиентском коде нужно как типы *Address*, однако, заглянув в заголовочный файл, можно узнать, что адреса хранятся как типы *String*.

Общий способ решения этой проблемы описал Скотт Мейерс в разделе 34 книги «Effective C++, 2d ed» (Meyers, 1998). Отделите интерфейс класса от его реализации, после чего включите в объявление класса указатель на его реализацию, но не включайте других деталей реализации.

#### Пример сокрытия деталей реализации класса (C++)

```
class Employee {
public:
   ...
   Employee( ... );
   ...
   FullName GetName() const;
   String GetAddress() const;
   ...
private:

Детали реализации скрыты при помощи указателя.

   EmployeeImplementation *m_implementation;
};
```

Теперь вы можете поместить детали реализации в класс *EmployeeImplementation*, который будет доступен только классу *Employee*, но не использующему этот класс коду.

Если вы уже написали много кода, не используя этой методики, то можете найти преобразование кода неоправданным. Что ж, в этом случае, *читая* код, раскрывающий детали реализации, постарайтесь хотя бы сопротивляться соблазну изучить *закрытые* разделы интерфейсов классов.

***Не делайте предположений о клиентах класса*** Класс следует спроектировать и реализовать так, чтобы он придерживался контракта, сформулированного посредством интерфейса. Выразив свои требования в интерфейсе, класс не должен делать предположений о том, как этот интерфейс будет или не будет использоваться. Подобные комментарии указывают на то, что класс требует от своих клиентов больше, чем следует:

```
-- инициализируйте x, y и z значением 1.0, потому что
-- при инициализации значением 0.0 DerivedClass не работает
```

***Избегайте использования дружественных классов*** Иногда — например, при реализации шаблона Состояние (State) — дисциплинированное использование дружественных классов помогает управлять сложностью (Gamma et al., 1995). Однако обычно дружественные классы нарушают инкапсуляцию. Они увеличивают объем кода, о котором приходится думать в каждый конкретный момент времени, повышая тем самым сложность программы.

***Не делайте метод открытым лишь потому, что он использует только открытые методы*** То, что метод использует только открытые методы, не играет особой роли. Лучше спросите себя, согласуется ли предоставление доступа к данному методу с абстракцией, формируемой интерфейсом.

***Цените легкость чтения кода выше, чем удобство его написания*** Даже во время первоначальной разработки программы код приходится читать гораздо чаще, чем писать. Выгода от подхода, повышающего удобство написания кода за счет легкости его чтения, обманчива. При разработке интерфейсов классов это справедливо вдвойне. Даже если метод плохо согласуется с абстракцией интерфейса, иногда так и тянет включить его в интерфейс, чтобы облегчить работу над конкретным клиентом класса. Однако это первый шаг к беде, и о нем лучше даже не помышлять.

> Если для понимания того, что происходит, нужно увидеть реализацию, это не абстракция.
>
> *Ф. Дж. Плоджер (P. J. Plauger)*

***Очень, очень настороженно относитесь к семантическим нарушениям инкапсуляции*** Когда-то мне казалось, что, научившись избегать синтаксических ошибок, я обрету покой. Но вскоре я обнаружил, что это просто открыло передо мной дверь в мир совершенно новых ошибок, большинство которых диагностировать и исправлять сложнее, чем синтаксические.

Аналогичные отношения имеют место между синтаксической и семантической инкапсуляцией. С точки зрения синтаксиса, не совать нос во внутренние дела другого класса относительно легко: достаточно просто объявить его внутренние методы и данные закрытыми. Достичь семантической инкапсуляции гораздо сложнее. Вот несколько примеров того, как вы можете нарушить инкапсуляцию семантически. Вы можете:

- решить не вызывать метод *InitializeOperations()* Класса A, потому что метод *PerformFirstOperation()* Класса A вызывает его автоматически;
- не вызвать метод *database.Connect()* перед вызовом метода *employee.Retrieve( database )*, потому что знаете, что при отсутствии соединения с БД метод *employee.Retrieve()* его установит;
- не вызвать метод *Terminate()* Класса A, так как знаете, что метод *PerformFinalOperation()* Класса A уже вызвал его;
- использовать указатель или ссылку на Объект B, созданный Объектом A, даже после выхода Объекта A из области видимости, потому что знаете, что Объект A хранит Объект B в статическом хранилище, вследствие чего Объект B все еще будет корректным;
- использовать константу *MAXIMUM_ELEMENTS* Класса B вместо константы *MAXIMUM_ELEMENTS* Класса A, потому что знаете, что они имеют одинаковые значения.

Со всеми этими примерами связана одна и та же проблема: зависимость клиентского кода от закрытой реализации класса, а не от его открытого интерфейса. Каждый раз, когда вы смотрите на реализацию класса, чтобы узнать, как его использовать, вы программируете не в соответствии с интерфейсом, а *сквозь* интерфейс *в соответствии* с реализацией. Программирование

сквозь интерфейс разрушает инкапсуляцию, а вскоре к ней присоединяется и абстракция.

Если исключительно по документации интерфейса разобраться с использованием класса не удается, изучение реализации класса по исходному коду *не* будет грамотным решением. Это хорошая инициатива, но плохое решение. Вы поступите правильно, если свяжетесь с автором класса и скажете ему: «Я не могу понять, как использовать этот класс». Автор класса поступит правильно, если *не* ответит вам, а изучит файл интерфейса, изменит соответствующую документацию, зарегистрирует файл в общих исходных кодах проекта и скажет: «Посмотрите, поймете ли вы работу класса сейчас». Желательно, чтобы этот диалог происходил в самом коде интерфейса: так он будет сохранен для будущих программистов. Если диалог будет происходить исключительно в вашем уме, это внесет тонкие семантические зависимости в код клиентов класса. Если же он будет межличностным, выгоду сможете извлечь только вы, и больше никто — это некрасиво.

***Остерегайтесь слишком жесткого сопряжения*** «Сопряжение» (coupling) характеризует силу связи между двумя классами. Как правило, чем сопряжение слабее, тем лучше. Из этого можно вывести несколько общих правил:

- минимизируйте доступность классов и их членов;
- избегайте дружественных классов, потому что они связаны жестко;
- делайте данные базового класса закрытыми, а не защищенными: это ослабляет сопряжение производных классов с базовым;
- не включайте данные-члены в открытый интерфейс класса;
- остерегайтесь семантических нарушений инкапсуляции;
- соблюдайте «Правило Деметры» (см. раздел 6.3).

Сопряжение идет рука об руку с абстракцией и инкапсуляцией. Жесткое сопряжение наблюдается при неудачной абстракции или нарушениях инкапсуляции. Если класс предлагает неполный набор услуг, другие методы могут попытаться прочитать или записать его данные непосредственно. Это открывает класс, превращая его из черного ящика в прозрачный, и практически устраняет инкапсуляцию.

# 6.3. Вопросы проектирования и реализации

Для создания высококачественной программы недостаточно определить удачные интерфейсы классов — не менее важно грамотно спроектировать и реализовать внутреннее устройство классов. В этом разделе мы обсудим вопросы, связанные с включением, наследованием, методами/данными-членами, сопряжением классов, конструкторами, а также объектами-значениями и объектами-ссылками.

## Включение (отношение «содержит»)

Сущность включения (containment) проста: один класс содержит примитивный элемент данных или другой класс. Наследованию в литературе уделяют гораздо больше внимания, но это объясняется его сложностью и подверженностью ошибкам, а не тем, что оно лучше включения. Включение — один из главных инструментов объектно-ориентированного программирования.

***Реализуйте с помощью включения отношение «содержит»*** Включение можно рассматривать как отношение «содержит». Например, объект «сотрудник» может «содержать» фамилию, номер телефона, идентификационный номер налогоплательщика и т. д. Это отношение можно реализовать, сделав фамилию, номер телефона и номер налогоплательщика данными-членами класса *Employee*.

***В самом крайнем случае реализуйте отношение «содержит» при помощи закрытого наследования*** Иногда включение не получается реализовать, делая один объект членом другого. Некоторые эксперты советуют при этом выполнять закрытое наследование класса-контейнера от класса, который должен в нем содержаться (Meyers, 1998; Sutter, 2000). Главным мотивом такого решения является предоставление классу-контейнеру доступа к защищенным методам/данным-членам содержащегося в нем класса. На практике этот подход устанавливает слишком близкие отношения между дочерним и родительским классом, нарушая инкапсуляцию. Обычно это указывает на ошибки проектирования, которые следует решить иначе, не прибегая к закрытому наследованию.

***Настороженно относитесь к классам, содержащим более семи элементов данных-членов*** При выполнении других заданий человек может удерживать в памяти 7±2 дискретных элементов (Miller, 1956). Если класс содержит более семи элементов данных-членов, подумайте, не разделить ли его на несколько менее крупных классов (Riel, 1996). Можете ориентироваться на верхнюю границу диапазона «7±2», если данные-члены являются примитивными типами, такими как целые числа и строки, и на нижнюю, если они являются сложными объектами.

## Наследование (отношение «является»)

Наследование подразумевает, что один класс является более специализированным вариантом другого класса. Цель наследования — создать более простой код, что достигается путем определения базового класса, идентифицирующего общие элементы двух или более производных классов. Общими элементами могут быть интерфейсы методов, их реализация, данные-члены или типы данных. Наследование помогает избегать повторения кода и данных в нескольких местах, централизуя их в базовом классе.

Планируя использовать наследование, вы должны принять несколько решений.

- Будет ли конкретный метод-член доступен производным классам? Будет ли он иметь реализацию по умолчанию? Можно ли будет переопределить его реализацию по умолчанию?
- Будут ли конкретные данные-члены (в том числе переменные, именованные константы, перечисления и т. д.) доступны производным классам?

Ниже аспекты этих решений обсуждаются подробнее.

> Самое важное правило объектно-ориентированного программирования на C++ таково: открытое наследование означает «является». Запомните это.
>
> *Скотт Мейерс (Scott Meyers)*

***Реализуйте при помощи открытого наследования отношение «является»*** Если программист решает создать новый класс путем наследования его от существующего класса, он по сути говорит, что новый класс «является» более специализированной версией существующего класса.

Базовый класс формулирует ожидания и ограничения, которым должен будет соответствовать производный класс (Meyers, 1998).

Если производный класс не собирается *полностью* придерживаться контракта, определенного интерфейсом базового класса, наследование выполнять не стоит. Попробуйте вместо этого применить включение или внести изменение на более высоком уровне иерархии наследования.

***Проектируйте и документируйте классы с учетом возможности наследования или запретите его*** Наследование повышает сложность программы, и в этом смысле оно может быть опасным. Поэтому гуру программирования на Java Джошуа Блох и сказал: «Проектируйте и документируйте классы с учетом возможности наследования или запретите его». Если при проектировании класса вы решили, что он не должен поддерживать наследование, не объявляйте его члены как *virtual* в случае C++ или *overridable* в случае Microsoft Visual Basic; если вы программируете на Java, объявите члены такого класса как *final*.

***Соблюдайте принцип подстановки Лисков (Liskov Substitution Principle, LSP)*** Барбара Лисков как-то заявила, что наследование стоит использовать, только если производный класс действительно «является» более специализированной версией базового класса (Liskov, 1988). Энди Хант и Дэйв Томас сформулировали LSP так: «Клиенты должны иметь возможность использования подклассов через интерфейс базового класса, не замечая никаких различий» (Hunt and Thomas, 2000).

Иначе говоря, все методы базового класса должны иметь в каждом производном классе то же значение.

Если у вас есть базовый класс *Account* (счет) и производные классы *CheckingAccount* (счет до востребования), *SavingsAccount* (депозитный счет) и *AutoLoanAccount* (счет ссуд), то при вызове каких бы то ни было методов класса *Account* в любом из его подтипов программист не должен заботиться о подтипе конкретного объекта «счет».

При соблюдении принципа подстановки Лисков наследование — мощное средство снижения сложности, позволяющее программисту сосредоточиться на общих атрибутах объекта, не волнуясь об его деталях. Если же программист должен постоянно помнить о семантических различиях реализаций подклассов, наследование только повышает сложность. Так, в нашем примере программисту пришлось бы думать: «Если я вызываю метод *InterestRate()* (процентная ставка) класса *CheckingAccount* или *SavingsAccount*, он возвращает процент, который банк выплачивает клиенту, однако метод *InterestRate()* класса *AutoLoanAccount* возвращает процент, выплачиваемый клиентом банку, поэтому я должен изменить знак результата». В соответствии с LSP, в данном случае класс *AutoLoanAccount* не должен быть производным от класса *Account*, потому что методы *InterestRate()* в этих классах имеют разные семантические значения.

***Убедитесь, что вы наследуете только то, что хотите наследовать*** Производный класс может наследовать интерфейсы методов-членов, их реализации или и то, и другое (табл. 6-1).

**Табл. 6-1. Разновидности наследуемых методов**

| | Переопределение метода возможно | Переопределение метода невозможно |
|---|---|---|
| Реализация по умолчанию имеется | Переопределяемый метод | Непереопределяемый метод. |
| Реализация по умолчанию отсутствует | Абстрактный переопределяемый метод | Этот вариант не используется (нет смысла в том, чтобы оставить метод без определения, не позволив его переопределить). |

Как следует из таблицы, наследуемые методы могут относиться к одной из трех категорий:

- абстрактный переопределяемый метод: производный класс наследует интерфейс метода, но не его реализацию;
- переопределяемый метод: производный класс наследует интерфейс метода и его реализацию по умолчанию, а также может переопределить эту реализацию;
- непереопределяемый метод: производный класс наследует интерфейс метода и его реализацию по умолчанию, переопределить которую не может.

Создавая новый класс при помощи наследования, обдумайте тип наследования каждого метода-члена. Не наследуйте реализацию только потому, что вы наследуете интерфейс, и не наследуйте интерфейс только для того, чтобы унаследовать реализацию. Если вам нужна реализация класса, но не его интерфейс, используйте включение, а не наследование.

***Не «переопределяйте» непереопределяемые методы-члены*** И C++, и Java позволяют программисту переопределить непереопределяемый метод-член — ну, или что-то вроде того. Если функция объявлена в базовом классе как *private*, в производном классе можно создать функцию с тем же именем. Программист, изучающий код производного класса, может прийти к ложному выводу, что эта функция является полиморфной, хотя на самом деле это не так — просто у нее то же имя. Иначе сформулировать это правило можно так: «Не используйте имена непереопределяемых методов базового класса в производных классах».

***Перемещайте общие интерфейсы, данные и формы поведения на как можно более высокий уровень иерархии наследования*** Чем ближе интерфейсы, данные и формы поведения к корню дерева наследования, тем легче производным классам их использовать. Какой уровень считать слишком высоким? Руководствуйтесь соображениями *абстракции*. Если вам кажется, что перемещение метода на более высокий уровень нарушит абстракцию соответствующего класса, не делайте этого.

***С подозрением относитесь к классам, объекты которых создаются в единственном экземпляре*** Использование единственного экземпляра класса может указывать на то, что вы спутали объекты с классами. Подумайте, можно ли просто создать объект вместо нового класса. Можно ли конкретный производный класс представить только данными, а не отдельным классом? Шаблон Одиночка (Singleton) — примечательное исключение из этого правила.

***С подозрением относитесь к базовым классам, имеющим только один производный класс*** Когда я вижу базовый класс, имеющий только один производный класс, то начинаю подозревать, что какой-то программист «проектировал наперед» — пытался предвосхитить будущие потребности, скорее всего не понимая их в полной мере. Лучший способ подготовки к будущей работе — не проектировать дополнительные уровни базовых классов, которые «когда-нибудь могут понадобиться», а написать максимально ясный, понятный и простой код. Это означает, что иерархию наследования не надо усложнять без крайней нужды.

***С подозрением относитесь к классам, которые переопределяют метод, оставляя его пустым*** Как правило, это говорит о неудачном проектировании базового класса. Допустим, вы создали класс *Cat*, включающий метод *Scratch()* (царапать), но после обнаружили, что некоторые коты лишены когтей и не могут царапаться. Вы могли бы унаследовать от класса *Cat* класс *ScratchlessCat*, переопределив в нем метод *Scratch()* так, чтобы он ничего не делал. Однако этот подход связан с рядом проблем.

- Он нарушает абстракцию (контракт интерфейса) класса *Cat*, изменяя семантику его интерфейса.
- При расширении на другие производные классы этот подход быстро становится неуправляемым. Что будет, когда вы найдете кота без хвоста? Или кота, который не ловит мышей? Или кота, который не пьет молоко? В итоге у вас могут появиться производные классы вроде *ScratchlessTaillessMicelessMilklessCat*.
- Код, написанный по этой методике, трудно сопровождать, потому что со временем поведение производных классов начинает сильно отличаться от интерфейсов и форм поведения базовых классов.

Исправлять эту проблему следует не в базовом классе, а в первоначальном классе *Cat*. Создайте класс *Claws* (когти) и включите его в класс *Cats*. Корень наших бед — предположение, что все коты царапаются; предложенный способ позволит устранить причину проблемы, а не бороться с ее следствиями.

***Избегайте многоуровневых иерархий наследования*** Объектно-ориентированное программирование поддерживает массу способов управления сложностью. Но использование любого мощного средства сопряжено с риском, и некоторые объектно-ориентированные подходы часто повышают сложность вместо того, чтобы снижать ее.

Артур Риэль в прекрасной книге «Object-Oriented Design Heuristics» (Riel, 1996) предлагает ограничивать иерархии наследования максимум шестью уровнями. Он основывает свой совет на «магическом числе 7±2», но мне кажется, что это слишком оптимистично. Опыт подсказывает мне, что большинству людей трудно удержать в уме более двух или трех уровней наследования сразу. «Магическое число 7±2» скорее характеризует максимально допустимое *общее количество подклассов* базового класса, а не уровней иерархии наследования.

Создание многоуровневых иерархий наследования значительно повышает число ошибок (Basili, Briand, and Melo, 1996). Тот, кто занимался отладкой сложной иерархии наследования, знает причину этого. Многоуровневые иерархии повышают сложность, что диаметрально противоположно цели наследования. Помните про

Главный Технический Императив и убедитесь, что вы используете наследование, чтобы избежать дублирования кода и *минимизировать сложность*.

***Предпочитайте полиморфизм, а не крупномасштабную проверку типов***
Наличие в коде большого числа блоков *case* может указывать на то, что программу лучше было бы спроектировать, используя наследование, хотя это верно не всегда. Вот классический пример кода, призывающего к использованию более объектно-ориентированного подхода:

**Пример кода, который следовало бы заменить вызовом полиморфного метода (C++)**

```
switch ( shape.type ) {
   case Shape_Circle:
      shape.DrawCircle();
      break;
   case Shape_Square:
      shape.DrawSquare();
      break;
   ...
}
```

Здесь методы *shape.DrawCircle()* и *shape.DrawSquare()* следует заменить на единственный метод *shape.Draw()*, поддерживающий рисование и окружностей, и прямоугольников.

С другой стороны, иногда блоки *case* служат для разделения по-настоящему разных видов объектов или форм поведения. Так, следующий фрагмент вполне уместен в объектно-ориентированной программе:

**Пример кода, который, пожалуй, не следует заменять вызовом полиморфного метода (C++)**

```
switch ( ui.Command() ) {
   case Command_OpenFile:
      OpenFile();
      break;
   case Command_Print:
      Print();
      break;
   case Command_Save:
      Save();
      break;
   case Command_Exit:
      ShutDown();
      break;
   ...
}
```

В данном случае можно было бы создать базовый класс и унаследовать от него ряд производных классов, выполняющих каждую команду при помощи полиморфного метода *DoCommand()* (как в шаблоне Команда). Но в подобной простой ситуа-

ции это неуместно: имя метода *DoCommand()* было бы настолько туманным, что почти утратило бы всякий смысл, тогда как блоки *case* довольно информативны.

***Делайте все данные закрытыми, а не защищенными*** Как говорит Джошуа Блох, «наследование нарушает инкапсуляцию» (Bloch, 2001). Выполняя наследование от класса, вы получаете привилегированный доступ к его защищенным методам и данным. Если производному классу на самом деле нужен доступ к атрибутам базового класса, включите в базовый класс защищенные методы доступа.

### Множественное наследование

Наследование — мощный и... довольно опасный инструмент. В некотором смысле наследование похоже на цепную пилу: при соблюдении мер предосторожности оно может быть невероятно полезным, но при неумелом обращении последствия могут оказаться очень и очень серьезными.

> С множественным наследованием в C++ связан один неоспоримый факт: оно открывает ящик Пандоры, полный проблем, которые просто невозможны при единичном наследовании.
>
> *Скотт Мейерс* (*Scott Meyers*)

Если наследование — цепная пила, то множественное наследование — это старинная цепная пила с барахлящим мотором, не имеющая предохранителей и не поддерживающая автоматического отключения. Иногда такой инструмент может пригодиться, но большую часть времени его лучше хранить в гараже под замком.

Некоторые эксперты рекомендуют широкое применение множественного наследования (Meyer, 1997), но по опыту могу сказать, что оно полезно главным образом только при создании «миксинов» — простых классов, позволяющих добавить ряд свойств в другой класс. Миксины называются так потому, что они позволяют «подмешать (mix in)» свойства в производные классы. Миксинами могут быть классы вроде *Displayable*, *Persistent*, *Serializable* или *Sortable*. Миксины почти всегда являются абстрактными и не поддерживают создания экземпляров независимо от других объектов.

Миксины требуют множественного наследования, но пока все миксины по-настоящему независимы друг от друга, вы можете не бояться классической проблемы, связанной с ромбовидной схемой наследования. Кроме того, «объединяя» атрибуты, они делают проект системы понятнее. Программисту легче разобраться с объектом, использующим миксины *Displayable* и *Persistent*, а не 11 более конкретных методов, которые понадобились бы для реализации этих двух свойств в противном случае.

Похоже, разработчики Java и Visual Basic понимали ценность миксинов, разрешив множественное наследование интерфейсов, но только единичное наследование классов. C++ поддерживает множественное наследование и интерфейсов, и реализации. Используйте множественное наследование, только тщательно рассмотрев все альтернативные варианты и проанализировав влияние выбранного подхода на сложность и понятность системы.

### Почему правил наследования так много?

В этом разделе были описаны многие правила избавления от проблем, связанных с наследованием. Все эти правила подразумевают, что *наследование часто противоречит главному техническому императиву програм-*

*мирования — управлению сложностью*. Ради управления сложностью относитесь к наследованию с подозрением. Вот как использовать наследование и включение:

**Перекрестная ссылка** О сложности см. подраздел «Главный Технический Императив Разработки ПО: управление сложностью» раздела 5.2.

- если несколько классов имеют общие данные, но не формы поведения, создайте общий объект, который можно было бы включить во все эти классы;
- если несколько классов имеют общие формы поведения, но не данные, сделайте эти классы производными от общего базового класса, определяющего общие методы;
- если несколько классов имеют общие данные и формы поведения, сделайте эти классы производными от общего базового класса, определяющего общие данные и методы;
- используйте наследование, если хотите, чтобы интерфейс определялся базовым классом, и включение, если хотите сами контролировать интерфейс.

## Методы-члены и данные-члены

**Перекрестная ссылка** О методах в общем см. главу 7.

Ниже я даю несколько советов по эффективной реализации методов-членов и данных-членов.

***Включайте в класс как можно меньше методов*** В одном исследовании программ на C++ было обнаружено, что большему числу методов в расчете на один класс соответствует большее число изъянов (Basili, Briand, and Melo, 1996). Однако важнее оказались другие конкурирующие факторы, в том числе многоуровневые иерархии наследования, большое число методов, вызываемых из класса, и сильное сопряжение между классами. Разрабатывая класс, стремитесь к оптимальному соответствию между этими факторами и минимальным числом методов.

***Блокируйте неявно сгенерированные методы и операторы, которые вам не нужны*** Иногда некоторые возможности, такие как создание объекта или присваивание, целесообразно блокировать. Вам может показаться, что сделать это невозможно, потому что компилятор генерирует эти операции автоматически. Однако вы можете запретить их использование в клиентском коде, объявив конструктор, оператор присваивания или другой метод или оператор как *private*. (Создание закрытого конструктора — стандартный способ определения класса-одиночки, о чем см. ниже.)

***Минимизируйте число разных методов, вызываемых классом*** Одно исследование показало, что число дефектов в коде класса статистически коррелирует с общим числом методов, вызываемых классом (Basili, Briand, and Melo, 1996). То же исследование показало, что число дефектов в коде класса повышается и при увеличении числа используемых в нем классов. Эти концепции иногда называют «коэффициентом разветвления по выходу (fan out)».

***Избегайте опосредованных вызовов методов других классов*** Непосредственные связи довольно опасны. Опосредованные связи, такие как *account.ContactPerson().DaytimeContactInfo().PhoneNumber()*, опасны еще больше. В связи с этим ученые сформулировали «Правило Деметры (Law of Demeter)» (Lieberherr and Holland, 1989), которое гласит, что Объект А может вызывать любые из собственных методов. Если он создает Объект В, он может вызывать любые методы Объекта

B, но ему не следует вызывать методы объектов, возвращаемых Объектом B. В нашем случае это означает, что вызов *account.ContactPerson()* приемлем, однако вызова *account.ContactPerson() .DaytimeContactInfo()* следовало бы избежать.

Это упрощенное объяснение — подробнее см. в книгах, указанных в конце главы.

**Дополнительные сведения** Хорошее обсуждение «Правила Деметры» см. в книгах «Pragmatic Programmer» (Hunt and Thomas, 2000), «Applying UML and Patterns» (Larman, 2001) и «Fundamentals of Object-Oriented Design in UML» (Page-Jones, 2000).

***Вообще минимизируйте сотрудничество класса с другими классами*** Старайтесь свести к минимуму все следующие показатели:

- число видов создаваемых объектов;
- число непосредственно вызываемых методов созданных объектов;
- число вызовов методов, принадлежащих объектам, возвращенным другими созданными объектами.

## Конструкторы

Советы по использованию конструкторов почти не зависят от языка (по крайней мере это касается C++, Java и Visual Basic). С деструкторами связано больше различий — см. материалы, указанные в разделе «Дополнительные ресурсы».

***Инициализируйте по мере возможности все данные-члены во всех конструкторах*** Инициализация всех данных-членов во всех конструкторах — простой прием защитного программирования.

***Создавайте классы-одиночки с помощью закрытого конструктора*** Если вы хотите определить класс, позволяющий создать только один объект, скройте все конструкторы класса и создайте статический метод *GetInstance()*, предоставляющий доступ к единственному экземпляру класса:

**Дополнительные сведения** Аналогичный код, написанный на C++, был бы очень похож. Подробнее см. раздел 26 книги «More Effective C++» (Meyers, 1998).

**Пример создания класса-одиночки с помощью закрытого конструктора (Java)**

```
public class MaxId {
   // конструкторы и деструкторы

Закрытый конструктор.
   private MaxId() {
      ...
   }
   ...
   // открытые методы

Открытый метод, предоставляющий доступ к единственному экземпляру класса.
   public static MaxId GetInstance() {
      return m_instance;
   }
   ...
   // закрытые члены
```

```
Единственный экземпляр класса.
    private static final MaxId m_instance = new MaxId();
    ...
}
```

Закрытый конструктор вызывается только при инициализации статического объекта *m_instance*. Для обращения к классу-одиночке *MaxId* нужно просто вызвать метод *MaxId.GetInstance()*.

***Если сомневаетесь, выполняйте полное копирование, а не ограниченное*** Одним из главных аспектов работы со сложными объектами является выбор типа их копирования: полного или ограниченного. Полная копия (deep copy) — это почленная копия данных-членов объекта; ограниченная копия (shallow copy) обычно просто указывает или ссылается на исходный объект, хотя конкретные значения «полного» и «ограниченного» копирования могут различаться.

Мотивом создания ограниченных копий обычно бывает повышение быстродействия программы. Однако создание нескольких копий крупных объектов редко приводит к заметному снижению быстродействия, хотя и выглядит эстетически непривлекательно. Полное копирование некоторых объектов действительно может снижать быстродействие, но программисты обычно очень плохо определяют, какой код вызывает проблемы (см. главу 25). Повышение сложности едва ли можно оправдать сомнительным улучшением быстродействия кода, поэтому, если не доказано обратное, лучше выполнять полное копирование.

Полные копии легче в реализации и сопровождении, чем ограниченные. При ограниченном копировании нужно написать не только специфический для объекта код, но и код подсчета ссылок, безопасного сравнения объектов, их безопасного уничтожения и т. д. Такой код может быть источником ошибок, поэтому без веской причины создавать его не следует.

Если вы находите, что вам все-таки нужно ограниченное копирование, прекрасное обсуждение этого подхода в контексте C++ см. в разделе 29 книги Скотта Мейерса «More Effective C++» (Meyers, 1996). В книге Мартина Фаулера «Refactoring» (Fowler, 1999) описываются специфические действия, нужные для преобразования ограниченных копий в полные и наоборот [Фаулер называет их объектами-ссылками (reference objects) и объектами-значениями (value objects)].

## 6.4. Разумные причины создания классов

**Перекрестная ссылка** Причины создания классов и методов во многом перекрываются (см. раздел 7.1).

Если вы верите всему, что читаете, у вас могло сложиться впечатление, что единственная причина создания класса — моделирование объектов реального мира. На самом деле это весьма далеко от истины. Список разумных причин создания класса приведен ниже.

**Перекрестная ссылка** Об идентификации объектов реального мира см. подраздел «Определите объекты реального мира» раздела 5.3.

***Моделирование объектов реального мира*** Пусть моделирование объектов реального мира — не единственная причина создания класса, но от этого она не становится менее хорошей! Создайте класс для каждого объекта реаль-

ного мира, моделируемого вашей программой. Поместите нужные объекту данные в класс и создайте сервисные методы, моделирующие поведение объекта. Примеры подобного моделирования см. в разделе 6.1.

***Моделирование абстрактных объектов*** Другой разумной причиной создания класса является моделирование *абстрактного объекта* — объекта, который не существует в реальном мире, но является абстракцией других конкретных объектов. Прекрасный пример — классический объект *Shape* (фигура). Объекты *Circle* (окружность) и *Square* (прямоугольник) существуют на самом деле, тогда как класс *Shape* — это абстракция конкретных фигур.

В мире программирования редко встречаются готовые абстракции вроде *Shape*, из-за чего поиск ясных абстракций усложняется. Процесс извлечения абстракций из разнообразия сущностей реального мира недетерминирован, и формирование абстракций может быть основано на разных принципах. Если бы нам не были известны окружности, прямоугольники, треугольники и другие геометрические фигуры, мы могли бы прийти в итоге к более необычным фигурам, таким как «кабачок», «брюква» и «Понтиак Ацтек». Нахождение адекватных абстрактных объектов — одна из главных проблем объектно-ориентированного проектирования.

***Снижение сложности*** Снижение сложности — самая важная причина создания класса. Создайте класс для сокрытия информации, чтобы о ней можно было не думать. Конечно, вам придется думать о ней при написании класса, но после этого вы сможете забыть о деталях и использовать класс, не зная о его внутренней работе. Другие причины создания классов — минимизация объема кода, облегчение сопровождения программы и снижение числа ошибок — тоже хороши, но без абстрагирующей силы классов сложные программы было бы невозможно охватить умом.

***Изоляция сложности*** Как бы ни проявлялась сложность — в форме запутанных алгоритмов, крупных наборов данных, замысловатых протоколов коммуникации, — она является источником ошибок. При возникновении ошибки ее будет проще найти, если она будет локализована в классе, а не распределена по всему коду. Изменения, обусловленные исправлением ошибки, не повлияют на остальной код, потому что вам придется исправить только один класс. Если вы найдете более эффективный, простой или надежный алгоритм, им будет легче заменить старый алгоритм, изолированный в классе. Во время разработки вам будет проще попробовать несколько вариантов проектирования и выбрать наилучший.

***Сокрытие деталей реализации*** Еще одна прекрасная причина создания класса — сокрытие деталей реализации, и таких сложных, как мудреный способ доступа к БД, и столь банальных, как отдельный элемент данных, хранимый в форме числа или строки.

***Ограничение влияния изменений*** Изолируйте области вероятных изменений, чтобы влияние изменений ограничивалось пределами одного или нескольких классов. Проектируйте приложение так, чтобы области вероятных изменений можно было изменить с максимальной легкостью. В число областей вероятных изменений входят зависимости от оборудования, подсистема ввода/вывода, сложные типы данных и бизнес-правила. Некоторые частые источники изменений

описаны в подразделе «Скрывайте секреты (к вопросу о сокрытии информации)» раздела 5.3.

**Перекрестная ссылка** О проблемах, связанных с глобальными данными, см. раздел 13.3.

***Сокрытие глобальных данных*** Используя глобальные данные, вы можете скрыть детали их реализации за интерфейсом класса. Обращение к глобальным данным через методы доступа имеет ряд преимуществ в сравнении с их непосредственным использованием. Вы можете менять структуру данных, не изменяя программу. Вы можете следить за доступом к данным. Кроме того, использование методов доступа подталкивает к размышлениям о том, на самом ли деле данные глобальны; часто оказывается, что «глобальные данные» на самом деле являются просто данными какого-то объекта.

***Упрощение передачи параметров в методы*** Если вы передаете один параметр в несколько методов, это может указывать на необходимость объединения этих методов в класс, чтобы они могли использовать параметр как данные объекта. Упрощение передачи параметров в методы само по себе не цель, но передача крупных объемов данных наводит на мысль, что другая организация классов могла бы быть более эффективной.

**Перекрестная ссылка** О сокрытии информации см. подраздел «Скрывайте секреты (к вопросу о сокрытии информации)» раздела 5.3.

***Создание центральных точек управления*** Управлять каждой задачей в одном месте — разумная идея. Управление может принимать разные формы. Знание числа элементов таблицы — одна форма. Управление файлами, соединениями с БД, принтерами и другими устройствами — другая. Использование одного класса для чтения и записи БД является формой централизованного управления. Если БД нужно будет преобразовать в однородный файл или данные «в памяти», изменения придется внести только в один класс.

Идея централизованного управления похожа на сокрытие информации, но она имеет уникальную эвристическую силу, так что не забудьте добавить ее в свой инструментарий.

***Облегчение повторного использования кода*** Код, разбитый на грамотно организованные классы, легче повторно использовать в других программах, чем тот же код, помещенный в один более крупный класс. Даже если фрагмент вызывается только из одного места программы и вполне понятен в составе более крупного класса, подумайте, может ли он понадобиться в другой программе. Если да, стоит поместить его в отдельный класс.

В Лаборатории проектирования ПО NASA были изучены десять проектов, в которых энергично преследовалось повторное использование кода (McGarry, Waligora, and McDermott, 1989). И при объектно-, и при функционально-ориентированном подходах разработчикам первоначально не удалось достичь этой цели, потому что в предыдущих проектах не была создана достаточная база кода. Впоследствии при работе над функциональными проектами около 35% кода удалось взять из предыдущих проектов. В проектах, основанных на объектно-ориентированном подходе, этот показатель составил 70%. Если благодаря заблаговременному планированию можно избежать написания 70% кода, грех этим не воспользоваться!

Заметьте, что ядро подхода NASA к созданию повторно используемых классов не включает «проектирование для повторного использования». Классы, претендующие на повторное использование, определяют в NASA в конце проектов. Все действия по упрощению повторного использования классов выполняются как специальный проект в конце основного проекта или как первый этап нового проекта. Этот подход помогает предотвращать «позолоту» — создание ненужной функциональности, только повышающей сложность.

**Перекрестная ссылка** О реализации минимального объема необходимой функциональности см. подраздел «Программа содержит код, который может когда-нибудь понадобиться» раздела 24.2.

***Планирование создания семейства программ*** Если вы ожидаете, что программу придется изменять, разумно изолировать области предполагаемых изменений в отдельных классах. После этого вы можете изменять классы, не влияя на остальную часть программы, или вообще заменить их на абсолютно новые классы. Размышление о том, как может выглядеть целое семейство программ, а не просто одна программа, — эффективный эвристический принцип предвосхищения целых категорий изменений (Parnas, 1976).

Как-то я руководил группой, работавшей над рядом программ, упрощавших заключение договоров страхования. Мы должны были адаптировать каждую программу к отдельным тарифам, формату отчетов конкретного клиента и т. д. Однако многие части программ были похожи: например, классы ввода данных о потенциальных заказчиках, классы, сохранявшие информацию в БД, классы просмотра тарифов и т. д. Мы организовали программу так, чтобы каждая «изменчивая» часть находилась в отдельном классе. Создание первоначальной программы заняло три месяца или около того, но зато когда к нам обращался новый клиент, мы просто переписывали несколько классов и включали их в остальной код. Несколько дней работы и — вуаля! — специализированное приложение!

***Упаковка родственных операций*** Если создание класса не удается обосновать сокрытием информации, совместным доступом к данным или обеспечением гибкости программы, вы все же можете упаковать наборы операций в более осмысленные группы, такие как группы тригонометрических функций, статистических функций, методов работы со строками, методов манипулирования битами, графических методов и т. д. Класс — не единственное средство объединения родственных операций. В зависимости от конкретного языка для этого также можно использовать пакеты, пространства имен или заголовочные файлы.

***Выполнение специфического вида рефакторинга*** Создание новых классов предусматривают многие специфические виды рефакторинга (см. главу 24), такие как разделение одного класса на два, сокрытие делегата, удаление класса-посредника и формирование класса-расширения. Создание этих новых классов может быть мотивировано стремлением к лучшему выполнению какой-либо задачи из описанных в данном разделе.

## Классы, которых следует избегать

Хотя в целом классы очень полезны, работая с ними, вы можете столкнуться с проблемами. Ниже описаны классы, создавать которые не следует.

***Избегайте создания «божественных» классов*** Избегайте создания классов, которые все знают и все могут. Если класс извлекает и задает данные других классов с использованием методов *Get()* и *Set()* (т. е. вмешивается в их дела и указывает им, что делать), спросите себя, не следует ли его функциональность реализовать в тех классах, а не выделять в божественный класс (Riel, 1996).

**Перекрестная ссылка** Такой вид класса обычно называют структурой. О структурах см. раздел 13.1.

***Устраняйте нерелевантные классы*** Если класс имеет только данные, но не формы поведения, спросите себя, действительно ли это класс. Возможно, этот класс следует разжаловать, сделав его данные-члены атрибутами одного или нескольких других классов.

***Избегайте классов, имена которых напоминают глаголы*** Как правило, класс, имеющий только формы поведения, но не данные, на самом деле классом не является. Подумайте о превращении класса вроде *DatabaseInitialization()* или *StringBuilder()* в метод какого-нибудь другого класса.

### Резюме причин создания класса

Вот список разумных причин создания класса:

- моделирование объектов реального мира;
- моделирование абстрактных объектов;
- снижение сложности;
- изоляция сложности;
- сокрытие деталей реализации;
- ограничение влияния изменений;
- сокрытие глобальных данных;
- упрощение передачи параметров в методы;
- создание центральных точек управления;
- облегчение повторного использования кода;
- планирование создания семейства программ;
- упаковка родственных операцией;
- выполнение специфического вида рефакторинга.

## 6.5. Аспекты, специфические для языков

Использование классов в разных языках программирования имеет интересные различия. Рассмотрим, например, переопределение метода-члена в производном классе при реализации полиморфизма. В Java все методы переопределяемы по умолчанию, а чтобы в производном классе метод нельзя было переопределить, его нужно объявить как *final*. В C++ методы по умолчанию непереопределяемы. Чтобы сделать метод переопределяемым, его нужно объявить в базовом классе как *virtual*. В Visual Basic переопределяемый метод должен быть объявлен в базовом классе как *overridable*, а в производном классе нужно использовать ключевое слово *overrides*.

Вот некоторые другие аспекты классов, во многом зависящие от языка:

- поведение переопределенных конструкторов и деструкторов в дереве наследования;
- поведение конструкторов и деструкторов при обработке исключений;
- важность конструкторов по умолчанию (конструкторов без аргументов);
- время вызова деструктора или метода финализации;
- целесообразность переопределения встроенных операторов языка, в том числе операторов присваивания и сравнения;
- управление памятью при создании и уничтожении объектов или при их объявлении и выходе из области видимости.

Подробно эти вопросы мы рассматривать не будем, но в разделе «Дополнительные ресурсы» я указал несколько хороших книг, посвященных конкретным языкам.

## 6.6. Следующий уровень: пакеты классов

**Перекрестная ссылка** О различии между классами и пакетами см. также подраздел «Уровни проектирования» раздела 5.2.

В настоящее время использование классов — лучший способ достижения модульности. Однако модульность — обширная тема, и она никак не ограничивается классами. В последние десятилетия отрасль разработки ПО развивалась во многом благодаря увеличению агрегаций, с которыми нам приходится работать. Первой агрегацией были операторы, что при сравнении с машинными командами казалось в то время большим достижением. Затем появились методы, а позднее придуманы классы.

Ясно, что мы могли бы лучше выполнять абстракцию и инкапсуляцию, если бы имели эффективные средства агрегации групп объектов. Java поддерживает пакеты, а язык Ada поддерживал их уже десять лет назад. Если используемый вами язык не поддерживает пакеты непосредственно, вы можете создать собственные версии пакетов, подкрепив их стандартами программирования, такими как:

- конвенции именования, проводящие различие между классами, которые можно применять вне пакета, и классами, предназначенными только для закрытого использования;
- конвенции именования, конвенции организации кода (структура проекта) или и те, и другие конвенции, определяющие принадлежность каждого класса к тому или иному пакету;
- правила, определяющие возможность использования конкретных пакетов другими пакетами, в том числе возможность наследования, включения или того и другого.

Это еще один удачный пример различия между программированием *на* языке и программированием *с использованием* языка (см. раздел 34.4).

### Контрольный список: качество классов

http://cc2e.com/0672

**Перекрестная ссылка** Этот контрольный список позволяет определить качество классов. Об этапах создания класса см. контрольный список «Процесс программирования с псевдокодом» в главе 9.

**Абстрактные типы данных**

- ❑ Обдумали ли вы классы программы как абстрактные типы данных, оценив их интерфейсы с этой точки зрения?

**Абстракция**

- ❑ Имеет ли класс главную цель?
- ❑ Удачное ли имя присвоено классу? Описывает ли оно главную цель класса?
- ❑ Формирует ли интерфейс класса согласованную абстракцию?
- ❑ Ясно ли интерфейс описывает использование класса?
- ❑ Достаточно ли абстрактен интерфейс, чтобы вы могли не думать о реализации класса? Можно ли рассматривать класс как «черный ящик»?
- ❑ Достаточно ли полон набор сервисов класса, чтобы другие классы могли не обращаться к его внутренним данным?
- ❑ Исключена ли из класса нерелевантная информация?
- ❑ Обдумали ли вы разделение класса на классы-компоненты? Разделен ли он на максимально возможное число компонентов?
- ❑ Сохраняется ли целостность интерфейса при изменении класса?

**Инкапсуляция**

- ❑ Сделаны ли члены класса минимально доступными?
- ❑ Избегает ли класс предоставления доступа к своим данным-членам?
- ❑ Скрывает ли класс детали реализации от других классов в максимально возможной степени, допускаемой языком программирования?
- ❑ Избегает ли класс предположений о своих клиентах, в том числе о производных классах?
- ❑ Независим ли класс от других классов? Слабо ли он связан?

**Наследование**

- ❑ Используется ли наследование только для моделирования отношения «является», т. е. придерживаются ли производные классы принципа подстановки Лисков?
- ❑ Описана ли в документации класса стратегия наследования?
- ❑ Избегают ли производные классы «переопределения» непереопределяемых методов?
- ❑ Перемещены ли общие интерфейсы, данные и формы поведения как можно ближе к корню дерева наследования?
- ❑ Не слишком ли много уровней включают иерархии наследования?
- ❑ Все ли данные — члены базового класса сделаны закрытыми, а не защищенными?

**Другие вопросы реализации**

- ❑ Класс содержит около семи элементов данных-членов или меньше?
- ❑ Минимально ли число встречающихся в классе непосредственных и опосредованных вызовов методов других классов?
- ❑ Сведено ли к минимуму сотрудничество класса с другими классами?
- ❑ Все ли данные-члены инициализируются в конструкторе?

- ❑ Спроектирован ли класс для использования полного, а не ограниченного копирования, если нет убедительной причины создания ограниченных копий?

**Аспекты, специфические для языков**

- ❑ Изучили ли вы особенности работы с классами, характерные для выбранного языка программирования?

# Дополнительные ресурсы

### Классы в общем

http://cc2e.com/0679

Meyer, Bertrand. *Object-Oriented Software Construction*, 2d ed. — New York, NY: Prentice Hall PTR, 1997. В этой книге Мейер рассматривает абстрактные типы данных и объясняет, как они формируют основу классов. В главах 14–16 подробно обсуждается наследование. В главе 15 Мейер приводит довод в пользу множественного наследования.

Riel, Arthur J. *Object-Oriented Design Heuristics*. — Reading, MA: Addison-Wesley, 1996. Эта книга включает множество советов по улучшению проектирования, относящихся большей частью к уровню классов. Я избегал ее несколько лет, потому что она казалась слишком большой — воистину сапожник без сапог! Однако основная часть книги занимает только около 200 страниц. Книга написана доступным и занимательным языком, а ее содержание сжато и практично.

### C++

http://cc2e.com/0686

Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*, 2d ed. — Reading, MA: Addison-Wesley, 1998.

Meyers, Scott. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. — Reading, MA: Addison-Wesley, 1996. Обе книги Мейерса являются каноническими для программистов на C++. Они очень интересны и позволяют приобрести глубокие знания некоторых нюансов C++.

### Java

http://cc2e.com/0693

Bloch, Joshua. *Effective Java Programming Language Guide*. — Boston, MA: Addison-Wesley, 2001. В книге Блоха можно найти много полезных советов по Java, а также описания более общих объектно-ориентированных подходов.

### Visual Basic

http://cc2e.com/0600

Ниже указаны книги, в которых хорошо рассмотрена работа с классами в контексте Visual Basic.

Foxall, James. *Practical Standards for Microsoft Visual Basic .NET*. — Redmond, WA: Microsoft Press, 2003.

Cornell, Gary, and Jonathan Morrison. *Programming VB .NET: A Guide for Experienced Programmers*. — Berkeley, CA: Apress, 2002.

Barwell, Fred, et al. *Professional VB.NET*, 2d ed. — Wrox, 2002.

## Ключевые моменты

- Интерфейс класса должен формировать согласованную абстракцию. Многие проблемы объясняются нарушением одного этого принципа.
- Интерфейс класса должен что-то скрывать — особенности взаимодействия с системой, аспекты проектирования или детали реализации.
- Включение обычно предпочтительнее, чем наследование, если только вы не моделируете отношение «является».
- Наследование — полезный инструмент, но оно повышает сложность, что противоречит Главному Техническому Императиву Разработки ПО, которым является управление сложностью.
- Классы — главное средство управления сложностью. Уделите их проектированию столько времени, сколько нужно для достижения этой цели.

ГЛАВА 7

# Высококачественные методы

**Содержание**



http://cc2e.com/0778

**Связанные темы**

- Этапы конструирования методов: раздел 9.3
- Классы: глава 6
- Общие методики проектирования: глава 5
- Архитектура ПО: раздел 3.5

В главе 6 мы подробно рассмотрели создание классов. В этой главе мы обратим внимание на методы и характеристики, отличающие хорошие методы от плохих. Если вам хотелось бы сначала разобраться в вопросах, влияющих на проектирование методов, прочитайте главу 5 и потом вернитесь к этой главе. Некоторые важные атрибуты высококачественных методов обсуждаются также в главе 8. Если вас больше интересуют этапы создания методов и классов, см. главу 9.

Прежде чем перейти к деталям, определим два базовых термина. Что такое «метод»? Метод — это отдельная функция или процедура, выполняющая одну задачу. В различных языках методы могут называться по-разному, но их суть от этого не меняется. Иногда макросы C и C++ также полезно рассматривать как методы. Многие советы по созданию высококачественных методов относятся и к макросам.

Что такое *высококачественный* метод? На этот вопрос ответить сложнее. Возможно, лучше всего просто показать, что *не* является высококачественным методом. Вот пример низкокачественного метода:

### Пример низкокачественного метода (C++)

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec,
   double & estimRevenue, double ytdRevenue, int screenX, int screenY,
   COLOR_TYPE & newColor, COLOR_TYPE & prevColor, StatusType & status,
   int expenseType )
{
int i;
for ( i = 0; i < 100; i++ ) {
   inputRec.revenue[i] = 0;
   inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
   }
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
if ( expenseType == 1 ) {
     for ( i = 0; i < 12; i++ )
           profit[i] = revenue[i] - expense.type1[i];
     }
else if ( expenseType == 2 )  {
          profit[i] = revenue[i] - expense.type2[i];
          }
else if ( expenseType == 3 )
          profit[i] = revenue[i] - expense.type3[i];
          }
```

Что тут не так? Подскажу: вы должны найти минимум 10 недостатков. Составив свой список, сравните его с моим.

- Неудачное имя: *HandleStuff()* ничего не говорит о роли метода.
- Метод недокументирован (вопрос документирования не ограничивается отдельными методами и обсуждается в главе 32).
- Метод плохо форматирован. Физическая организация кода почти не дает представления о его логической организации. Стратегии форматирования используются непоследовательно: сравните стили операторов *if* с условиями *expenseType == 2* и *expenseType == 3* (о форматировании см. главу 31).
- Входная переменная *inputRec* внутри метода изменяется. Если это входная переменная, изменять ее нежелательно (в случае C++ ее следовало бы объявить как *const*). Если изменение значения предусмотрено, переменную не стоило называть *inputRec*.
- Метод читает и изменяет глобальные переменные: читает *corpExpense* и изменяет *profit*. Взаимодействие этого метода с другими следовало бы сделать более непосредственным, без использования глобальных переменных.
- Цель метода размыта. Он инициализирует ряд переменных, записывает данные в БД, выполняет вычисления — все эти действия не кажутся связанными между собой. Метод должен иметь одну четко определенную цель.

- Метод не защищен от получения плохих данных. Если переменная *crntQtr* равна 0, выражение *ytdRevenue * 4.0 / (double) crntQtr* вызывает ошибку деления на 0.
- Метод использует несколько «магических» чисел: 100, 4.0, 12, 2 и 3 (о магических числах см. раздел 12.1).
- Параметры *screenX* и *screenY* внутри метода не используются.
- Параметр *prevColor* передается в метод неверно: он передается по ссылке (&), но значение ему внутри метода не присваивается.
- Метод принимает слишком много параметров. Как правило, чтобы параметры можно было охватить умом, их должно быть не более 7 — этот метод принимает 11. Параметры представлены таким неудобочитаемым образом, что большинство разработчиков даже не попытаются внимательно изучить их или хотя бы подсчитать.
- Параметры метода плохо упорядочены и не документированы (об упорядочении параметров см. эту главу, о документировании — главу 32).

Если не считать сами компьютеры, методы — величайшее изобретение в области компьютерных наук. Методы облегчают чтение и понимание программ в большей степени, чем любая другая возможность любого языка программирования, и оскорблять столь заслуженных в мире программирования деятелей таким кодом, что был приведен выше, — настоящее преступление.

http://cc2e.com/0799

**Перекрестная ссылка** Классы также претендуют на роль величайшего изобретения в области информатики. Об эффективном использовании классов см. главу 6.

Кроме того, методы — самый эффективный способ уменьшения объема и повышения быстродействия программ. Представьте, насколько объемнее были бы ваши программы, если б вместо каждого вызова метода нужно было вставить соответствующий код. Представьте, насколько сложнее было бы оптимизировать код, если бы он был распространен по всей программе, а не локализован в одном методе. Программирование, каким мы его знаем сегодня, оказалось бы без методов невозможным.

«Хорошо, — скажете вы. — Я уже знаю, что методы очень полезны и постоянно их использую. Чего ж вы от меня хотите?»

Я хочу, чтобы вы поняли, что есть много веских причин, а также правильных и неправильных способов создания методов. Будучи студентом факультета информатики, я думал, что главная причина создания методов — предотвращение дублирования кода. Во вводном учебнике, по которому я учился, полезность методов обосновывалась тем, что предотвращение дублирования кода делает программу более простой в разработке, отладке, документировании и сопровождении. Точка. Если не считать синтаксические детали использования параметров и локальных переменных, на этом обсуждение методов в той книге заканчивалось. Такое объяснение теории и практики использования методов нельзя считать ни удачным, ни полным. В следующих разделах я постараюсь это исправить.

# 7.1. Разумные причины создания методов

Ниже я привел список причин создания метода. Они несколько перекрываются и не исключают одна другую.

***Снижение сложности*** Самая важная причина создания метода — снижение сложности программы. Создайте метод для сокрытия информации, чтобы о ней можно было не думать. Конечно, при написании метода думать о ней придется, но после этого вы сможете забыть о деталях и использовать метод, не зная о его внутренней работе. Другие причины создания методов — минимизация объема кода, облегчение сопровождения программы и снижение числа ошибок — также хороши, но без абстрагирующей силы методов сложные программы было бы невозможно охватить умом.

Одним из признаков того, что метод следует разделить, является глубокая вложенность внутренних циклов или условных операторов. Упростите такой метод, выделив вложенную часть в отдельный метод.

***Формирование понятной промежуточной абстракции*** Выделение фрагмента кода в удачно названный метод — один из лучших способов документирования его цели. Вместо того, чтобы работать с фрагментами вида:

```
if ( node <> NULL ) then
   while ( node.next <> NULL ) do
      node = node.next
      leafName = node.name
   end while
else
   leafName = ""
end if
```

вы можете иметь дело с чем-нибудь вроде:

```
leafName = GetLeafName( node )
```

Новый метод так прост, что для документирования достаточно присвоить ему удачное имя. В сравнении с первоначальными восемью строками кода имя метода формирует абстракцию более высокого уровня, что облегчает чтение и понимание кода, а также снижает его сложность.

***Предотвращение дублирования кода*** Несомненно, самая популярная причина создания метода — желание избежать дублирования кода. Действительно, включение похожего кода в два метода указывает на ошибку декомпозиции. Уберите повторяющийся фрагмент из обоих методов, поместите его общую версию в базовый класс и создайте два специализированных метода в подклассах. Вы также можете выделить общий код в отдельный метод и вызвать его из двух первоначальных методов. В результате программа станет компактнее. Изменять ее станет проще, так как в случае чего вам нужно будет изменить только один метод. Код станет надежнее, потому что для его проверки нужно будет проанализировать только один фрагмент. Изменения будут реже приводить к ошибкам, поскольку вы не сможете по невнимательности внести в идентичные фрагменты программы чуть различающиеся изменения.

***Поддержка наследования*** Переопределить небольшой грамотно организованный метод легче, чем длинный и плохо спроектированный. Кроме того, стремление к простоте переопределяемых методов уменьшает вероятность ошибок при реализации подклассов.

***Сокрытие очередности действий*** Скрывать очередность обработки событий — разумная идея. Например, если программа обычно сначала вызывает метод, запрашивающий информацию у пользователя, а после этого — метод, читающий вспомогательные данные из файла, никакой из этих двух методов не должен зависеть от порядка их выполнения. В качестве другого примера можно привести две строки кода, первая из которых читает верхний элемент стека, а вторая уменьшает переменную *stackTop*. Вместо того чтобы распространять такой код по всей системе, скройте предположение о необходимом порядке выполнения двух операций, поместив две эти строки в метод *PopStack()*.

***Сокрытие операций над указателями*** Операции над указателями не отличаются удобочитаемостью и часто являются источником ошибок. Изолировав такие операции в методах, вы сможете сосредоточиться на их сути, а не на механизме манипуляций над указателями. Кроме того, выполнение операций над указателями в одном месте облегчает проверку правильности кода. Если же вы найдете более эффективный тип данных, чем указатели, изменения затронут лишь несколько методов.

***Улучшение портируемости*** Использование методов изолирует непортируемый код, явно определяя фрагменты, которые придется изменить при портировании приложения. В число непортируемых аспектов входят нестандартные возможности языка, зависимости от оборудования и операционной системы и т. д.

***Упрощение сложных булевых проверок*** Понимание сложных булевых проверок редко требуется для понимания пути выполнения программы. Поместив такую проверку в метод, вы сможете упростить код, потому что (1) детали проверки будут скрыты и (2) описательное имя метода позволит лучше охарактеризовать суть проверки.

Создание отдельного метода для проверки подчеркивает ее значимость. Это мотивирует программистов сделать детали проверки внутри метода более удобочитаемыми. В результате и основной путь выполнения кода, и сама проверка становятся более понятными. Упрощение булевых проверок является примером снижения сложности, которого мы уже не раз касались.

***Повышение быстродействия*** Методы позволяют выполнять оптимизацию кода в одном месте, а не в нескольких. Они облегчают профилирование кода, направленное на определение неэффективных фрагментов. Если код централизован в методе, его оптимизация повысит быстродействие всех фрагментов, в которых этот метод вызывается как непосредственно, так и косвенно, а реализация метода на более эффективном языке или с применением улучшенного алгоритма окажется более выгодной.

***Для уменьшения объема других методов?*** Нет. При наличии стольких разумных причин создания методов эта не нужна. На самом деле для решения некоторых задач лучше использовать один крупный метод (об оптимальном размере метода см. раздел 7.4).

**Перекрестная ссылка** О сокрытии информации см. подраздел «Скрывайте секреты (к вопросу о сокрытии информации)» раздела 5.3.

## Операция кажется слишком простой, чтобы создавать для нее метод

Один из главных ментальных барьеров, препятствующих созданию эффективных методов, — нежелание создавать простой метод для простой цели. Создание метода для двух или трех строк кода может показаться пальбой из пушки по воробьям, но опыт свидетельствует о том, что небольшие методы могут быть чрезвычайно полезны.

Небольшие методы обеспечивают несколько преимуществ, и одно из них — облегчение чтения кода. Так, однажды я обнаружил следующую строку примерно в десятке мест программы:

**Пример вычисления (псевдокод)**

```
points = deviceUnits * ( POINTS_PER_INCH / DeviceUnitsPerInch() )
```

Наверняка это не самая сложная строка кода в вашей жизни. Большинство людей в итоге поняло бы, что она преобразует некоторую величину, выраженную в аппаратных единицах, в соответствующее число точек, а кроме того, что каждая из десятка строк делает одно и то же. Однако эти фрагменты можно было сделать еще более ясными, поэтому я создал метод с выразительным именем, выполняющий преобразование в одном месте:

**Пример вычисления, преобразованного в функцию (псевдокод)**

```
Function DeviceUnitsToPoints  ( deviceUnits Integer ): Integer
   DeviceUnitsToPoints = deviceUnits *
      ( POINTS_PER_INCH / DeviceUnitsPerInch() )
End Function
```

В результате все десять первоначальных фрагментов стали выглядеть примерно так:

**Пример вызова функции (псевдокод)**

```
points = DeviceUnitsToPoints( deviceUnits )
```

Эта строка более понятна и даже кажется очевидной.

Данный пример позволяет назвать еще одну причину создания отдельных методов для простых операций: дело в том, что простые операции имеют свойство усложняться с течением времени. После того как я написал метод *DeviceUnitsPerInch()*, оказалось, что в определенных условиях при активности определенных устройств он возвращает 0. Для предотвращения деления на 0 мне пришлось написать еще три строки кода:

**Пример кода, расширяющегося при сопровождении программы (псевдокод)**

```
Function DeviceUnitsToPoints( deviceUnits: Integer ) Integer;
   if ( DeviceUnitsPerInch() <> 0 )
      DeviceUnitsToPoints = deviceUnits *
         ( POINTS_PER_INCH / DeviceUnitsPerInch() )
```

```
   else
      DeviceUnitsToPoints = 0
   end if
End Function
```

Если бы в коде по-прежнему использовалась первоначальная строка, мне пришлось бы повторить проверку десять раз, добавив в общей сложности 30 строк кода. Создание простого метода позволило уменьшить это число до 3.

### Резюме причин создания методов

Вот список разумных причин создания методов:

- снижение сложности;
- формирование понятной промежуточной абстракции;
- предотвращение дублирования кода;
- поддержка наследования;
- сокрытие очередности действий;
- сокрытие операций над указателями;
- улучшение портируемости;
- упрощение сложных булевых проверок;
- повышение быстродействия.

Кроме того, разумными причинами создания методов можно считать многие из причин создания классов:

- изоляция сложности;
- сокрытие деталей реализации;
- ограничение влияния изменений;
- сокрытие глобальных данных;
- создание центральных точек управления;
- облегчение повторного использования кода;
- выполнение специфического вида рефакторинга.

## 7.2. Проектирование на уровне методов

Идею связности впервые представили Уэйн Стивенс, Гленфорд Майерс и Ларри Константайн (Stevens, Myers, and Constantine, 1974). На уровне проектирования классов ее практически вытеснили более современные концепции, такие как абстракция и инкапсуляция, однако на уровне проектирования отдельных методов эвристический принцип связности по-прежнему полезен.

В случае методов связность характеризует соответствие выполняемых в методе операций единой цели. Некоторые программисты предпочитают использовать термин «сила» (strength): насколько сильно связаны операции в методе? Например, метод *Cosine()* (косинус) имеет одну четко определенную цель и потому обладает прекрасной связностью. Метод *CosineAndTan()* (косинус и тангенс) имеет меньшую связность, потому что он выполняет сразу

**Перекрестная ссылка** О связности см. подраздел «Стремитесь к максимальной связности» раздела 5.3.

две функции. Наша цель в том, чтобы каждый метод эффективно решал одну задачу и больше ничего не делал.

Вознаграждением будет более высокая надежность кода. В одном исследовании 450 методов было обнаружено, что дефекты отсутствовали в 50% методов, обладающих высокой связностью, и только в 18% методов с низкой связностью (Card, Church, and Agresti, 1986). Другое исследование 450 методов (это просто совпадение, хотя и весьма необычное) показало, что в сравнении с методами, имеющими самое низкое отношение «сопряжение/связность» (coupling-to-cohesion), методы с максимальным отношением «сопряжение/связность» содержали в 7 раз больше ошибок, а исправление этих методов было в 20 раз более дорогим (Selby and Basili, 1991).

Обсуждение связности обычно касается нескольких ее уровней. Понять эти концепции важнее, чем запомнить специфические термины. Используйте концепции как средства, помогающие сделать методы максимально связными.

*Функциональная связность* — самый сильный и лучший вид связности; она имеет место, когда метод выполняет одну и только одну операцию. Примерами методов, обладающих высокой связностью, являются методы *sin()* (синус), *GetCustomerName()* (получить фамилию заказчика), *EraseFile()* (удалить файл), *CalculateLoanPayment()* (вычислить плату за кредит) и *AgeFromBirthdate()* (определить возраст по дате рождения). Конечно, такая оценка связности предполагает, что эти методы соответствуют своим именам — иначе они имеют неудачные имена, а об их связности нельзя сказать ничего определенного.

Ниже описаны другие виды связности, которые обычно считаются менее эффективными.

- *Последовательная связность* (sequential cohesion) наблюдается в том случае, когда метод содержит операции, которые обязательно выполняются в определенном порядке, используют данные предыдущих этапов и не формируют в целом единую функцию.

  Примером метода с последовательной связностью является метод, вычисляющий по дате рождения возраст сотрудника и срок до его ухода на пенсию. Если метод вычисляет возраст и затем использует этот результат для нахождения срока до ухода сотрудника на пенсию, он имеет последовательную связность. Если метод находит возраст сотрудника, после чего в абсолютно другом вычислении определяет срок до ухода на пенсию, применяя те же данные о дате рождения, он имеет только коммуникационную связность.

  Как сделать такой метод функционально связным? Создать два отдельных метода: метод, вычисляющий по дате рождения возраст сотрудника, и метод, определяющий по дате рождения срок до ухода сотрудника на пенсию. Второй метод мог бы вызывать метод нахождения возраста. Оба этих метода имели бы функциональную связность. Другие методы могли бы вызывать любой из них или оба.

- *Коммуникационная связность* (communicational cohesion) имеет место, когда выполняемые в методе операции используют одни и те же данные и не связаны между собой иным образом. Если метод печатает отчет, после чего заново инициализи-

рует переданные в него данные, он имеет коммуникационную связность: две операции объединяет только то, что они обращаются к одним и тем же данным.
Чтобы повысить связность этого метода, выполняйте повторную инициализацию данных около места их создания, которое не должно находиться в методе печати отчета. Разделите операции на два метода: первый будет печатать отчет, а второй — выполнять повторную инициализацию данных неподалеку от кода, создающего или изменяющего данные. Вызовите оба этих метода вместо первоначального метода, имевшего коммуникационную связность.

- *Временная связность* (temporal cohesion) наблюдается, когда операции объединяются в метод на том основании, что все они выполняются в один интервал времени. Типичные примеры — методы *Startup()* (запуск программы) *CompleteNewEmployee()* (прием нового сотрудника на работу) и *Shutdown()* (завершение программы). Временную связность порой считают неприемлемой, поскольку иногда она связана с плохими методиками программирования, такими как включение слишком разнообразного кода в метод *Startup()*.
  Для устранения этой проблемы рассматривайте методы с временной связностью как способы организации других событий. Так, метод *Startup()* мог бы читать конфигурационный файл, инициализировать вспомогательный файл, настраивать менеджер памяти и выводить первоначальное окно программы. Чтобы сделать метод с временной связностью максимально эффективным, не выполняйте в нем конкретных операций непосредственно, а вызывайте для их выполнения другие методы. Тогда всем будет ясно, что суть метода — согласование действий, а не их выполнение.
  Этот пример поднимает вопрос выбора имени, описывающего такой метод с адекватным уровнем абстракции. Вы могли бы назвать метод *ReadConfigFileInitScratchFileEtc()* (прочитать конфигурационный файл, инициализировать вспомогательный файл и т. д.), но из этого следовало бы, что он имеет только случайную связность. Если же вы назовете метод *Startup()*, будет очевидно, что он имеет одну цель и поэтому обладает функциональной связностью.

Остальные виды связности обычно неприемлемы. Они приводят к созданию плохо организованного кода, который трудно отлаживать и изменять. Метод с плохой связностью лучше переписать, чем тратить время и средства на поиск проблем. Однако знание того, чего следует избегать, может пригодиться, поэтому ниже я привел описания плохих видов связности.

- *Процедурная связность* (procedural cohesion) имеет место, когда операции в методе выполняются в определенном порядке. В качестве примера можно привести метод, получающий фамилию сотрудника, затем его адрес, а после этого номер телефона. Порядок этих операций важен только потому, что он соответствует порядку, в котором пользователя просят ввести данные. Остальные данные о сотруднике получает другой метод. В данном случае операции выполняются в определенном порядке и не объединены больше ничем, поэтому метод имеет процедурную связность.
  Для достижения лучшей связности поместите разные операции в отдельные методы. Сделайте так, чтобы вызывающий метод решал одну задачу, причем полностью: пусть он соответствует имени *GetEmployee()* (получить данные о со-

труднике), а не *GetFirstPartOfEmployeeData()* (получить первую часть данных о сотруднике). Вероятно, при этом придется изменить и методы, получающие остальные данные. Довольно часто достижение функциональной связности требует изменения двух или более первоначальных методов.

- *Логическая связность* (logical cohesion) имеет место, когда метод включает несколько операций, а выбор выполняемой операции осуществляется на основе передаваемого в метод управляющего флага. Этот вид связности называется логическим потому, что операции метода объединены только управляющей «логикой» метода: крупным оператором *if* или рядом блоков *case*. Какой-нибудь другой по-настоящему «логической» связи между операциями нет. Поскольку определяющим атрибутом логической связности является отсутствие отношений между операциями, возможно, лучше было бы назвать ее «нелогичной связностью».

  В качестве примера такого метода можно привести метод *InputAll()*, принимающий в зависимости от полученного флага фамилии клиентов, данные карт учета времени сотрудников или инвентаризационные данные. Другие примеры — методы *ComputeAll()*, *EditAll()*, *PrintAll()* и *SaveAll()*. Главная проблема с ними в том, что передавать флаг для управления работой метода нецелесообразно. Вместо метода, выполняющего одну из трех операций в зависимости от полученного флага, лучше создать три метода, выполняющих по одной операции. Если операции используют некоторый одинаковый код или общие данные, код следует переместить в метод более низкого уровня, а методы упаковать в класс.

  Однако логически связный метод вполне приемлем, если его код состоит исключительно из ряда операторов *if* или *case* и вызовов других методов. Если единственная роль метода — координация выполнения команд и сам он не выполняет действий, это обычно удачное проектное решение. Такие методы еще называют «обработчиками событий». Обработчики часто используются в интерактивных средах, таких как Apple Macintosh, Microsoft Windows и других средах с GUI.

  **Перекрестная ссылка** Связность такого метода может быть удовлетворительной, однако при этом возникает один вопрос проектирования более высокого уровня: использовать ли операторы *case* вместо полиморфного метода? См. также подраздел «Замена условных операторов (особенно многочисленных блоков *case*) на вызов полиморфного метода» раздела 24.3.

- При *случайной связности* (coincidental cohesion) каких-либо ясных отношений между выполняемыми в методе операциями нет. Этот вариант можно еще называть «отсутствием связности» или «хаотичной связностью». Низкокачественный метод C++, приведенный в начале этой главы, имеет случайную связность. Случайную связность трудно преобразовать в более приемлемый вид связности — как правило, методы со случайной связностью нужно проектировать и реализовать заново.

Никакой из этих терминов не является магическим или священным. Изучайте идеи, а не терминологию. Стремитесь создавать методы с функциональной связностью — это возможно почти всегда.

## 7.3. Удачные имена методов

Имя метода должно ясно описывать все, что он делает. Советы по выбору удачных имен методов приведены ниже.

**Перекрестная ссылка** Об именовании переменных см. главу 11.

***Описывайте все, что метод выполняет*** Опишите в имени метода все выходные данные и все побочные эффекты. Если метод вычисляет сумму показателей в отчете и открывает выходной файл, имя *ComputeReportTotals()* не будет адекватным. *ComputeReportTotalsAndOpenOutputFile()* — имя адекватное, но слишком длинное и несуразное. Создавая методы с побочными эффектами, вы получите много длинных несуразных имен. Выход из этого положения — не использование менее описательных имен, а создание ясных методов без побочных эффектов.

***Избегайте невыразительных и неоднозначных глаголов*** Некоторые глаголы могут описывать практически любое действие. Имена вроде *HandleCalculation()*, *PerformServices()*, *OutputUser()*, *ProcessInput()* и *DealWithOutput()* не говорят о работе методов почти ничего. В лучшем случае по этим именам можно догадаться, что методы имеют какое-то отношение к вычислениям, сервисам, пользователям, вводу и выводу соответственно. Исключением было бы использование глагола «handle» в специфическом техническом смысле обработки события.

Иногда единственным недостатком метода является невыразительность его имени; сам метод при этом может быть спроектирован очень хорошо. Если имя *HandleOutput()* заменить на *FormatAndPrintOutput()*, роль метода станет очевидной.

В других случаях невыразительность глагола в имени метода может объясняться аналогичным поведением метода. Неясная цель — невыразительное имя. Если это так, лучше всего выполнить реструктуризацию метода и всех родственных методов, чтобы все они получили более четкие цели и более выразительные имена, точно их описывающие.

***Не используйте для дифференциации имен методов исключительно номера*** Один разработчик написал весь свой код в форме единственного объемного метода. Затем он разбил код на фрагменты по 15 строк и создал методы *Part1*, *Part2* и т. д. После этого он создал один высокоуровневый метод, вызывающий каждую часть кода. Подобный способ создания и именования методов глуп до невозможности (и столь же редок, надеюсь). И все же программисты иногда используют номера для дифференциации таких методов, как *OutputUser*, *OutputUser1* и *OutputUser2*. Номера в конце каждого из этих имен ничего не говорят о различиях представляемых методами абстракций, поэтому такие имена нельзя признать удачными.

***Не ограничивайте длину имен методов искусственными правилами*** Исследования показывают, что оптимальная длина имени переменной равняется в среднем 9–15 символам. Как правило, методы сложнее переменных, поэтому и адекватные имена методов обычно длиннее. В то же время к именам методов часто присоединяются имена объектов, что по сути предоставляет методам часть имени «бесплатно». Главной задачей имени метода следует считать как можно более ясное и понятное описание сути метода, поэтому имя может иметь любую длину, удовлетворяющую этой цели.

**Перекрестная ссылка** О различии между процедурами и функциями см. раздел 7.6.

***Для именования функции используйте описание возвращаемого значения*** Функция возвращает значение, и это следует должным образом отразить в ее имени. Так, имена *cos()*, *customerId.Next()*, *printer.IsReady()* и *pen.CurrentColor()* ясно указывают, что возвращают функции, и потому являются удачными.

***Для именования процедуры используйте выразительный глагол, дополняя его объектом*** Процедура с функциональной связностью обычно выполняет операцию над объектом. Имя должно отражать выполняемое процедурой действие и объект, над которым оно выполняется, что приводит нас к формату «глагол + объект». Примеры удачных имен процедур — *PrintDocument()*,*CalcMonthlyRevenues()*, *CheckOrderInfo()* и *RepaginateDocument()*.

В случае объектно-ориентированных языков имя объекта в имя процедуры включать не нужно, потому что объекты и так входят в состав вызовов, принимающих вид *document.Print()*, *orderInfo.Check()* и *monthlyRevenues.Calc()*. Имена вида *document.PrintDocument()* страдают от избыточности и могут стать в производных классах неверными. Если *Check* — класс, производный от класса *Document*, суть вызова *check.Print()* кажется очевидной: печать чека. В то же время вызов *check.PrintDocument()* похож на печать записи чековой книжки или ежемесячной выписки со счета, но никак не чека.

**Перекрестная ссылка** Похожий список антонимов, используемых в именах переменных, см. в подразделе «Антонимы, часто встречающиеся в именах переменных» раздела 11.1.

***Дисциплинированно используйте антонимы*** Применение конвенций именования, подразумевающих использование антонимов, поддерживает согласованность имен, что облегчает чтение кода. Антонимы вроде first/last понятны всем. Пары вроде FileOpen() и _lclose() несимметричны и вызывают замешательство. Вот некоторые антонимы, популярные в программировании:

```
add/remove          increment/decrement          open/close
begin/end           insert/delete                show/hide
create/destroy      lock/unlock                  source/target
first/last          min/max                      start/stop
get/put             next/previous                up/down
get/set             old/new
```

***Определяйте конвенции именования часто используемых операций*** При работе над некоторыми системами важно различать разные виды операций. Самым легким и надежным способом определения этих различий часто оказывается конвенция именования.

В одном из моих проектов каждый объект имел уникальный идентификатор. Мы не потрудились выработать конвенцию именования методов, возвращающих идентификатор объекта, и в итоге получили такие имена, как:

```
employee.id.Get()
dependent.GetId()
supervisor()
candidate.id()
```

Класс *Employee* предоставлял доступ к объекту *id*, который в свою очередь включал метод *Get()*. Класс *Dependent* предоставлял для этой цели метод *GetId()*. Разра-

ботчик класса *Supervisor* сделал *id* значением, возвращаемым по умолчанию. Класс *Candidate* предоставлял доступ к объекту *id*, который по умолчанию возвращал значение идентификатора. К середине проекта никто из нас уже не мог вспомнить, какой из методов предполагалось использовать для того или иного объекта, но мы уже написали слишком много кода, чтобы возвращаться назад и все согласовывать. Поэтому каждому члену группы пришлось тратить лишние усилия на запоминание несогласованных подробностей синтаксиса получения *id* из каждого класса. Конвенция именования, определяющая получение *id*, сделала бы такую неприятность невозможной.

## 7.4. Насколько объемным может быть метод?

На пути в Америку пилигримы[1] спорили о лучшей максимальной длине метода. И вот они прибыли к Плимутскому камню и начали составлять Мейфлауэрское соглашение. О максимальной длине методов пилигримы так и не договорились, а так как до подписания соглашения они не могли высадиться на берег, то сдались и не включили этот пункт в соглашение. Результатом стали нескончаемые дебаты о допустимой длине методов.

В теоретических работах длину метода часто советуют ограничивать числом строк, помещающихся на экране монитора, или же одной-двумя страницами, что соответствует примерно 50–150 строкам. Следуя этому правилу, в IBM однажды ограничили методы 50 строками, а в TRW — двумя страницами (McCabe, 1976). Современные программы обычно включают массу очень коротких методов, вызываемых из нескольких более крупных методов. Однако длинные методы далеки от вымирания. Незадолго до завершения работы над этой книгой я в течение месяца посетил двух клиентов. В одном случае программисты боролись с методом, включавшим примерно 4000 строк, а во втором пытались укротить метод, содержавший более 12 000 строк!

Длина методов уже давно стала предметом исследований. Некоторые из них устарели, а другие актуальны и по сей день.

- Базили и Перриконе обнаружили обратную корреляцию между размером метода и уровнем ошибок: при росте размера методов (вплоть до 200 строк) число ошибок в расчете на одну строку снижалось (Basili and Perricone, 1984).
- Другое исследование показало, что с числом ошибок коррелировали структурная сложность и объем используемых данных, но не размер метода (Shen et al., 1985).
- В исследовании 1986 г. было обнаружено, что небольшой размер методов (32 строки или менее) не коррелировал с меньшими затратами на их разработку или меньшим числом дефектов (Card, Church, and Agresti, 1986; Card and Glass,

[1] Пилигримы (pilgrims) — пассажиры английского судна «Мейфлауэр» («Mayflower»), основатели Плимутской колонии в Северной Америке, заключившие Мейфлауэрское соглашение (Mayflower Compact) о создании «гражданской политической организации» для поддержания порядка и безопасности, «принятия справедливых и обеспечивающих равноправие законов». Плимутский камень (Plymouth Rock) — по преданию гранитная глыба, на которую ступил первый сошедший с корабля пилигрим в декабре 1620 г. Почитается в США как национальная святыня. — *Прим. перев.*

1990). Разработка крупных методов (65 строк или более) в расчете на одну строку кода была дешевле.

- Опытное изучение 450 методов показало, что небольшие методы (включавшие менее 143 команд исходного кода с учетом комментариев) содержали на 23% больше ошибок в расчете на строку кода, чем более крупные методы, но исправление меньших методов было в 2,4 раза менее дорогим (Selby and Basili, 1991).
- Исследования позволили обнаружить, что код требовал минимальных изменений, если методы состояли в среднем из 100–150 строк (Lind and Vairavan, 1989).
- Исследование, проведенное в IBM, показало, что максимальный уровень ошибок был характерен для методов, размер которых превышал 500 строк кода. При дальнейшем увеличении методов уровень ошибок возрастал пропорционально числу строк (Jones, 1986a).

Так какую же длину методов считать приемлемой в объектно-ориентированных программах? Многие методы в объектно-ориентированных программах будут методами доступа, обычно очень короткими. Время от времени реализация сложного алгоритма будет требовать создания более длинного метода, и тогда методу можно будет позволить вырасти до 100–200 строк (строкой считается непустая строка исходного кода, не являющаяся комментарием). Десятилетия исследований говорят о том, что методы такой длины не более подвержены ошибкам, чем методы меньших размеров. Пусть длину метода определяют не искусственные ограничения, а такие факторы, как связность метода, глубина вложенности, число переменных, число точек принятия решений, число комментариев, необходимых для объяснения метода, и другие соображения, связанные со сложностью кода.

Что касается методов, включающих более 200 строк, то к ним следует относиться настороженно. Ни в одном из исследований, в которых было обнаружено, что более крупным методам соответствует меньшая стоимость разработки, меньший уровень ошибок или оба фактора, эта тенденция не усиливалась при увеличении размера свыше 200 строк, а при превышении этого предела методы неизбежно становятся менее понятными.

## 7.5. Советы по использованию параметров методов

Интерфейсы между методами — один из основных источников ошибок. В одном часто цитируемом исследовании, проведенном Базили и Перриконе (Basili and Perricone, 1984), было обнаружено, что 39% всех ошибок были ошибками внутренних интерфейсов — ошибками коммуникации между методами. Вот несколько советов по предотвращению подобных проблем.

**Перекрестная ссылка** О документировании параметров методов см. подраздел «Комментирование методов» раздела 32.5, а о форматировании параметров — раздел 31.7.

***Передавайте параметры в порядке «входные значения — изменяемые значения — выходные значения»*** Вместо упорядочения параметров случайным образом или по алфавиту указывайте в списке сначала исключительно входные параметры, затем входные-и-выходные параметры и наконец — исключительно выходные параметры. Такой по-

рядок соответствует последовательности выполняемых в методе операций: ввод данных, их изменение и возврат результата. Вот примеры списков параметров, написанные на языке Ada:

**Примеры размещения параметров в порядке «входные значения — изменяемые значения — выходные значения» (Ada)**

```
procedure InvertMatrix(
   originalMatrix: in Matrix;
   resultMatrix: out Matrix
);
...

procedure ChangeSentenceCase(
   desiredCase: in StringCase;
   sentence: in out Sentence
);
...

procedure PrintPageNumber(
   pageNumber: in Integer;
   status: out StatusType
);
```

В языке Ada ключевые слова *in* и *out* поясняют суть входных и выходных параметров.

Такая конвенция упорядочения параметров противоречит конвенции библиотек C, предполагающей указание изменяемого параметра в первую очередь. Конвенция «входные значения — изменяемые значения — выходные значения» кажется мне более разумной, но, даже если вы будете согласованно упорядочивать параметры любым иначе, вы окажете услугу программистам, которым придется читать ваш код.

***Подумайте о создании собственных ключевых слов* in *и* out** В отличие от Ada другие языки не поддерживают ключевые слова *in* и *out*. Однако даже в этом случае вы скорее всего сможете создать их с помощью препроцессора:

**Пример определения собственных ключевых слов *In* и *Out* (C++)**

```
#define IN
#define OUT
void InvertMatrix(
   IN Matrix originalMatrix,
   OUT Matrix *resultMatrix
);
...

void ChangeSentenceCase(
   IN StringCase desiredCase,
   IN OUT Sentence *sentenceToEdit
);
...
```

```
void PrintPageNumber(
   IN int pageNumber,
   OUT StatusType &status
);
```

В данном примере ключевые слова-макросы *IN* и *OUT* используются для документирования. Чтобы значение параметра можно было изменить в вызванном методе, параметр все же нужно передавать по указателю или по ссылке.

Прежде чем принять этот подход, обдумайте два его важных недостатка. Собственные ключевые слова *IN* и *OUT* окажутся незнакомыми большинству программистов, которые будут читать ваш код. Расширяя язык таким образом, делайте это согласованно, лучше всего в масштабе всего проекта. Второй недостаток в том, что компилятор не будет проверять соответствие параметров ключевым словам *IN* и *OUT*, из-за чего вы сможете отметить параметр как *IN* и все же изменить его внутри метода. Так вы только введете программиста, читающего ваш код, в заблуждение. Обычно для определения исключительно входных параметров лучше применять ключевое слово *const* языка C++.

***Если несколько методов используют похожие параметры, передавайте их в согласованном порядке*** Порядок параметров может как облегчить, так и затруднить их запоминание. Например, в C прототипы методов *fprintf()* и *printf()* различаются только тем, что *fprintf()* принимает файл в качестве дополнительного первого аргумента. Похожее отношение наблюдается и между методами *fputs()* и *puts()*, но в *fputs()* файл передается последним. Это досадное различие только затрудняет запоминание параметров названных методов.

С другой стороны, методы *strncpy()* и *memcpy()* в том же C принимают аргументы в одинаковом порядке: строка-приемник, строка-источник и максимальное число копируемых байт. Такое сходство помогает запомнить параметры обоих методов.

***Используйте все параметры*** Если вы передаете параметр в метод, используйте его, в противном случае удалите параметр из интерфейса метода. Наличие неиспользуемых параметров соответствует более высокому уровню ошибок. Исследования показали, что ошибки отсутствовали в 46% методов, не включавших неиспользуемых переменных, и только в 17–29% методов, содержавших более одной неиспользуемой переменной (Card, Church, and Agresti, 1986).

Это правило допускает одно исключение. При условной компиляции кода из компиляции могут быть исключены части метода, использующие некоторый параметр. Опасайтесь этого подхода, но, если вы убеждены, что все правильно, он вполне допустим. В общем, если у вас есть серьезная причина не использовать параметр, оставьте его в списке. Если таковой нет, очистите интерфейс метода от примесей.

***Передавайте переменные статуса или кода ошибки последними*** Переменные статуса и переменные, указывающие на ошибку, следует располагать в списке параметров последними. Они второстепенны по отношению к главной цели метода и являются исключительно выходными параметрами, поэтому такая конвенция вполне разумна.

***Не используйте параметры метода в качестве рабочих переменных*** Использовать передаваемые в метод параметры как рабочие переменные опасно. Создайте для этой цели локальные переменные. Так, в следующем фрагменте Java-кода переменная *inputVal* некорректно служит для хранения промежуточных результатов вычислений:

**Пример некорректного использования входного параметра (Java)**

```
int Sample( int inputVal ) {
   inputVal = inputVal * CurrentMultiplier( inputVal );
   inputVal = inputVal + CurrentAdder( inputVal );
   ...

Переменная inputVal уже не содержит входного значения.

   return inputVal;
}
```

В этом фрагменте переменная *inputVal* вводит в заблуждение, потому что при завершении метода она больше не содержит входного значения; она содержит результат вычисления, частично основанного на входном значении, и поэтому ее имя неудачно. Если позднее вам придется задействовать первоначальное входное значение в другом месте метода, вы, вероятно, задействуйте переменную *inputVal*, предполагая, что она содержит первоначальное значение, но это предположение будет ошибочным.

Можно ли решить эту проблему путем переименования *inputVal*? Наверное, нет. Переменной можно было бы присвоить имя вроде *workingVal*, но такое решение было бы неполным, так как это имя не говорит о том, что первоначальное значение переменной передается в метод извне. Вы могли бы присвоить ей нелепое имя *inputValThatBecomesWorkingVal* (входное значение, которое становится рабочим значением) или сдаться и просто назвать ее *x* или *val*, но все эти подходы неудачны.

Лучше избегать настоящих и будущих проблем, используя рабочие переменные явно, например:

**Пример корректного использования входного параметра (Java)**

```
int Sample( int inputVal ) {
   int workingVal = inputVal;
   workingVal = workingVal * CurrentMultiplier( workingVal );
   workingVal = workingVal + CurrentAdder( workingVal );
   ...
```

Если первоначальное значение *inputVal* понадобится здесь или где-то еще, оно все еще доступно.

```
     ...
   return workingVal;
}
```

Создание новой переменной *workingVal* поясняет роль *inputVal* и исключает возможность ошибочного использования *inputVal* в неподходящий момент. (Не рассматривайте это рассуждение как оправдание присвоения переменным имен *inputVal* или *workingVal*. Имена *inputVal* и *workingVal* просто ужасны и служат в данном примере только для пояснения ролей переменных.)

Присвоение входного значения рабочей переменной подчеркивает тот факт, что значение поступает в метод извне. Кроме того, это исключает возможность случайного изменения параметров. В C++ ответственность за это можно возложить на компилятор при помощи ключевого слова *const*. Отметив параметр как *const*, вы не сможете изменить его значение внутри метода.

**Перекрестная ссылка** О связанных с интерфейсами предположениях см. также главу 8, о документировании кода — главу 32.

***Документируйте выраженные в интерфейсе предположения о параметрах*** Если вы предполагаете, что передаваемые в метод данные должны иметь определенные характеристики, сразу же документируйте эти предположения. Документирование предположений и в самом методе, и в местах его вызова нельзя назвать пустой тратой времени. Пишите комментарии, не дожидаясь завершения работы над методом: к тому времени вы многое забудете. Еще лучше применить утверждения (assertions), позволяющие встроить предположения в код.

Какие типы предположений о параметрах следует документировать? Вот какие:

- вид параметров: являются ли они исключительно входными, изменяемыми или исключительно выходными;
- единицы измерения (дюймы, футы, метры и т. д.);
- смысл кодов статуса и ошибок, если для их представления не используются перечисления;
- диапазоны допустимых значений;
- специфические значения, которые никогда не должны передаваться в метод.

***Ограничивайте число параметров метода примерно семью*** 7 — магическое число. Психологические исследования показали, что люди, как правило, не могут следить более чем за семью элементами информации сразу (Miller, 1956). Это открытие используется в огромном числе дисциплин, поэтому резонно предположить, что большинство людей не может удерживать в уме более семи параметров метода одновременно.

**Перекрестная ссылка** Анализ интерфейсов см. в подразделе «Хорошая абстракция» раздела 6.2.

На практике возможность ограничения числа параметров зависит от того, как в выбранном вами языке реализована поддержка сложных типов данных. Программируя на современном языке, поддерживающем структурированные данные, вы можете передать в метод составной тип данных, содержащий 13 полей, и рассматривать его как один «элемент» данных. При использовании более примитивного языка вам, возможно, придется передать все 13 полей по отдельности.

Если вам постоянно приходится передавать в методы слишком большое число аргументов, ваши методы имеют слишком сильное сопряжение. Проектируйте методы или группы методов так, чтобы сопряжение было слабым. Если вы передаете одни и те же данные во многие разные методы, сгруппируйте эти методы и данные в класс.

***Подумайте об определении конвенции именования входных, изменяемых и выходных параметров*** Если нужно провести различие между входными, изменяемыми и выходными параметрами, сформулируйте соответствующую конвенцию их именования. Например, вы можете дополнить их префиксами *i_*, *m_* и *o_*. Программисты пословоохотливее могут использовать префиксы *Input_*, *Modify_* и *Output_*.

***Передавайте в метод те переменные или объекты, которые нужны ему для поддержания абстракции интерфейса*** Есть два конкурирующих подхода к передаче членов объекта в методы. Допустим, у вас есть объект, предоставляющий доступ к данным посредством 10 методов доступа, но вызываемому методу нужны лишь три элемента данных объекта.

Сторонники первого подхода утверждают, что в метод следует передать только три нужных ему элемента. Они считают, что это позволяет поддерживать минимальное сопряжение между методами, способствует пониманию методов, облегчает их повторное использование и т. д. Они говорят, что передача всего объекта в метод нарушает принцип инкапсуляции, позволяя вызванному методу использовать все 10 методов доступа.

Сторонники второго подхода утверждают, что следует передать весь объект. Они говорят, что если вызываемый метод получит доступ к дополнительным членам объекта, это позволит сохранить стабильность интерфейса метода. Им кажется, что именно передача трех конкретных элементов нарушает инкапсуляцию, потому что это указывает на конкретные элементы данных, используемые методом.

Я думаю, что оба этих правила слишком упрощены и не учитывают самого важного: *какую абстракцию формирует интерфейс метода*? Если абстракция подразумевает, что метод ожидает три конкретных элемента данных, которые по чистой случайности принадлежат одному объекту, передайте три элемента по отдельности. Если же абстракция состоит в том, что элементы данных всегда принадлежат конкретному объекту, над которым метод должен выполнять ту или иную операцию, тогда, раскрывая три этих специфических элемента, вы на самом деле нарушаете абстракцию.

Если при передаче всего объекта вы создаете объект, заполняете его тремя элементами, нужными методу, а после вызова извлекаете эти элементы из объекта, значит, вам следует передать в метод только три конкретных элемента, а не весь объект. (Обычно наличие кода, «подготавливающего» данные перед вызовом метода или «разбирающего» объект после вызова, — признак неудачного проектирования метода.)

Если же вам часто приходится изменять список параметров метода, при этом каждый раз параметры относятся к одному и тому же объекту, в метод следует передавать весь объект, а не конкретные элементы.

***Используйте именованные параметры*** Некоторые языки позволяют явно сопоставить формальные параметры с фактическими. Это делает применение параметров более ясным и помогает избегать ошибок, обусловленных неправильным сопоставлением параметров, например:

**Пример явной идентификации параметров (Visual Basic)**

```
Private Function Distance3d( _
```

Объявления формальных параметров.

```
   ByVal xDistance As Coordinate, _
   ByVal yDistance As Coordinate, _
   ByVal zDistance As Coordinate _
)
   ...
End Function
...
Private Function Velocity( _
   ByVal latitude as Coordinate, _
   ByVal longitude as Coordinate, _
   ByVal elevation as Coordinate _
)
   ...
```

Сопоставление фактических параметров с формальными.

```
   Distance = Distance3d( xDistance := latitude, yDistance := longitude, _
      zDistance := elevation )
   ...
End Function
```

Данный подход особенно полезен при использовании длинных списков параметров одинакового типа, потому что в этом случае вероятность неправильного сопоставления параметров более высока, а компилятор эту ошибку определить не может. Во многих средах явное сопоставление параметров может оказаться пальбой из пушки по воробьям, но в средах, от которых зависит безопасность людей, или других средах с повышенными требованиями к надежности дополнительный способ гарантии правильного сопоставления параметров не помешает.

***Убедитесь, что фактические параметры соответствуют формальным*** Формальные параметры, известные также как «фиктивные параметры» (dummy parameters), — это переменные, объявленные в определении метода. Фактическими параметрами называют переменные, константы или выражения, на самом деле передаваемые в метод.

По невнимательности довольно часто передают в метод переменную неверного типа — например, целое число вместо числа с плавающей запятой. (Эта проблема характерна только для слабо типизированных языков, таких как C, при использовании неполного набора предупреждений компилятора. Строго типизированные языки, такие как C++ и Java, не имеют этого недостатка.) Если аргументы являются исключительно входными, это редко становится проблемой: обычно компилятор при вызове метода преобразует фактический тип в формальный. Если это приводит к проблеме, компилятор обычно генерирует предупреждение. Но иногда, особенно если аргумент является и входным, и выходным, передача аргумента неверного типа может привести к серьезным последствиям.

Постарайтесь всегда проверять типы аргументов в списках параметров и внимательно изучайте предупреждения компилятора о несоответствии типов параметров.

## 7.6. Отдельные соображения по использованию функций

Современные языки, такие как C++, Java и Visual Basic, поддерживают и функции, и процедуры. Функция — это метод, возвращающий значение; процедуры значений не возвращают. В C++ все методы обычно называют «функциями», однако, с точки зрения семантики, функция, «возвращающая» *void*, является процедурой. Различие между функциями и процедурами выражено в семантике не слабее, чем в синтаксисе, и именно семантике следует уделять наибольшее внимание.

### Когда использовать функцию, а когда процедуру?

Пуристы утверждают, что функция должна возвращать только одно значение подобно математической функции. В этом случае все функции принимали бы только входные параметры и возвращали единственное значение традиционным путем. Функции всегда назывались бы в соответствии с возвращаемым значением: *sin()*, *CustomerID()*, *ScreenHeight()* и т. д. Процедуры, с другой стороны, могли бы принимать входные, изменяемые и выходные параметры в любом количестве.

Довольно часто можно встретить функцию, работающую как процедура, но возвращающую при этом код статуса. Логически она является процедурой, но из-за возврата значения официально ее следует называть функцией. Так, объект *report* мог бы иметь метод *FormatOutput()*, используемый подобным образом:

```
if ( report.FormatOutput( formattedReport ) = Success ) then ...
```

В этом примере метод *report.FormatOutput()* работает как процедура в том смысле, что он имеет входной параметр *formattedReport*, но технически это функция, потому что сам метод тоже возвращает значение. В защиту этого подхода вы могли бы сказать, что возвращаемое функцией значение не имеет отношения ни к главной цели функции (форматированию вывода), ни к имени метода (*report.FormatOutput()*). В этом смысле метод больше похож на процедуру, пусть даже технически он является функцией. Если возвращаемое значение служит для определения успеха или неудачи выполнения процедуры согласованно, это не вызывает замешательства.

Альтернативный подход — создание процедуры, принимающей переменную статуса в качестве явного параметра, например:

```
report.FormatOutput( formattedReport, outputStatus )
if ( outputStatus = Success ) then ...
```

Я предпочитаю именно этот вариант, но не потому, что трепетно отношусь к различию между функциями и процедурами, а потому, что такой код ясно разделяет вызов метода и проверку переменной статуса. Объединение вызова и проверки в одной строке увеличивает «плотность» команды, а значит, и ее сложность. Следующий вариант использования функции также хорош:

```
outputStatus = report.FormatOutput( formattedReport )
if ( outputStatus = Success ) then ...
```

Словом, используйте функцию, если основная цель метода — возврат значения, указанного в имени функции. Иначе применяйте процедуру.

### Возврат значения из функции

Использование функции сопряжено с риском того, что функция возвратит некорректное значение. Обычно это объясняется наличием нескольких путей выполнения функции, один из которых не устанавливает возвращаемого значения. Следуя моим советам, вы сведете этот риск к минимуму.

***Проверяйте все возможные пути возврата*** Создав функцию, проработайте в уме каждый возможный путь ее выполнения, дабы убедиться, что функция возвращает значение во всех возможных обстоятельствах. Целесообразно инициализировать возвращаемое значение в начале функции значением по умолчанию: это будет страховкой на тот случай, если функция не установит корректное возвращаемое значение.

***Не возвращайте ссылки или указатели на локальные данные*** Как только выполнение метода завершается и локальные данные выходят из области видимости, ссылки и указатели на локальные данные становятся некорректными. Если объект должен возвращать информацию о своих внутренних данных, пусть он сохранит ее в форме данных — членов класса. Реализуйте для него функции доступа, возвращающие данные-члены, а не ссылки или указатели на локальные данные.

## 7.7. Методы-макросы и встраиваемые методы

**Перекрестная ссылка** Даже если ваш язык не поддерживает препроцессор макросов, вы можете создать собственный препроцессор (см. раздел 30.5).

С методами-макросами связаны некоторые уникальные соображения. Следующие правила и примеры рассматриваются в контексте препроцессора C++. Если вы используете другой язык или препроцессор, адаптируйте правила к своей ситуации.

***Разрабатывая макрос, заключайте в скобки все, что можно*** Так как макросы и их аргументы расширяются в код, следите за тем, чтобы они расширялись так, как вам нужно. Одну частую проблему иллюстрирует следующий макрос:

#### Пример макроса, который расширяется неверно (C++)

```
#define Cube( a ) a*a*a
```

Если вы передадите в этот макрос неатомарное значение *a*, он выполнит умножение неверно. Так, выражение *Cube( x+1 )* расширится в *x+1 * x + 1 * x + 1*, что из-за приоритета операций умножения и сложения приведет к получению ошибочного результата. Вот улучшенная, но все еще не совсем правильная версия этого макроса:

**Пример макроса, который все еще расширяется неверно (C++)**

```
#define Cube( a ) (a)*(a)*(a)
```

Цель уже близка. Однако, если вы используете макрос *Cube()* в выражении, включающем операции с более высоким приоритетом, чем умножение, выражение *(a)*(a)*(a)* будет вычислено неверно. Что делать? Заключите в скобки все выражение:

**Пример макроса, с которым все в порядке (C++)**

```
#define Cube( a ) ((a)*(a)*(a))
```

***Заключайте макрос, включающий несколько команд, в фигурные скобки*** Макрос может включать несколько команд, что может привести к проблемам, если вы будете рассматривать их как единый блок, например:

**Пример неправильного макроса, состоящего из нескольких команд (C++)**

```
#define LookupEntry( key, index ) \
   index = (key - 10) / 5; \
   index = min( index, MAX_INDEX ); \
   index = max( index, MIN_INDEX );
...

for ( entryCount = 0; entryCount < numEntries; entryCount++ )
   LookupEntry( entryCount, tableIndex[ entryCount ] );
```

Этот макрос работает не так, как работал бы обычный метод: единственной частью макроса, выполняемой в цикле *for*, является первая строка:

```
index = (key - 10) / 5;
```

Чтобы устранить эту проблему, заключите макрос в фигурные скобки:

**Пример правильного макроса, состоящего из нескольких команд (C++)**

```
#define LookupEntry( key, index ) { \
   index = (key - 10) / 5; \
   index = min( index, MAX_INDEX ); \
   index = max( index, MIN_INDEX ); \
}
```

Замена вызовов методов макросами обычно считается рискованным и малопонятным (короче, плохим) подходом, так что используйте его только при необходимости.

***Называйте макросы, расширяющиеся в код подобно методам, так, чтобы при необходимости их можно было заменить методами*** Конвенция именования макросов в C++ подразумевает использование только заглавных букв. Если же макрос может быть заменен методом, называйте его в соответствии с конвенцией именования методов. Это позволит вам заменять макросы на методы и наоборот, не изменяя остального кода.

Следование этой рекомендации связано с риском. Если вы часто используете операции ++ и –– ради их побочных эффектов (в составе других выражений), то, принимая макросы за методы, вы столкнетесь с неприятностями. Это еще одна причина избегать побочных эффектов.

## Ограничения использования методов-макросов

Современные языки вроде C++ поддерживают много альтернатив макросам:

- ключевое слово *const* для объявления констант;
- ключевое слово *inline* для определения функций, которые будут компилироваться как встраиваемый код;
- шаблоны для безопасного в плане типов определения стандартных операций, таких как *min*, *max* и т. д.;
- ключевое слово *enum* для определения перечислений;
- директиву *typedef* для простых замен одного типа другим.

Бьерн Страуструп, создатель C++, пишет: «Макрос почти всегда указывает на недостаток языка программирования, программы или программиста... Если вы используете макросы, значит, вам не хватает возможностей отладчиков, инструментов, генерирующих перекрестные ссылки, средств профилирования и т. д.» (Stroustrup, 1997). Макросы полезны для выполнения условной компиляции (см. раздел 8.6), но добросовестные программисты обычно используют макросы вместо методов только в крайнем случае.

## Встраиваемые методы

Язык C++ поддерживает ключевое слово *inline*, служащее для определения встраиваемых методов. Иначе говоря, программист может разрабатывать код как метод, но во время компиляции компилятор постарается встроить каждый экземпляр метода прямо в код. Теоретически встраивание методов может повысить быстродействие кода, позволяя избежать затрат, связанных с вызовами методов.

***Не злоупотребляйте встраиваемыми методами*** Встраиваемые методы нарушают инкапсуляцию, потому что C++ требует, чтобы программист поместил код встраиваемого метода в заголовочный файл, доступный остальным программистам.

При встраивании метода каждый его вызов заменяется на полный код метода, что во всех случаях увеличивает объем кода и само по себе может создать проблемы.

Практическое применение встраивания аналогично применению прочих методик повышения быстродействия кода: профилируйте код и оценивайте результаты. Если ожидаемое повышение быстродействия не оправдывает забот, связанных с профилированием, нужным для проверки выгоды, оно не оправдывает и снижения качества кода.

**Контрольный список: высококачественные методы**

http://cc2e.com/0792

**Общие вопросы**

- ❑ Достаточна ли причина создания метода?
- ❑ Все ли части метода, которые целесообразно поместить в отдельные методы, сделаны отдельными методами?

- ❑ Имеет ли имя процедуры вид «выразительный глагол + объект»? Описывает ли имя функции возвращаемое из нее значение?
- ❑ Описывает ли имя метода все выполняемые в методе действия?
- ❑ Задали ли вы конвенции именования часто выполняемых операций?
- ❑ Имеет ли метод высокую функциональную связность? Решает ли он только одну задачу и хорошо ли он с ней справляется?
- ❑ Имеют ли методы слабое сопряжение? Являются ли связи метода с другими методами малочисленными, детальными, заметными и гибкими?
- ❑ Обусловлена ли длина метода его ролью и логикой, а не искусственным стандартом кодирования?

**Передача параметров**

- ❑ Формирует ли в целом список параметров метода согласованную абстракцию интерфейса?
- ❑ Разумно ли упорядочены параметры метода? Соответствует ли их порядок порядку параметров аналогичных методов?
- ❑ Документированы ли выраженные в интерфейсе предположения?
- ❑ Метод имеет семь параметров или меньше?
- ❑ Все ли входные параметры используются?
- ❑ Все ли выходные параметры используются?
- ❑ Не используются ли входные параметры в качестве рабочих переменных?
- ❑ Если метод является функцией, возвращает ли он корректное значение во всех возможных случаях?

**Перекрестная ссылка** Этот контрольный список позволяет определить качество методов. Вопросы, касающиеся этапов создания метода, приведены в контрольном списке «Процесс Программирования Псевдокода» (глава 9).

# Ключевые моменты

- Самая важная, но далеко не единственная причина создания методов — улучшение интеллектуальной управляемости программы. Сокращение кода — не такая уж и важная причина; повышение его удобочитаемости, надежности и облегчение его изменения куда важнее.
- Иногда огромную выгоду можно извлечь, создав отдельный метод для простой операции.
- Связность методов можно разделить на несколько видов. Самая лучшая — функциональная — достижима практически всегда.
- Имя метода является признаком его качества. Плохое, но точное имя часто указывает на плохое проектирование метода. Плохое и неточное имя не описывает роль метода. Как бы то ни было, плохое имя предполагает, что программу нужно изменить.
- Функцию следует использовать, только когда главной целью метода является возврат конкретного значения, описываемого именем функции.
- Добросовестные программисты используют методы-макросы с осторожностью и только в крайнем случае.

ГЛАВА 8

# Защитное программирование

http://cc2e.com/0861

**Содержание**



**Связанные темы**

- Сокрытие информации: подраздел «Скрывайте секреты (к вопросу о сокрытии информации)» раздела 5.3
- Дизайн изменений: подраздел «Определите области вероятных изменений» раздела 5.3
- Архитектура программного обеспечения: раздел 3.5
- Дизайн в проектировании: глава 5
- Отладка: глава 23

Защитное программирование не означает защиту своего кода словами: «Это так работает!» Его идея совпадает с идеей внимательного вождения, при котором вы готовы к любым выходкам других водителей: вы не пострадаете, даже если они совершат что-то опасное. Вы берете на себя ответственность за собственную защиту и в тех случаях, когда виноват другой водитель. В защитном программировании главная идея в том, что если методу передаются некорректные данные, то его работа не нарушится, даже если эти данные испорчены по вине другой программы. Обобщая, можно сказать, что в программах всегда будут проблемы, программы будут модифицироваться и разумный программист будет учитывать это при разработке кода.

Эта глава рассказывает, как защититься от беспощадного мира неверных данных, событий, которые «никогда» не могут случиться, и других программистских ошибок. Если вы опытный программист, можете пропустить следующий раздел про обработку входных данных и перейти к разделу 8.2, который рассказывает об утверждениях.

## 8.1. Защита программы от неправильных входных данных

Вы, возможно, слышали в школе выражение: «Мусор на входе — мусор на выходе»[1]. Это вариант предостережения потребителю от разработчиков ПО: пусть пользователь остерегается.

Для промышленного ПО принцип «мусор на входе — мусор на выходе» не слишком подходит. Хорошая программа никогда не выдает мусор независимо от того, что у нее было на входе. Вместо этого она использует принципы: «мусор на входе — ничего на выходе», «мусор на входе — сообщение об ошибке на выходе» или «мусор на входе не допускается». По сегодняшним стандартам «мусор на входе — мусор на выходе» — признак небрежного, небезопасного кода.

Существует три основных способа обработки входных мусорных данных, перечисленные далее.

***Проверяйте все данные из внешних источников*** Получив данные из файла, от пользователя, из сети или любого другого внешнего интерфейса, удостоверьтесь, что все значения попадают в допустимый интервал. Проверьте, что числовые данные имеют разрешенные значения, а строки достаточно коротки, чтобы их можно было обработать. Если строка должна содержать определенный набор значений (скажем, идентификатор финансовой транзакции или что-либо подобное), проконтролируйте, что это значение допустимо в данном случае, если же нет — отклоните его. Если вы работаете над приложением, требующим соблюдения безопасности, будьте особенно осмотрительны с данными, которые могут атаковать вашу систему: попыткам переполнения буфера, внедренным SQL-командам, внедренному HTML- или XML-коду, переполнениям целых чисел, данным, передаваемым системным вызовам и т. п.

***Проверяйте значения всех входных параметров метода*** Проверка значений входных параметров метода практически то же самое, что и проверка данных из внешнего источника, за исключением того, что данные поступают из другого метода, а не из внешнего интерфейса. В разделе 8.5 вы узнаете, как определить, какие методы должны проверять свои входные данные.

---

[1] «Garbage in, garbage out». Возможно, эту их школьную поговорку следует перевести нашей студенческой: «Каков стол, таков и стул». — *Прим. перев.*

***Решите, как обрабатывать неправильные входные данные*** Что делать, если вы обнаружили неверный параметр? В зависимости от ситуации вы можете выбрать один из дюжины подходов, подробно описанных в разделе 8.3.

Защитное программирование — это полезное дополнение к другим способам улучшения качества программ, описанным в этой книге. Лучший способ защитного кодирования — изначально не плодить ошибок. Итеративное проектирование, написание псевдокода и тестов до начала кодирования и низкоуровневая проверка соответствия проекту — это все, что помогает избежать добавления дефектов. Поэтому этим технологиям должен быть дан более высокий приоритет, чем защитному программированию. К счастью, вы можете использовать защитное программирование в сочетании с ними.

Защита от проблем, кажущихся несущественными, может иметь большее значение, чем можно подумать (рис. 8-1). В оставшейся части этой главы я расскажу о проверке данных из внешних источников, проверке входных параметров и обработке неправильных входных данных.


***Рис. 8-1.*** *Часть плавучего моста Interstate-90 в Сиэтле затонула во время шторма, потому что резервуары были оставлены открытыми. Они наполнились водой, и мост стал слишком тяжел, чтобы держаться на плаву. Обеспечение защиты от мелочей во время проектирования может значить больше, чем кажется*

## 8.2. Утверждения

Утверждение (assertion) — это код (обычно метод или макрос), используемый во время разработки, с помощью которого программа проверяет правильность своего выполнения. Если утверждение истинно, то все работает так, как ожидалось. Если ложно — значит, в коде обнаружена ошибка. Например, если система предполагает, что длина файла с информацией о заказчиках никогда не будет превышать 50 000 записей, программа могла бы содержать утверждение, что число записей меньше или равно 50 000. Пока это число меньше или равно 50 000, утвер-

ждение будет хранить молчание. Но как только записей станет больше 50 000, оно громко провозгласит об ошибке в программе.

Утверждения особенно полезны в больших и сложных программах, а также в программах, требующих высокой надежности. Они позволяют нам быстрее выявить несоответствия в интерфейсах, ошибки, вкравшиеся при изменении кода и т. п.

Обычно утверждение принимает два аргумента: логическое выражение, описывающее предположение, которое должно быть истинным, и сообщение, выводимое в противном случае. Вот как будет выглядеть утверждение на языке Java, если переменная *denominator* должна быть ненулевой:

### Пример утверждения (Java)

```
assert denominator != 0 : "denominator is unexpectedly equal to 0.";
```

В этом утверждении объявляется, что *denominator* не должен быть равен *0*. Первый аргумент — *denominator != 0* — логическое выражение, принимающее значение *true* или *false*. Второй — это сообщение, выводимое, когда первый аргумент равен *false* (т. е. утверждение ложно).

Используйте утверждения, чтобы документировать допущения, сделанные в коде, и чтобы выявить непредвиденные обстоятельства. Например, утверждения можно применять при проверке таких условий:

- значение входного (выходного) параметра попадает в ожидаемый интервал;
- файл или поток открыт (закрыт), когда метод начинает (заканчивает) выполняться;
- указатель файла или потока находится в начале (конце), когда метод начинает (заканчивает) выполняться;
- файл или поток открыт только для чтения, только для записи или для чтения и записи;
- значение входной переменной не изменяется в методе;
- указатель ненулевой;
- массив или другой контейнер, передаваемый в метод, может вместить по крайней мере X элементов;
- таблица инициализирована для помещения реальных значений;
- контейнер пуст (заполнен), когда метод начинает (заканчивает) выполняться;
- результаты работы сложного, хорошо оптимизированного метода совпадают с результатами метода более медленного, но написанного яснее.

Разумеется, это только основы, и ваши методы будут содержать много более специфических допущений, которые вы сможете документировать, используя утверждения.

Утверждения не предназначены для показа сообщений в промышленной версии — они в основном применяются при разработке и поддержке. Обычно их добавляют при компиляции кода во время разработки и удаляют при компиляции промышленной версии. В период разработки утверждения выявляют противоречивые допущения, непредвиденные условия, некорректные значения, переданные

методам, и т. п. При компиляции промышленной версии они могут быть удалены и, таким образом, не повлияют на производительность системы.

## Создание собственного механизма утверждений

**Перекрестная ссылка** Создание собственной процедуры утверждений — хороший пример программирования «с использованием языка», а не просто программирования «на языке». Подробнее об этих различиях см. раздел 34.4.

Многие языки программирования, включая C++, Java, и Microsoft Visual Basic, имеют встроенную поддержку утверждений. Если ваш язык не поддерживает процедуры утверждений напрямую, их легко написать. Стандартный макрос *assert* языка C++ не предусматривает вывода текстового сообщения. Вот пример улучшенного макроса *ASSERT* на C++:

**Пример макроса утверждения (C++)**

```
#define ASSERT( condition, message ) {       \
   if ( !(condition) ) {                     \
       LogError( "  Assertion failed: ",     \
           #condition, message );            \
       exit( EXIT_FAILURE );                 \
    }                                        \
}
```

## Общие принципы использования утверждений

Далее перечислены общие положения по применению утверждений.

***Используйте процедуры обработки ошибок для ожидаемых событий и утверждения для событий, которые происходить не должны*** Утверждения проверяют условия событий, которые *никогда* не должны происходить. Обработчик ошибок проверяет внештатные события, которые могут и не происходить слишком часто, но были предусмотрены писавшим код программистом и должны обрабатываться и в промышленной версии. Обработчик ошибок обычно проверяет некорректные входные данные, утверждения — ошибки в программе.

Если для обработки аномальной ситуации служит обработчик ошибок, он позволит программе адекватно отреагировать на ошибку. Если же в случае аномальной ситуации сработало утверждение, для исправления просто отреагировать на ошибку мало — необходимо изменить исходный код программы, перекомпилировать и выпустить новую версию ПО.

Будет правильно рассматривать утверждения как выполняемую документацию — работать программу с их помощью вы не заставите, но вы можете документировать допущения в коде более активно, чем это делают комментарии языка программирования.

***Старайтесь не помещать выполняемый код в утверждения*** Если в утверждении содержится код, возникает возможность удаления этого кода компилятором при отключении утверждений. Допустим, у вас есть следующее утверждение:

### Пример опасного использования утверждения (Visual Basic)

```
Debug.Assert( PerformAction() ) ' Невозможно выполнить действие.
```

**Перекрестная ссылка** Можете рассматривать этот случай как одну из многих проблем, связанных с размещением нескольких операторов на одной строке. Другие примеры см. в подразделе «Размещение одного оператора на строке» раздела 31.5.

Проблема здесь в том, что, если вы не компилируете утверждения, вы не компилируете и код, который выполняет указанное действие. Вместо этого поместите выполняемые выражения в отдельных строках, присвойте результаты статусным переменным и проверяйте значения этих переменных. Вот пример безопасного использования утверждения:

### Пример безопасного использования утверждения (Visual Basic)

```
actionPerformed = PerformAction()
Debug.Assert( actionPerformed ) '    Невозможно выполнить действие.
```

***Используйте утверждения для документирования и проверки предусловий и постусловий*** Предусловия и постусловия — это часть подхода к проектированию и разработке программ, известному как «проектирование по контракту» (Meyer, 1997). При использовании пред- и постусловий каждый метод или класс заключает контракт с остальной частью программы.

**Дополнительные сведения** О предусловиях и постусловиях см. «Object-Oriented Software Construction» (Meyer, 1997).

*Предусловия* — это соглашения, которые клиентский код, вызывающий метод или класс, обещает выполнить до вызова метода или создания экземпляра объекта. Предусловия — это обязательства клиентского кода перед кодом, который он вызывает.

*Постусловия* — это соглашения, которые метод или класс обещает выполнить при завершении своей работы. Постусловия — это обязательства метода или класса перед кодом, который их использует.

Утверждения — удобный инструмент для документирования пред- и постусловий. С этой целью можно использовать и комментарии, но в отличие от них утверждения могут динамически проверять, выполняются ли пред- и постусловия.

В следующем примере утверждения документируют пред- и постусловия в функции *Velocity*:

### Пример использования утверждений для документирования пред- и постусловий (Visual Basic)

```
Private Function Velocity ( _
   ByVal latitude As Single, _
   ByVal longitude As Single, _
   ByVal elevation As Single _
   ) As Single

   ' Предусловия
   Debug.Assert ( -90 <= latitude And latitude <= 90 )
   Debug.Assert ( 0 <= longitude And longitude < 360 )
   Debug.Assert ( -500 <= elevation And elevation <= 75000 )

   ...
```

```
    ' Постусловия
    Debug.Assert ( 0 <= returnVelocity And returnVelocity <= 600 )

    ' Возвращаемое значение
    Velocity = returnVelocity
End Function
```

Если бы переменные *latitude*, *longitude* и *elevation* поступили из внешнего источника, корректность их значений должна была быть проверена и обработана в коде обработчика ошибок, а не с помощью утверждений. Но если эти переменные поступили из доверенного внутреннего источника, а метод спроектирован в предположении, что их значения будут в разрешенном интервале, то применение утверждений допустимо.

**Перекрестная ссылка** Об устойчивости см. «Устойчивость против корректности» раздел 8.3.

***Для большей устойчивости кода проверяйте утверждения, а затем все равно обработайте возможные ошибки*** Каждая потенциально ошибочная ситуация обычно проверяется или утверждением, или кодом обработчика ошибок, но не тем и другим вместе. Некоторые эксперты утверждают, что необходим только один тип проверки (Meyer, 1997).

Однако реальные программы и проекты бывают слишком запутанными, чтобы можно было полагаться на одни лишь утверждения. В больших, долгоживущих системах различные части могут разрабатываться несколькими проектировщиками 5–10 лет и более. Разработка будет производиться в разное время и в разных версиях продукта. Эти проекты будут основаны на разных технологиях и сосредоточены на различных вопросах разработки системы. Проектировщики могут быть удалены друг от друга географически, особенно если элементы системы приобретались у независимых компаний. Программисты будут использовать различные стандарты кодирования в разное время жизни системы. В большой команде разработчиков некоторые неминуемо будут добросовестнее других, поэтому часть кода будет проверяться более тщательно, чем остальная. В любом случае, когда тестовые команды работают в нескольких географических регионах, а требования бизнеса приводят к изменению тестового покрытия от версии к версии, рассчитывать на всестороннее низкоуровневое тестирование системы нельзя.

В этих обстоятельствах одна и та же ошибка может быть проверена и с помощью утверждения, и обработчиком ошибок. Так, в исходном коде Microsoft Word условия, которые должны быть истинными, сперва помещаются в утверждения, а затем и в коде обработки ошибок рассматривается ситуация, когда утверждение ложно. В столь сложных и долгоживущих приложениях, как Word, утверждения служат для выявления как можно большего числа ошибок периода разработки. Но поскольку приложение очень сложное (миллионы строк кода) и прошло через столько изменений, неразумно ожидать обнаружения и исправления всех мыслимых ошибок до начала поставки приложения пользователям. Поэтому ошибки должны обрабатываться и в промышленной версии системы.

Вот как это можно сделать на примере функции *Velocity*:

**Пример использования утверждений для документирования пред- и постусловий (Visual Basic)**

```
Private Function Velocity ( _
   ByRef latitude As Single, _
   ByRef longitude As Single, _
   ByRef elevation As Single _
   ) As Single

   ' Предусловия
```

Так выглядит код утверждения.

```
   Debug.Assert ( -90 <= latitude And latitude <= 90 )
   Debug.Assert ( 0 <= longitude And longitude < 360 )
   Debug.Assert ( -500 <= elevation And elevation <= 75000 )
   ...

   ' Откорректируйте входные данные. Значения должны попадать
   ' в интервалы, указанные в вышестоящих утверждениях. Иначе
   ' они будут заменены ближайшими допустимыми значениями.
```

Таким может быть код, обрабатывающий неверные входные данные во время выполнения программы.

```
   If ( latitude < -90 ) Then
      latitude = -90
   ElseIf ( latitude > 90 ) Then
      latitude = 90
   End If
   If ( longitude < 0 ) Then
      longitude = 0
   ElseIf ( longitude > 360 ) Then
   ...
```

## 8.3. Способы обработки ошибок

Утверждения применяют для обработки ошибок, которые никогда не должны происходить. А что делать с возможными ошибками? В зависимости от обстоятельств вы можете вернуть некое нейтральное значение, заменить следующим корректным блоком данных, вернуть тот же результат, что и в предыдущий раз, подставить ближайшее допустимое значение, записать предупреждающее сообщение в файл, вернуть код ошибки, вызвать метод или объект — обработчик ошибки или прекратить выполнение. Вы также можете использовать несколько способов одновременно.

Рассмотрим эти приемы подробней.

***Вернуть нейтральное значение*** Иногда наилучшей реакцией на неправильные данные будет продолжение выполнения и возврат заведомо безопасного значения. Численные расчеты могут возвращать 0. Операция со строкой может вернуть пустую строку, а операция с указателем — пустой указатель. Метод рисования в видеоигре, получивший неправильное исходное значение цвета, может по

умолчанию использовать цвет фона или изображения. Однако в методе рисования рентгеновского снимка ракового больного вряд ли стоит применять «нейтральное значение». В таких случаях лучше прекратить выполнение программы, чем показать пациенту неправильные результаты.

***Заменить следующим корректным блоком данных*** Условия обработки потока данных иногда таковы, что следует просто вернуть следующие допустимые данные. Если при чтении информации из базы данных встречена испорченная запись, можно просто продолжить считывание, пока не будут найдены корректные данные. Если вы считываете показания термометра 100 раз в секунду и один раз не получили достоверного измерения, можно просто подождать 1/100 секунды и обратиться к следующему показанию.

***Вернуть тот же результат, что и в предыдущий раз*** Если программа считывания показаний термометра один раз не получила измерение, она может просто вернуть то же значение, что и в предыдущий раз. В зависимости от приложения температура скорее всего не сильно изменится за 1/100 секунды. Если в видеоигре запросу на прорисовку части экрана передано неверное значение цвета, вы можете просто вернуть тот же цвет, что и раньше. Но, авторизуя транзакции в банкомате, вы, пожалуй, не захотите использовать «то же значение, что и в предыдущий раз» — ведь это будет номер счета предыдущего клиента!

***Подставить ближайшее допустимое значение*** В некоторых случаях вы можете вернуть ближайшее допустимое значение, как выше в примере функции *Velocity*. Часто это обоснованный подход для получения показаний откалиброванных инструментов. Так, термометр мог бы быть откалиброван от 0 до 100 градусов по Цельсию. Если вы получаете значение меньше 0, можно заменить его на 0, как ближайшее допустимое значение. Если же значение больше 100, можно подставить 100. Если в операции со строкой ее длина заявлена меньшей 0, можно принять ее за 0. Мой автомобиль использует этот подход к обработке ошибок, когда я двигаюсь задним ходом. Так как спидометр не показывает отрицательную скорость, то при езде задним ходом, скорость просто равна 0 — ближайшему допустимому значению.

***Записать предупреждающее сообщение в файл*** Обнаружив неверные данные, вы можете решить записать предупреждение в файл журнала и продолжить работу. Этот подход можно сочетать с другими способами, такими как подстановка ближайшего допустимого значения или замена следующим корректным блоком данных. Используя такой журнальный файл, задумайтесь, можно ли его безопасно сделать общедоступным или же его надо зашифровывать либо защищать каким-либо иначе.

***Вернуть код ошибки*** Вы можете решить, что только определенные части системы будут обрабатывать ошибки. Другие же не будут обрабатывать ошибки локально, а будут просто сообщать, что обнаружена ошибка, и надеяться, что какой-либо другой вышестоящая в иерархии вызовов метод эту ошибку обработает. Конкретный механизм оповещения остальной системы об ошибке может быть следующим:

- установить значение статусной переменной;
- вернуть статус в качестве возвращаемого значения функции;

- сгенерировать исключение, используя встроенный в язык программирования механизм обработки исключений.

В этом случае не столь важно выбрать механизм обработки ошибок, как решить, какая часть системы будет обрабатывать ошибки напрямую, а какая — только сообщать об их возникновении. Если система должна быть безопасной, убедитесь, что вызывающие методы всегда проверяют коды возврата.

***Вызвать процедуру или объект — обработчик ошибок*** Другим подходом к централизованной обработке ошибок является создание глобальной специализированной процедуры или объекта. Преимущество его в том, что контроль над обработкой ошибок сосредоточен в одном месте, что облегчает отладку. С другой стороны, вся программа целиком будет зависеть от этого кода. Если же вы захотите повторно использовать какую-то часть программы в другой системе, придется перетаскивать туда и весь механизм обработки ошибок.

Этот подход может очень серьезно повлиять на безопасность. Если в программе возникнет переполнение буфера, злоумышленник сможет узнать адрес метода (объекта)-обработчика. Таким образом, при переполнении буфера во время работы приложения использовать этот способ небезопасно.

***Показать сообщение об ошибке, где бы она ни случилась*** Этот подход минимизирует накладные расходы на обработку ошибок. Однако он приводит к расползанию сообщений пользовательского интерфейса по коду приложения. Это может создавать сложности, если вы хотите реализовать целостный интерфейс пользователя, отделить этот интерфейс от остальной части системы или локализовать ваше ПО. Остерегайтесь также сообщить потенциальным злоумышленникам слишком многое — они часто используют сообщения об ошибках для поиска способа проникновения в систему.

***Обработать ошибку в месте возникновения наиболее подходящим способом*** В некоторых проектах предлагается обрабатывать ошибки локально, а выбор используемого метода остается за программистом, реализующим ту часть системы, где происходит ошибка.

Такой подход предоставляет разработчикам большую гибкость. Однако он таит в себе опасность, что система в целом не будет удовлетворять требованиям корректности и устойчивости (см. ниже). А в зависимости от того, какой в конечном итоге будет реакция на ошибку, этот метод может привести к потенциальному расползанию кода пользовательского интерфейса по системе. Это приведет к тем же проблемам, что и в случае с выводом сообщений об ошибках.

***Прекратить выполнение*** Некоторые системы прекращают работу при возникновении любой ошибки. Этот подход оправдан в приложениях, критичных к безопасности. Например, какая реакция на ошибку будет наилучшей, если ПО, контролирующее радиационное оборудование для лечения рака, получит некорректное значение радиационной дозы? Надо ли использовать то же значение, что и в предыдущий раз? А может, ближайшее допустимое или нейтральное значение? В этом случае остановка работы — наилучший вариант. Мы охотнее предпочтем перезагрузить машину, чем рискнуть применить неправильную дозу.

Похожий подход применим и для повышения безопасности Microsoft Windows. По умолчанию Windows продолжает работать, даже если журнал безопасности

переполнен. Но вы можете изменить конфигурацию Windows так, что при заполнении журнала сервер будет прекращать работу. Это может быть полезно для систем повышенной секретности.

## Устойчивость против корректности

Как нам показали примеры с видеоигрой и рентгеновской установкой, выбор подходящего метода обработки ошибки зависит от приложения, в котором эта ошибка происходит. Кроме того, обработка ошибок в общем случае может стремиться либо к большей корректности, либо к большей устойчивости кода. Разработчики привыкли применять эти термины неформально, но, строго говоря, эти термины находятся на разных концах шкалы. *Корректность* предполагает, что нельзя возвращать неточный результат; лучше не вернуть ничего, чем неточное значение. *Устойчивость* требует всегда пытаться сделать что-то, что позволит программе продолжить работу, даже если это приведет к частично неверным результатам.

Приложения, требовательные к безопасности, часто предпочитают корректность устойчивости. Лучше не вернуть никакого результата, чем неправильный результат. Радиационная машина — хороший пример применения такого принципа.

В потребительских приложениях устойчивость, напротив, предпочтительнее корректности. Какой-то результат всегда лучше, чем прекращение работы. Текстовый редактор, которым я пользуюсь, временами показывает последнюю на экране строку лишь частично. Хочу ли я, чтобы при обнаружении этой ситуации редактор завершал выполнение? Нет: когда я в следующий раз нажму Page Up или Page Down, экран обновится, и изображение исправится.

## Влияние выбора метода обработки ошибок на проектирование высокого уровня

При наличии такого широкого выбора надо стараться реагировать на неправильные значения параметров одинаково во всей программе. Способ обработки ошибок влияет на соответствие ПО требованиям корректности, устойчивости и другим атрибутам, не относящимся к функциональности. Выбор общего подхода к работе с некорректными данными — это вопрос архитектуры или высокоуровневого проектирования, и он должен быть рассмотрен на одном из этих этапов разработки системы.

Выбрав подход, придерживайтесь его неукоснительно. Если вы решили обрабатывать ошибки на высоком уровне, а в низкоуровневом коде просто сообщать о них, удостоверьтесь, что высокоуровневый код действительно их обрабатывает! Некоторые языки позволяют игнорировать возвращаемое функцией значение (в C++ вы не обязаны что-то то делать с возвращенным результатом), но не игнорируйте информацию об ошибке! Проверяйте значение, возвращаемое из функции. Даже если вы считаете, что ошибка в функции возникнуть не может, все равно проверяйте. Весь смысл защитного программирования в защите от ошибок, которых вы не ожидаете.

Эти принципы относятся к системным функциям, так же как и вашим собственным. Если только архитектура ПО не предусматривает игнорирования сбоев системных вызовов, проверяйте коды ошибок после каждого такого вызова. При обнаружении ошибки укажите ее номер и описание.

## 8.4. Исключения

Исключения — это специальное средство, позволяющее передать в вызывающий код возникшие ошибки или исключительные ситуации. Если код в некотором методе встречает неожиданную ситуацию и не знает, как ее обработать, то он генерирует исключение, т. е. фактически умывает руки со словами: «Я не знаю, что с этим делать, надеюсь, кто-нибудь другой знает, как на это реагировать!» Код, не имеющий понятия о контексте ошибки, может вернуть управление другой части системы, которая, возможно, лучше знает, как интерпретировать ошибку и сделать с ней что-то осмысленное.

Кроме того, исключения могут быть полезны для упрощения запутанной логики участка кода, как в примере «Переписатьс помощью *try-finally*» в разделе 17.3. Вот принцип действия исключений: метод, применяя оператор *throw*, создает объект-исключение. Код какого-либо другого метода, стоящего выше в иерархии вызовов, перехватит это исключение в блоке *try-catch*.

Популярные языки программирования по-разному реализуют исключения (табл. 8.1):

**Табл. 8-1. Поддержка исключений в популярных языках программирования**

| Параметры обработки исключений | C++ | Java | Visual Basic |
|---|---|---|---|
| Поддержка *try-catch* | Да. | Да. | Да. |
| Поддержка *try-catch-finally* | Нет. | Да. | Да. |
| Что генерируется | Объект класса *Exception* или производного от него, указатель на объект, объектная ссылка, другие типы данных, например, строка или целое число. | Объект класса *Exception* или производного от него. | Объект класса *Exception* или производного от него. |
| Эффект при не перехваченном исключении | Вызывается функция *std::unexpected()*, которая по умолчанию вызывает *std::terminate()*, в свою очередь по умолчанию вызывающая функцию *abort()*. | Если это «проверяемое исключение», то прекращается работа потока, в котором оно возникло. Если это «исключение периода выполнения», то оно игнорируется. | Программа завершает работу. |
| Генерируемые исключения должны быть определены в интерфейсе класса | Нет. | Да. | Нет. |
| Перехватываемые исключения должны быть определены в интерфейсе класса | Нет. | Да. | Нет. |

Программы, использующие исключения как часть нормальной работы алгоритма, страдают от всех проблем с читабельностью и удобством сопровождения так же, как и классический спагетти-код.

*Энди Хант и Дэйв Томас (Andy Hunt and Dave Thomas)*

Исключения и наследование имеют общее свойство: используемые разумно, они могут уменьшить сложность. Используемые чрезмерно, они могут сделать код абсолютно нечитаемым. Этот раздел содержит предложения по реализации преимуществ исключений и способы избежать трудностей, которые часто с ними связаны.

***Используйте исключения для оповещения других частей программы об ошибках, которые нельзя игнорировать*** Основное преимущество исключений состоит в их способности сигнализировать об ошибке так, что ее нельзя проигнорировать (Meyers, 1996). При других подходах к обработке ошибок есть вероятность, что сбойная ситуация останется незамеченной. Исключения устраняют такую возможность.

***Генерируйте исключения только для действительно исключительных ситуаций*** Применение исключений должно быть зарезервировано только для действительно исключительных случаев — иначе говоря, для ситуаций, которые нельзя реализовать другими методами кодирования. Исключения используются в таких же обстоятельствах, как и утверждения: для событий, которые не просто редко происходят, а которые *никогда* не должны случаться.

Исключения представляют собой компромисс между возможностью обработки непредвиденных ситуаций, с одной стороны, и повышением сложности — с другой. Исключения ухудшают инкапсуляцию, требуя от кода, вызывающего метод, знать, какие исключения могут быть сгенерированы внутри него. Это усложняет код, что противоречит Главному Техническому Императиву ПО (см. главу 5), суть которого в снижении сложности.

***Не используйте исключения по мелочам*** Если ошибка может быть обработана локально, там ее и обрабатывайте. Не генерируйте в коде неперехватываемое исключение, если ошибка может быть исправлена на месте.

***Избегайте генерировать исключения в конструкторах и деструкторах, если только вы не перехватываете их позднее*** Правила обработки исключений очень быстро усложняются, когда исключения генерируются в конструкторах и деструкторах. Так, в C++ деструктор не вызывается, пока объект не создан полностью. Это значит, что, если код в конструкторе сгенерировал исключение, деструктор вызван не будет, что приведет к возможной утечке ресурсов (Meyers, 1996; Stroustrup, 1997). Аналогичные замысловатые правила относятся и к исключениям внутри деструкторов.

Приверженцы языка могут сказать, что запомнить эти правила очень легко. Но все же программисты — простые смертные, у них могут возникнуть трудности с запоминанием правил. Наилучшая программистская практика — избегать излишней сложности, которая создается при написании такого кода.

**Перекрестная ссылка** О поддержании целостных абстракций интерфейса см. подраздел «Хорошая абстракция» раздела 6.2.

***Генерируйте исключения на правильном уровне абстракции*** Интерфейс метода и класса должен представлять собой целостную абстракцию. Генерируемые исключения — такая же часть интерфейса, как и специальные типы данных.

Решив передать исключение, удостоверьтесь, что уровень абстракции исключения и метода совпадают. Вот как не надо делать:

**Плохой пример Java-класса, который генерирует исключение на неверном уровне абстракции**

```
class Employee {
   ...
```

Объявление исключения с неправильным уровнем абстракции.

```
   public TaxId GetTaxId() throws EOFException {
      ...
   }
   ...
}
```

Функция *GetTaxId()* передает низкоуровневое исключение *EOFException* вызывающей стороне. Она не обрабатывает исключение сама, а раскрывает некоторые детали своей реализации, генерируя низкоуровневое исключение. Это привязывает клиентский код не к классу *Employee*, а к коду внутри класса *Employee*, генерирующему исключение *EOFException*. Инкапсуляция нарушена, управляемость кода ухудшается.

Вместо этого код *GetTaxId()* должен передавать исключение, соответствующее интерфейсу класса, частью которого он является, например, так:

**Хороший пример Java-класса, который генерирует исключение на правильном уровне абстракции**

```
class Employee {
   ...
```

Объявление исключения, соответствующего уровню абстракции.

```
   public TaxId GetTaxId() throws EmployeeDataNotAvailable {
      ...
   }
   ...
}
```

Код обработки исключений внутри *GetTaxId()*, возможно, просто устанавливает соответствие между исключениями *io_disk_not_ready* и *EmployeeDataNotAvailable*, что гораздо лучше, так как сохраняется абстракция интерфейса.

***Вносите в описание исключения всю информацию о его причинах*** Каждое исключение возникает при определенных обстоятельствах, обнаруженных кодом в момент генерации этого исключения. Эти сведения недоступны тому, кто читает сообщение об исключении. Убедитесь, что это сообщение содержит достаточно информации для понимания причины генерации исключения. Например, если причиной был неправильный индекс элемента массива, включите в описание верхнюю и нижнюю границы массива и некорректное значение индекса.

***Избегайте пустых блоков catch*** Иногда возникает искушение оставить без внимания исключение, которое вы не знаете, как обработать. Например:

**Плохой пример игнорирования исключения (Java)**

```
try {
   ...
   // много кода
  ...
} catch ( AnException exception ) {
}
```

Такой подход говорит о том, что либо код внутри блока *try* генерирует исключение без причины, либо код в блоке *catch* не обрабатывает возможную исключительную ситуацию. Выясните, в чем суть проблемы, и исправьте блоки *try* или *catch*.

Изредка можно столкнуться с ситуацией, когда исключение более низкого уровня не соответствует уровню абстракции вызывающего метода. В этом случае хотя бы задокументируйте, почему блок *catch* должен быть пустым. Это можно сделать в комментариях или записав сообщение в файл журнала, например:

**Хороший пример игнорирования исключения на Java**

```
try {
   ...
   // много кода
  ...
} catch ( AnException exception ) {
   LogError( "Unexpected exception" );
}
```

***Выясните, какие исключения генерирует используемая библиотека*** Если вы работаете с языком, не требующим, чтобы метод или класс объявляли возможные исключения, убедитесь, что вам известно, какие исключения могут возникнуть в коде используемых библиотек. Неперехваченное исключение из библиотеки приведет к аварийному завершению программы так же легко, как и исключение, сгенерированное в вашем коде. Если библиотечные исключения не документированы, создайте код-прототип и протестируйте библиотеки, чтобы их выявить.

***Рассмотрите вопрос о централизованном выводе информации об исключениях*** Поддержание целостности в обработке исключений обеспечивает централизованный генератор сообщений об исключениях. Он содержит базу знаний о том, какие это исключения, как каждое из них должно быть обработано, каков формат их сообщений и т. п.

Вот пример упрощенного обработчика исключений. Он просто печатает диагностическое сообщение:

**Пример централизованного генератора сообщений об исключениях, часть 1 (Visual Basic)**

```
Sub ReportException( _
   ByVal className, _
   ByVal thisException As Exception _
)
   Dim message As String
   Dim caption As String

   message = "Exception: " & thisException.Message & "." & ControlChars.CrLf & _
      "Class:   " & className & ControlChars.CrLf & _
      "Routine: " & thisException.TargetSite.Name & ControlChars.CrLf
   caption = "Exception"
   MessageBox.Show( message, caption, MessageBoxButtons.OK, _
      MessageBoxIcon.Exclamation )

End Sub
```

**Дополнительные сведения** Об этой технологии см. «Practical Standards for Microsoft Visual Basic .NET» (Foxall, 2003).

Этот обработчик можно использовать следующим образом:

**Пример централизованного генератора сообщений об исключениях, часть 2 (Visual Basic)**

```
Try
   ...
Catch exceptionObject As Exception
   ReportException( CLASS_NAME, exceptionObject )
End Try
```

Код этой версии *ReportException()* несложен. В реальных приложениях вы можете сделать отчет настолько кратким или подробным, насколько это необходимо в вашем обработчике исключений.

Если вы решили создать централизованный генератор сообщений, примите во внимание основные проблемы с централизованной обработкой ошибок, обсуждаемые в подразделе «Вызвать процедуру или объект — обработчик ошибок» раздела 8.3.

***Стандартизуйте использование исключений в вашем проекте*** Чтобы сохранить процедуру обработки исключений максимально интеллектуально управляемой, вы можете стандартизовать использование исключений несколькими способами.

- Если вы работаете с языком, таким как C++, который позволяет генерировать исключения разных типов, стандартизуйте, что конкретно будет создаваться. В целях совместимости с другими языками подумайте об использовании только объектов, порожденных от базового класса *Exception*.
- Подумайте о создании собственного класса исключений, который может служить базовым классом для всех исключений, возникающих в вашем проекте. Это поможет централизовать и стандартизовать регистрацию, обработку и другие действия с ошибками.

- Определите конкретные случаи, в которых код может использовать синтаксис *throw-catch* для локальной обработки ошибок.
- Определите конкретные случаи, в которых код может сгенерировать исключение, не перехватываемое локально.
- Решите, будет ли использоваться централизованный генератор сообщений об исключениях.
- Определите, допускаются ли исключения в конструкторах и деструкторах.

**Перекрестная ссылка** Об альтернативных подходах к обработке ошибок см. раздел 8.3.

***Рассмотрите альтернативы исключениям*** Некоторые языки поддерживают исключения 5–10 лет и более. Однако до сих пор нет общепринятых правил их безопасного использования.

Некоторые программисты применяют исключения для обработки ошибок только потому, что их язык программирования предоставляет такой механизм. Вам всегда следует принимать во внимание все возможные методы обработки ошибок: локальную обработку ошибок, возврат кода ошибки, запись отладочной информации в файл, прекращение работы системы и др. Обрабатывать ошибки с помощью исключений только потому, что это позволяет язык, — классический пример программирования *на* языке, а не *с использованием* языка (см. разделы 4.3 и 34.4).

И напоследок подумайте, действительно ли вашей программе необходимо обрабатывать исключения. Точка. Как заметил Бьерн Страуструп, иногда лучшей реакцией на серьезную ошибку периода выполнения будет освобождение всех ресурсов и прекращение работы. Пусть пользователь перезапустит программу с надлежащими входными данными (Stroustrup, 1997).

## 8.5. Изоляция повреждений, вызванных ошибками

Изоляция повреждений, или баррикада, — это стратегия, сходная с тем, как изолируются отсеки в трюме корабля. Если корабль налетает на айсберг и в днище появляется пробоина, отсеки задраиваются, и остальная часть корабля не страдает. Баррикады также аналогичны брандмауэрам, предотвращающим распространение огня из одной части здания в другую. (Ранее баррикады и назывались брандмауэрами, но сейчас термин «брандмауэр» обычно относится к блокировке нежелательного сетевого трафика.)

Один из способов изоляции в целях защитного программирования состоит в разработке набора интерфейсов в качестве оболочки для «безопасных» частей кода. Проверяйте корректность данных, пересекающих границу безопасной области, и реагируйте соответственно, если данные неправильные (рис. 8-2).

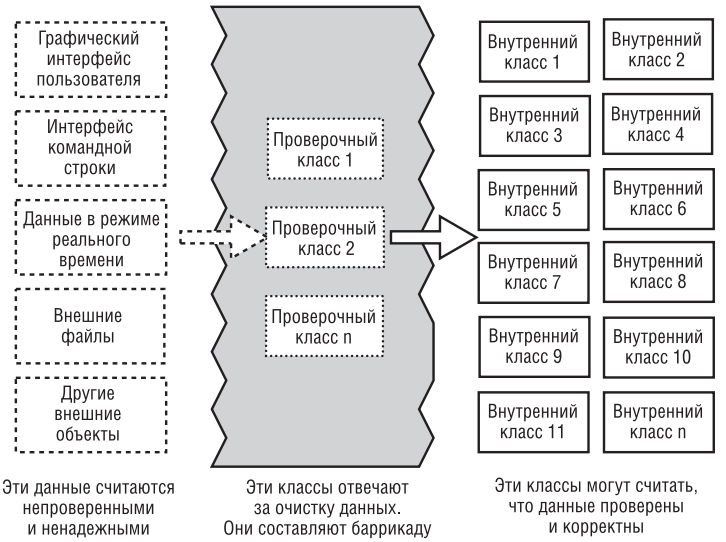



***Рис. 8-2.*** *Выделение части кода для работы с непроверенными данными, а части — для работы с только корректными данными, может быть эффективным способом освободить большую часть программы от ответственности за проверку допустимости данных*

Тот же подход применим и на уровне класса. Открытые методы класса предполагают, что данные небезопасны и отвечают за их проверку и исправление. Если данные были проверены открытыми методами класса, закрытые методы могут считать, что данные безопасны.

Этот подход можно представить и такой аналогией: данные стерилизуются, прежде чем войти в операционную. Все, что находится в ней, считается безопасным. Ключевой вопрос проектирования — решить, что должно быть в операционной, что — остаться снаружи и где быть дверям. Иначе говоря, какие методы поместить внутри безопасной зоны, какие — снаружи, а какие будут проверять данные. Простейший способ — проверка внешних данных по мере их поступления. Но информацию часто необходимо проверять неоднократно, на нескольких уровнях, поэтому иногда требуется многоуровневая стерилизация.

***Преобразовывайте входные данные к нужному типу в момент ввода***
Данные на входе обычно представлены в форме строки или числа. Иногда значение соответствует булевому типу, например, «да» или «нет». Иногда — перечислимому, скажем, *Color_Red*, *Color_Green*, и *Color_Blue*. Сохранение данных неопределенного типа в течение неизвестного периода времени усложняет программу и увеличивает шансы, что кто-нибудь может вывести программу из строя, указав «Да» в качестве цвета. Преобразуйте входные данные в надлежащую форму как можно раньше.

### Связь между баррикадами и утверждениями

Применение баррикад делает отчетливым различие между утверждениями и обработкой ошибок. Методы с внешней стороны баррикады должны использовать обработчики ошибок, поскольку небезопасно делать любые предположения о данных. Методы внутри баррикад должны использовать утверждения, так как данные, переданные им, считаются проверенными при прохождении баррикады. Если один из методов внутри баррикады обнаруживает некорректные данные, это следует считать ошибкой в программе, а не в данных.

Использование баррикад также иллюстрирует значимость принятия решения об обработке ошибок на уровне архитектуры. Решение, какой код находится внутри, а какой — снаружи баррикады, принимается на уровне архитектуры.

## 8.6. Отладочные средства

Еще один ключевой аспект защитного программирования — отладка, способная стать могучим союзником в быстром обнаружении ошибок.

### Не применяйте ограничения промышленной версии к отладочной версии автоматически

**Дополнительные сведения** Об использовании отладочного кода в защитном программировании см. «Writing Solid Code» (Maguire, 1993).

Известным заблуждением программистов является предположение, что ограничения промышленной версии относятся и ко времени разработки. Промышленная версия должна работать быстро. Отладочная — может себе позволить работать медленно. Промышленная версия должна быть экономна с ресурсами. Отладочная — может быть расточительной. Промышленная версия не должна позволять пользователю делать опасные действия. Отладочная — может предоставлять дополнительные возможности без риска нарушить безопасность.

Одна моя программа интенсивно использовала четырехсвязный список. Код этого списка содержал ошибки, и иногда список повреждался. Я добавил пункт меню для проверки целостности этого связного списка.

В отладочном режиме Microsoft Word содержит код, который в момент простоя каждые несколько секунд проверяет целостность объекта *Document*. Это помогает быстро обнаруживать повреждение данных, что упрощает диагностику ошибок.

Будьте готовы поступиться скоростью и ресурсоемкостью во время разработки в обмен на встроенные средства, позволяющие процессу разработки двигаться более гладко.

### Внедрите поддержку отладки как можно раньше

Чем раньше вы добавите отладочные средства, тем больше они помогут. Обычно вы не добавляете отладочную информацию, пока несколько раз не столкнетесь с проблемой. Если же вы внедрите поддержку отладки после первого раза или перенесете ее из предыдущего проекта, она будет помогать вам на протяжении всей работы.

## Используйте наступательное программирование

Исключительные случаи должны обрабатываться так, чтобы во время разработки они были очевидны, а в промышленном коде — позволяли продолжить работу. Майкл Ховард и Дэвид Леблан назвали этот подход «наступательным программированием» (Howard and LeBlanc, 2003).

**Перекрестная ссылка** Об обработке непредвиденных случаев см. подраздел «Советы по использованию операторов *case*» раздела 15.2.

Допустим, у вас есть оператор *case*, который, как вы ожидаете, будет обрабатывать только 5 видов событий. Во время разработки вариант по умолчанию нужно использовать для генерации предупреждения «Эй! Здесь еще один вариант! Исправьте программу!». Однако в промышленной версии реакция в этом случае должна быть более вежливой. Можно, например, делать запись в журнал ошибок.

Вот некоторые приемы наступательного программирования.

> Неработающая программа обычно приносит меньше вреда, чем работающая плохо.
>
> *Энди Хант и Дэйв Томас* (*Andy Hunt and Dave Thomas*)

- Убедитесь, что утверждения завершают работу программы. Нельзя, чтобы у программистов вошло в привычку просто нажимать клавишу Enter для пропуска уже известной проблемы. Сделайте проблему достаточно мучительной, чтобы ее исправили.
- Заполняйте любую выделенную память, чтобы можно было обнаружить ошибки выделения памяти.
- Заполняйте все файлы и потоки, чтобы выявить ошибки файловых форматов.
- Убедитесь, что при попадании в ветвь *default* или *else* всех операторов *case* программа прекращает работу или еще как-то заставляет обратить на это внимание.
- Заполняйте объекты мусором перед самым их удалением.
- Настройте программу на отправку вам журналов ошибок по электронной почте, чтобы видеть, какие ошибки происходят в рабочем ПО, если, конечно, это можно сделать в ваших программах.

Иногда нападение — лучшая защита. Чем жестче требования во время разработки, тем проще эксплуатация программы.

## Запланируйте удаление отладочных средств

Если вы пишете код для себя, возможно, было бы хорошо оставить всю отладочную информацию в программе. Но в коде для коммерческого использования требования к скорости и размеру скорее всего этого не допустят. Решите заранее, как избежать постоянного перетаскивания отладочного кода в программу и из нее. Вот несколько способов сделать это:

***Для контроля версий и сборки программы используйте инструменты ant и make*** Средства контроля версий позволяют создавать варианты программы из одних и тех же исходных файлов. А инструменты для сборки позволяют вам настроить включение отладочного кода в режиме разработки и его исключение в коммерческой версии.

**Перекрестная ссылка** О контроле версий см. раздел 28.2.

***Используйте встроенный препроцессор*** Если в вашей программной среде есть препроцессор, как, например, в C++, вы можете добавлять или удалять отладочный код простым изменением параметра компиляции. Препроцессор можно задействовать непосредственно, а можно писать макросы, работающие с его определениями. Вот пример написания кода, напрямую использующего препроцессор:

**Пример непосредственного использования препроцессора для управления отладочным кодом (C++)**

Для добавления кода отладки используйте директиву *#define*, чтобы определить символ *DEBUG*. Для исключения отладочного кода просто не определяйте *DEBUG*.

```
#define DEBUG
...

#if defined( DEBUG )
// отладочный код
...

#endif
```

У этой темы несколько вариаций. Вместо простого определения *DEBUG* вы можете присвоить ему значение, а затем проверять именно это значение, а не просто факт определения символа. Так вы можете различать несколько уровней отладочного кода. Какой-то код вы хотели бы использовать в программе все время, поэтому вы окружаете его операторами вроде *#if DEBUG > 0*. Другой отладочный код может понадобиться только в специальных целях, и вы можете заключить его в операторы *#if DEBUG == POINTER_ERROR*. В других местах вы захотите установить различные уровни отладки, для чего годятся такие выражения, как *#if DEBUG > LEVEL_A*.

Если вы не хотите распространять *#if defined()* по всему коду, можно написать макрос препроцессора, выполняющий ту же задачу. Вот его пример:

**Пример использования макроса препроцессора для управления отладочным кодом на C++**

```
#define DEBUG
#if defined( DEBUG )
define DebugCode( code_fragment )   { code_fragment }
else
#define DebugCode( code_fragment )
#endif
...

DebugCode(
```

Этот код добавляется или удаляется в зависимости от того, определен ли символ *DEBUG*.

```
   statement 1;
   statement 2;
   ...
   statement n;
```

```
);
...
```

Как и в первом примере применения препроцессора, эта технология может быть изменена различными способами, что позволит выполнить более изощренные действия, чем полное включение или исключение отладочного кода.

***Напишите собственный препроцессор*** Если язык не содержит препроцессор, то для включения/исключения отладочного кода довольно легко написать свой. Разработайте правила для добавления отладочного кода и напишите свой прекомпилятор, следующий этим соглашениям. Скажем, в Java вы могли бы написать прекомпилятор для обработки ключевых слов *//*#BEGIN DEBUG* и *//*#END DEBUG*. Напишите сценарий для вызова препроцессора, а затем скомпилируйте полученный после него код. В долгосрочной перспективе вы сбережете время. Кроме того, вы не сможете случайно скомпилировать необработанный код.

**Перекрестная ссылка** О препроцессорах и источниках информации об их написании см. подраздел «Препроцессоры» раздела 30.3.

***Используйте отладочные заглушки*** Зачастую для выполнения отладочных проверок вы можете вызвать процедуру. Во время разработки она могла бы выполнять несколько операций перед тем, как управление вернется вызывающей стороне. В промышленном коде сложную процедуру можно заменить процедурой-заглушкой, которая сразу вернет управление или выполнит перед этим пару быстрых операций. Этот подход лишь немного снижает производительность и является более быстрым решением, чем написание собственного препроцессора. Храните и отладочную, и итоговую версии процедур, и вы сможете быстро переключаться между ними.

**Перекрестная ссылка** О заглушках см. раздел 22.5.

Вы можете начать с метода, разработанного для проверки переданных ему указателей:

**Пример метода, использующего отладочную заглушку (C++)**

```
void DoSomething(
   SOME_TYPE *pointer;
   ...
   ) {

   // проверка переданных сюда параметров

Это строка вызывает процедуру для проверки указателя.
   CheckPointer( pointer );
   ...

}
```

Во время разработки процедура *CheckPointer()* будет выполнять полную проверку указателя. Это будет медленно, но эффективно, например, так:

**Пример метода, проверяющего указатели во время разработки (C++)**

Этот метод проверяет любой переданный ему указатель. Во время разработки ее можно использовать для стольких проверок, сколько вы сможете выдержать.

```
void CheckPointer( void *pointer ) {
   // выполнить проверку 1 – например, что указатель не равен NULL
   // выполнить проверку 2 – например, что какой-то его
    // обязательный признак действителен
   // выполнить проверку 3 – например, что область,
    // на которую он указывает, не повреждена
   ...
   // выполнить проверку n-...
}
```

Когда код готов к эксплуатации, вас, возможно, не устроят накладные расходы, связанные с такой проверкой указателей. Тогда вы удалите предыдущий код и добавьте следующий метод:

**Пример метода, проверяющего указатели во время эксплуатации (C++)**

Эта процедура сразу же возвращает управление.

```
void CheckPointer( void *pointer ) {
   // никакого кода; просто возврат управления
}
```

Это отнюдь не исчерпывающий обзор всех способов удаления средств отладки. Но его должно быть достаточно, чтобы подать вам идею о том, что может работать в вашей программной среде.

## 8.7. Доля защитного программирования в промышленной версии

Один из парадоксов защитного программирования состоит в том, что во время разработки вы бы хотели, чтобы ошибка была заметной: лучше пусть она надоедает, чем будет существовать риск ее пропустить. Но во время эксплуатации вы бы предпочли, чтобы ошибка была как можно более ненавязчивой, чтобы программа могла элегантно продолжить или прекратить работу. Далее перечислены основные принципы для определения, какие инструменты защитного программирования следует оставить в промышленной версии, а какие — убрать.

***Оставьте код, которые проверяет существенные ошибки*** Решите, какие части программы могут содержать скрытые ошибки, а какие — нет. Скажем, разрабатывая электронную таблицу, вы можете скрывать ошибки, касающиеся обновления экрана, так как в худшем случае это приведет к неправильному изображению. А вот в вычислительном модуле скрытых ошибок быть не должно, поскольку такие ошибки могут привести к неверным расчетам в электронной таблице. Большинство пользователей предпочтут помучиться с некорректным выводом на экран, чем с неправильным расчетом налогов и аудитом налоговой службы.

***Удалите код, проверяющий незначительные ошибки*** Если последствия ошибки действительно незначительны, удалите код, который ее проверяет. В нашем примере вы могли бы удалить код, проверяющий обновление экрана электронной таблицы. При этом «удалить» значит не физически убрать код, но использовать управление версиями, переключатели прекомпилятора или другую технологию для компиляции программы без этого кода. Если занимаемое место несущественно, проверочный код можно оставить, но настроив для ненавязчивой записи сообщений в журнал ошибок.

***Удалите код, приводящий к прекращению работы программы*** Как я уже говорил, если на стадии разработки ваша программа обнаруживает ошибку, ее надо сделать позаметнее, чтобы ее могли исправить. Часто наилучшим действием при выявлении ошибки будет печать диагностического сообщения и прекращение работы. Это полезно даже для незначительных ошибок.

Во время эксплуатации пользователям нужна возможность сохранения своей работы, прежде чем программа рухнет. И поэтому они, вероятно, будут согласны терпеть небольшие отклонения в обмен на поддержание работоспособности программы на достаточное для сохранения время. Пользователи не приветствуют ничего, что приводит к потере результатов их работы, независимо от того, насколько это помогает отладке и в конце концов улучшает качество продукта. Если ваша программа содержит отладочный код, способный привести к потере данных, уберите его из промышленной версии.

***Оставьте код, который позволяет аккуратно завершить работу программы*** Если программа содержит отладочный код, определяющий потенциально фатальные ошибки, оставьте его — это позволит элегантно завершить работу. Например, в марсоходе Pathfinder инженеры намеренно оставили часть отладочного кода. Ошибка произошла после того, как Pathfinder совершил посадку. С помощью отладочных средств, оставленных в нем, инженеры из лаборатории реактивных двигателей смогли диагностировать проблему и загрузить исправленный код. В результате Pathfinder полностью выполнил свою миссию (March, 1999).

***Регистрируйте ошибки для отдела технической поддержки*** Обдумайте возможность оставить отладочные средства в промышленной версии, но изменить их поведение на более подходящее. Если вы заполнили ваш код утверждениями, прекращающими выполнение программы на стадии разработки, на стадии эксплуатации можно не удалять их совсем, а настроить процедуру утверждения на запись сообщений в файл.

***Убедитесь, что оставленные сообщения об ошибках дружелюбны*** Если вы оставляете в программе внутренние сообщения об ошибках, проверьте, что они дружественны к пользователю. Пользователь одной из моих первых программ сообщила мне, что получила сообщение, гласившее: «У тебя неправильно выделена память для указателя, черт возьми!» К счастью для меня, у нее было чувство юмора. Общепринятым и эффективным подходом является уведомление пользователя о «внутренней ошибке» и вывод телефона и адреса электронной почты, по которым о ней можно сообщить.

# 8.8. Защита от защитного программирования

> Слишком много чего-либо — это плохо, но слишком много виски — это просто достаточно.
>
> *Марк Твен*

Избыток защитного программирования сам по себе создает проблемы. Если вы проверяете данные, передаваемые через параметры, во всех возможных местах и всеми возможными способами, ваша программа будет слишком большой и медленной. Еще хуже то, что дополнительный код, необходимый для защитного программирования, увеличивает сложность. Написанный в этих целях код не лишен недостатков, и вы можете находить дефекты в коде защитного программирования, так же как и в обычном коде, особенно если вы добавляете его мимоходом. Подумайте, где надо защищаться, и соответственно расставьте приоритеты защитного программирования.

### Контрольный список: защитное программирование

http://cc2e.com/0868

**Общие**

- ❑ Реализована ли в методе защита от некорректных входных данных?
- ❑ Используете ли вы утверждения для документирования допущений, включая пред- и постусловия?
- ❑ Используются ли утверждения для документирования только тех условий, которые никогда не должны происходить?
- ❑ Определены ли в архитектуре или высокоуровневом проекте системы технологии обработки ошибок?
- ❑ Указано ли в архитектуре или высокоуровневом проекте системы, чему будет отдаваться предпочтение при обработке ошибок: устойчивости или корректности?
- ❑ Построены ли баррикады для изоляции разрушительного эффекта ошибок и уменьшения объема кода, занятого в обработке ошибок?
- ❑ Установлены ли отладочные средства таким образом, что их можно активизировать и деактивировать без особых проблем?
- ❑ Хватает ли защитного кода: не слишком много и не слишком мало?
- ❑ Использованы ли технологии наступательного программирования, чтобы затруднить пропуск ошибок на стадии разработки?

**Исключения**

- ❑ Определен ли в проекте стандартизованный подход к обработке исключений?
- ❑ Рассмотрены ли альтернативы использованию исключений?
- ❑ Обрабатывается ли ошибка по возможности локально или генерируется нелокальное исключение?
- ❑ Возможны ли исключения в конструкторах и деструкторах?
- ❑ Генерируются ли исключения в методах на подходящих уровнях абстракции?
- ❑ Содержит ли каждое исключение все относящиеся к нему исходные данные?
- ❑ Свободен ли код от пустых блоков *catch*? (Или, если блок *catch* действительно допустим, задокументировано ли это?)

**Вопросы безопасности**

- Действительно ли код, проверяющий некорректные входные данные, контролирует попытки переполнения буфера, внедрения SQL- и HTML-кода, переполнения целых чисел и других злонамеренных действий?
- Все ли ошибочные коды возврата проверяются?
- Все ли исключения перехватываются?
- Не содержат ли сообщения об ошибках информацию, которая может помочь злоумышленнику взломать систему?

# Дополнительные ресурсы

Просмотрите следующие ресурсы по защитному программированию.

http://cc2e.com/0875

### Безопасность

Howard, Michael, and LeBlanc David. *Writing Secure Code*, 2d ed. Redmond, WA: Microsoft Press, 2003. Авторы обсуждают важность вопроса доверия ко входным данным для безопасности системы. Книга открывает глаза на то, какими многочисленными способами может быть нарушена работа программы. Некоторые из них связаны с проектированием, но большинство — нет. Книга охватывает весь диапазон вопросов о требованиях к системе, проектировании, кодировании и тестировании.

### Утверждения

Maguire, Steve. *Writing Solid Code*. Redmond, WA: Microsoft Press, 1993. Глава 2 содержит отличное обсуждение вопросов применения утверждений, в том числе несколько интересных примеров утверждений из широко известных продуктов Microsoft.

Stroustrup, Bjarne. *The C++ Programming Language*, 3d ed. Reading, MA: Addison-Wesley, 1997. В разделе 24.3.7.2 описано несколько вариантов реализации утверждений в C++, включая взаимосвязь утверждений и пред- и постусловий.

Meyer, Bertrand. *Object-Oriented Software Construction*, 2d ed. New York, NY: Prentice Hall PTR, 1997. Эта книга содержит наиболее полное обсуждение пред- и постусловий.

### Исключения

Meyer, Bertrand. *Object-Oriented Software Construction*, 2d ed. New York, NY: Prentice Hall PTR, 1997. Глава 12 содержит подробное обсуждение обработки исключений.

Stroustrup, Bjarne. *The C++ Programming Language*, 3d ed. Reading, MA: Addison-Wesley, 1997. В главе 14 подробно обсуждается обработка исключений на C++. Раздел 14.11 содержит отличное резюме, состоящее из 21 совета по использованию исключений в C++.

Meyers, Scott. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Reading, MA: Addison-Wesley, 1996. В темах 9–15 описаны разнообразные нюансы по обработке исключений в C++.

Arnold, Ken, James Gosling, and David Holmes. *The Java Programming Language*, 3d ed. Boston, MA: Addison-Wesley, 2000. Глава 8 содержит обсуждение обработки исключений на Java.

Bloch, Joshua. *Effective Java Programming Language Guide*. Boston, MA: Addison-Wesley, 2001. В темах 39–47 описаны нюансы обработки исключений на Java.

Foxall, James. *Practical Standards for Microsoft Visual Basic .NET*. Redmond, WA: Microsoft Press, 2003. В главе 10 описана обработка исключений на Visual Basic.

## Ключевые моменты

- Промышленный код должен обрабатывать ошибки более изощренно, чем по принципу «мусор на входе — мусор на выходе».
- С помощью технологии защитного программирования ошибки легче находить, легче исправлять, и они наносят меньше вреда промышленному коду.
- Утверждения позволяют обнаружить ошибки на ранней стадии, особенно в больших системах, системах повышенной надежности и в системах с часто изменяемым кодом.
- Выбор способа обработки некорректных входных данных — это ключевое решение обработки ошибок, принимаемое на этапе высокоуровневого проектирования.
- Исключения предоставляют возможность обработки ошибок в измерении, отличном от нормального хода алгоритма. Если они используются с осторожностью, то являются важным дополнением в интеллектуальном инструментальном наборе программиста. Применять их следует после сравнения с другими технологиями обработки ошибок.
- Ограничения, применяемые к промышленной версии системы, не обязательно должны относиться и ко времени разработки. Пользуясь этим преимуществом, вы можете добавлять в отладочную версию любой код, помогающий быстро выявлять ошибки.

ГЛАВА 9

# Процесс программирования с псевдокодом

**Содержание**

http://cc2e.com/0936



**Связанные темы**

- Создание высококачественных классов: глава 6
- Характеристики высококачественных методов: глава 7
- Проектирование при конструировании: глава 5
- Стиль комментирования: глава 32

Всю эту книгу можно рассматривать как подробное описание процесса программирования, в результате которого создаются классы и методы, и в данной главе этапы этого процесса рассматриваются во всех деталях. Здесь описывается «программирование в малом» — конкретные шаги построения отдельных классов и составляющих их методов, шаги, неизбежные в проекте любого размера. В этой главе также рассмотрен процесс программирования с псевдокодом (ППП, Pseudocode Programming Process), уменьшающий объем работы по проектированию и документированию и улучшающий качество и первого, и второго.

Если вы опытный программист, можете пропустить эту главу, но взгляните на обзор этапов и просмотрите советы по конструированию методов с помощью ППП в разделе 9.3. Некоторые программисты применяют описываемый процесс на полную, поскольку он приносит большую выгоду.

ППП — это не единственная процедура создания классов и методов. В разделе 9.4 описаны наиболее популярные альтернативные подходы, включая разработку с изначальными тестами и проектирование по контракту.

# 9.1. Этапы создания классов и методов

К конструированию классов можно подходить по-разному, но обычно это итеративный процесс создания общей структуры класса, создание списка его методов, их конструирование и проверка класса как единого целого. Создание класса может быть запутанным процессом (рис. 9-1), поскольку проектирование как таковое — тоже процесс запутанный (причины описаны в разделе 5.1).


***Рис. 9-1.*** *Детали конструирования классов могут различаться, но обычно порядок действий такой*

## Этапы создания класса

Ниже перечислены основные этапы создания класса.

***Создание общей структуры класса*** Проектирование класса связано с множеством отдельных вопросов. Определите функции класса, «секреты», скрытые в нем и точный уровень абстракции, предоставляемый интерфейсом класса. Определите, наследуется ли класс от другого класса и могут ли другие классы наследоваться от него. Определите основные открытые методы класса и спроектируйте все нетривиальные структуры данных, используемые им. Пройдитесь по всем этим пунктам столько раз, сколько необходимо для создания четкой структуры класса. Эти и многие другие вопросы подробно обсуждаются в главе 6.

***Конструирование всех методов класса*** Определив на первом этапе основные функции класса, переходите к конструированию каждого отдельного метода. При конструировании отдельных методов обычно выясняется потребность в дополнительных методах (как более низкого, так и более высокого уровня), и вопросы, возникающие при их создании, приводят к необходимости пересмотра общей структуры класса.

***Оценка и тестирование всего класса*** Обычно каждый метод тестируется при его создании. После того как класс в целом становится работоспособным, его нужно пересмотреть и протестировать как единое целое для выявления тех проблем, которые невозможно определить на уровне отдельных методов.

## Этапы построения метода

Многие метода класса, такие как аксессоры или интерфейсы к другим классам, могут быть простыми и понятными для реализации. Реализация других будет сложнее, и для их создания требуется систематический подход. Основные действия по созданию метода — проектирование, проверка структуры, кодирование и проверка кода — обычно выполняются в такой последовательности (рис. 9-2):

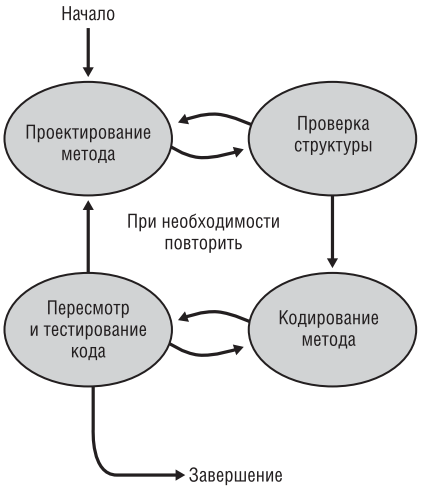



***Рис. 9-2.*** *Основные действия по созданию метода*

Специалисты разработали множество подходов к созданию методов, но мой любимый — процесс программирования с псевдокодом — описан в следующем разделе.

# 9.2. Псевдокод для профи

Псевдокодом называют неформальную нотацию на естественном языке, описывающую работу алгоритма, метода, класса или программы. «Процесс программирования с псевдокодом» относится к конкретной методике применения псевдокода для эффективного создания кода методов.

Поскольку псевдокод напоминает естественный язык, разумно предположить, что любая перефразировка ваших мыслей, выполненная на нем, будет иметь одинаковый эффект. На практике же вы обнаружите, что некоторые стили псевдокода предпочтительней других.

- Применяйте формулировки, в точности описывающие отдельные действия.
- Избегайте синтаксических элементов языков программирования. Псевдокод позволяет проектировать на несколько более высоком уровне, чем код. Применяя конструкции языка программирования, вы мыслите на более низком уровне и теряете преимущества проектирования на высоком уровне, загружая себя ненужными синтаксическими ограничениями.

**Перекрестная ссылка** Об уровнях комментирования см. подраздел «Виды комментариев» раздела 32.4.

- Пишите псевдокод на уровне намерений. Описывайте назначение подхода, а не то, как этот подход нужно реализовать на выбранном языке программирования.
- Пишите псевдокод на достаточно низком уровне, так чтобы код из него генерировался практически автоматически. Если псевдокод написан на слишком высоком уровне, могут возникнуть проблемы кодирования. Детализируйте псевдокод до тех пор, пока вам не покажется, что проще было бы написать код.[1]

Написав псевдокод, вы окружаете его кодом, а псевдокод превращаете в комментарии программы. Если вы руководствуетесь перечисленными правилами создания псевдокода, комментарии в вашей программе будут полными и ясными.

Вот пример псевдокода, в котором нарушены практически все перечисленные правила:

### Пример плохого псевдокода

```
увеличить номер ресурса на 1
выделить структуру dlg посредством malloc
если malloc() возвращает NULL вернуть 1
вызвать OSrsrc_init для инициализации ресурса
*hRsrcPtr = номер ресурса
вернуть 0
```

Какие намерения описывает этот блок псевдокода? Трудно сказать, поскольку написан он плохо. Этот так называемый псевдокод плох потому, что включает конкретику целевого языка программирования: **hRsrcPtr* (описание указателя, специфичное для языка C) и *malloc*() (функция C). Этот блок псевдокода показывает, как будет написан код, а не описывает его назначение. Он вдается в излишние подробности: вернет ли процедура *1* или *0*. Если посмотреть, можно ли превратить этот псевдокод в нормальные комментарии, видно, что толку от него мало.

А вот описание тех же действий на гораздо лучшем псевдокоде:

### Пример хорошего псевдокода

```
Отслеживать текущее число используемых ресурсов
Если другой ресурс доступен
   Выделить структуру для диалогового окна
   Если структура для диалогового окна может быть выделена
      Учесть, что используется еще один ресурс
      Инициализировать ресурс
      Хранить номер ресурса в вызывающей программе
   Конец «если»
Конец «если»
Вернуть true, если новый ресурс был создан; иначе вернуть false
```

[1] К сожалению, эти советы хорошо применимы лишь для псевдокода на английском языке или для случаев использования экзотических языков программирования с конструкциями, основанными на естественном языке, отличном от английского. — *Прим. перев.*

Этот псевдокод лучше предыдущего, так как полностью написан на естественном языке и не использует специфических конструкций целевого языка программирования. В первом примере псевдокод подлежал реализации только на C. Во втором же псевдокод не накладывает ограничений на используемый язык. Второй блок также написан на уровне описания намерений. О чем речь во втором блоке? Наверное, это легче понять, чем в первом блоке.

Хотя второй блок написан на понятном естественном языке, он достаточно точен и подробен, чтобы быть основой программы. Когда предложения этого псевдокода будут преобразованы в комментарии, они станут хорошим пояснением назначения кода.

Вот выгоды, которые вы получите, применяя такой стиль псевдокода.

- Псевдокод упрощает пересмотр конструкции — вам не потребуется вникать в исходный код.
- Псевдокод поддерживает идею итеративного усовершенствования. Вы начинаете с высокоуровневой конструкции, уточняете ее до псевдокода, который в свою очередь преобразуете в исходный код. Такое последовательное усовершенствование, осуществляемое шаг за шагом, позволяет проверять свои проектные решения по мере перехода на более низкие уровни. В результате вы обнаруживаете высокоуровневые ошибки на самом верхнем уровне, среднеуровневые — на среднем, а низкоуровневые — на самом нижнем, прежде чем они создадут проблемы.
- Псевдокод упрощает внесение изменений. Что проще: исправить линию на чертеже или снести стену и сдвинуть ее на метр в сторону? В программировании эффект не столь драматичен в плане физических усилий, но идея та же: несколько строк псевдокода легче исправить, чем страницу кода. Одна из основ успеха проекта — отловить ошибку на «наименее значимой стадии», когда для ее исправления требуется минимум усилий. Поиск ошибки на стадии псевдокода требует гораздо меньше затрат, чем после полного кодирования, тестирования и отладки, так что есть экономический стимул обнаружить ошибку как можно раньше.

  **Дополнительные сведения** О преимуществах внесения изменения на наименее значимых стадиях см. книгу Энди Гроува «High Output Management» (Grove, 1983).
- Псевдокод упрощает комментирование программ. В типичной ситуации вы сначала пишете код, а затем добавляете комментарии. В ППП предложения псевдокода становятся комментариями, так что на самом деле их проще оставить, чем удалить.
- Псевдокод сопровождать проще, чем другие виды проектной документации. При других подходах проектная документация отделена от кода, и внесение в нее изменений порождает несоответствие. В ППП предложения псевдокода становятся комментариями программы. Внося изменения в комментарии, вы, таким образом, поддерживаете в корректном состоянии проектную документацию.

Псевдокод как инструмент проектирования трудно переоценить. Исследования показали, что программисты предпочитают псевдокод за его возможности упрощать создание программных конструкций, помощь в определении некорректных проектных решений, простоту документирования и внесения изменений (Ramsey, Atwood, and Van Doren, 1983). Псевдокод — не един-

ственный инструмент детального проектирования, но наряду с ППП — полезная вещь в инструментарии программиста. Попробуйте его в деле. В следующем разделе я расскажу как.

## 9.3. Конструирование методов с использованием ППП

В этом разделе описаны этапы конструирования методов, а именно:

- проектирование метода;
- кодирование метода;
- проверка кода;
- наведение глянца;
- повторение предыдущих шагов при необходимости.

### Проектирование метода

**Перекрестная ссылка** О других аспектах проектирования см. главы с 5 по 8.

Определив состав методов класса, приступайте к их проектированию. Допустим, вы хотите написать метод вывода сообщения об ошибке, основанного на коде ошибки, и назвали этот метод *ReportErrorMessage*(). Вот неформальная спецификация *ReportErrorMessage*():

> *ReportErrorMessage()* принимает в качестве входного параметра код ошибки и выводит сообщение об ошибке, соответствующее этому коду. Он отвечает за обработку недопустимых кодов. Если программа интерактивная, *ReportErrorMessage()* выводит сообщение пользователю. Если она работает в режиме командной строки, *ReportErrorMessage()* заносит сообщение в файл. После вывода сообщения *ReportErrorMessage()* возвращает значение статуса, указывающее, успешно ли он завершился.

Этот метод используется в качестве примера во всей главе, а в оставшейся части этого раздела описывается его проектирование.

**Перекрестная ссылка** О проверке предварительных условий см. главы 3 и 4.

***Проверка предварительных условий*** Прежде чем что-либо делать с самой процедурой, убедитесь, что функции метода четко определены и соответствуют общим проектным решениям.

***Определите задачу, решаемую методом*** Сформулируйте задачу, решаемую методом настолько детально, чтобы можно было переходить созданию метода. Если проект высокого уровня достаточно подробен, эта работа уже сделана. Проект верхнего уровня должен содержать по крайней мере:

- информацию, скрываемую методом;
- входные данные;
- выходные данные;

**Перекрестная ссылка** О пред- и постусловиях см. раздел 8.2.

- предусловия, которые гарантированно должны соблюдаться до вызова метода (входные значения находятся в заданном диапазоне, потоки инициализированы, файлы открыты или закрыты, буферы заполнены или очищены и т. д.);

- постусловия, которые гарантированно должны соблюдаться, прежде чем метод вернет управление вызывающей программе (выходные значения находятся в заданном диапазоне, потоки инициализированы, файлы открыты или закрыты, буферы заполнены или очищены и т. д.).

Вот как это выглядит для метода *ReportErrorMessage*():

- метод скрывает текст сообщения и текущий метод обработки (интерактивный или командной строки);
- выполнение каких-либо предусловий не требуется;
- входными данными является код ошибки;
- выходные данные двух видов: сообщение об ошибке и статус, возвращаемый *ReportErrorMessage*() вызывающей программе;
- возвращаемый статус должен принимать одно из двух значений: *Success* или *Failure*.

***Название метода*** Вопрос именования метода кажется тривиальным, но хорошее название — признак высокого стиля программирования и дело это непростое. Вообще метод должен иметь понятное, недвусмысленное имя. Затруднения в выборе имени метода могут свидетельствовать о том, что его назначение не совсем понятно. Неясное, невыразительное имя метода сродни предвыборным обещаниям политиков. Вроде бы о чем-то оно говорит, но если задуматься — непонятно о чем. Если можно дать методу более ясное имя, сделайте это. Если невыразительное имя — результат неясных проектных решений, вернитесь к ним и измените их.

**Перекрестная ссылка** Об именовании методов см. раздел 7.3.

В нашем примере *ReportErrorMessage*() — вполне недвусмысленное имя. Хорошее имя.

***Решите, как тестировать метод*** В процессе написания метода думайте о том, как вы будете его тестировать. Это принесет пользу вам при блочном тестировании и тестировщикам, проводящим независимое тестирование.

В нашем примере входные данные просты, так что можно планировать тестирование *ReportErrorMessage*() со всеми допустимыми кодами ошибок и различными неверными кодами.

**Дополнительные сведения** О различных подходах к конструированию, ориентированных на предварительное написание тестов см. книгу «Test-Driven Development: By Example» (Beck, 2003).

***Исследуйте функциональность, предоставляемую стандартными библиотеками*** Основная возможность улучшить качество и производительность своего кода — повторно использовать имеющийся хороший код. Если метод кажется вам слишком сложным и у вас возникают проблемы с его проектированием, спросите себя, не реализована ли часть его функциональности в библиотеках языка, платформы или средства разработки, которые вы применяете. Нет ли нужного кода в стандартных библиотеках вашей компании? Множество алгоритмов уже реализовано, протестировано, обсуждено в профессиональной литературе, пересмотрено и усовершенствовано. Не тратьте время на реализацию готового алгоритма, по которому написана кандидатская диссертация.

***Продумайте обработку ошибок*** Подумайте обо всем плохом, что может случиться с вашим методом. Подумайте о плохих входных данных, недопустимых значениях, возвращаемых другими методами, и т. д.

Методы могут обрабатывать ошибки разными способами, и вам нужно четко определиться с одним из них. Если стратегию обработки ошибок определяет архитектура программы, вы можете просто следовать этой стратегии. В других случаях вам следует решить, какой подход будет оптимален в данном конкретном случае.

***Думайте об эффективности*** В зависимости от ситуации вы можете подходить к эффективности одним из двух способов. В первом случае — в подавляющем большинстве систем — эффективность некритична. При этом убедитесь, что интерфейс метода достаточно абстрагирован, а код читабелен и при необходимости вы сможете его легко усовершенствовать. При хорошей инкапсуляции вы сможете заменить медленные, ресурсоемкие конструкции языка высокого уровня более эффективным алгоритмом или реализацией на быстром, компактном языке низкого уровня, не затронув при этом другие методы.

**Перекрестная ссылка** Об эффективности см. главы 25 и 26.

Во втором случае — в незначительном числе систем — производительность критична. Проблемы производительности могут быть связаны с недостатком соединений с базой данных, ограниченной памятью, малым количеством доступных дескрипторов и другими ресурсами. Архитектура должна указывать, сколько ресурсов каждому методу (или классу) может быть предоставлено и как быстро он должен выполнять свои операции.

Как правило, не стоит тратить много усилий на оптимизацию отдельных методов. Эффективность в основном определяется конструкцией высокого уровня. Обычно микрооптимизация выполняется, только когда закончена вся программа и выясняется, что высокоуровневая конструкция исчерпала свои возможности обеспечить нужную производительность. Не теряйте время на вылизывание отдельных методов, пока не выяснится, что это необходимо.

***Исследуйте алгоритмы и типы данных*** Если доступные стандартные библиотеки не предоставляют нужной функциональности, имеет смысл исследовать литературу с описанием алгоритмов. Если вы нашли подходящий готовый алгоритм, корректно адаптируйте его к применяемому вами языку программирования.

***Пишите псевдокод*** У вас не будет сложностей, если вы прошли все предыдущие этапы, основное назначение которых в том, чтобы у вас сложилось четкое понимание того, что нужно писать.

**Перекрестная ссылка** Это обсуждение предполагает, что при создании псевдокода метода применялись правильные способы проектирования (см. главу 5).

Закончив предыдущие этапы, можно приступать к написанию высокоуровневого псевдокода. Запускайте редактор кода или интегрированную среду разработки и пишите псевдокод, который станет основой исходного текста программы.

Начните с основных моментов, а затем детализируйте их. Самая главная часть метода — заголовок-комментарий, описывающий действия метода, так что начните с краткой формулировки назначения метода. Написание этой формулировки поможет вам прояснить ваше понимание метода. Если вы испытываете затруднения при написании этого обобщенного комментария, вам, видимо, следует лучше разобраться с ролью этого метода. Вот пример краткого заголовка-комментария метода:

### Пример заголовка метода

```
Этот метод выводит сообщение об ошибке на основании кода ошибки, получаемого от
вызывающей программы. Способ вывода сообщения зависит от режима работы, который он
определяет сам. Он возвращает значение, указывающее на успешное завершение или сбой.
```

Написав общий комментарий, добавьте высокоуровневый псевдокод:

### Пример псевдокода метода

```
Этот метод выводит сообщение об ошибке на основании кода ошибки, получаемого от
вызывающей программы. Способ вывода сообщения зависит от режима работы, который он
определяет сам. Он возвращает значение, указывающее на успешное завершение или сбой.

Установить статус по умолчанию в "сбой".
Найти сообщение, соответствующее коду ошибки.

Если код ошибки корректен
    Если работа в интерактивном режиме, вывести сообщение
    и указать успешный статус.

    Если работа в режиме командной строки, запротоколировать
    сообщение об ошибке и указать успешный статус.

Если код ошибки некорректен, информировать пользователя
об обнаружении внутренней ошибки.

Вернуть статус
```

Еще раз: этот псевдокод достаточно высокого уровня, он не содержит конструкций языка программирования, а объясняет последовательность действий на естественном языке.

***Продумайте применение данных*** К структуре данных можно подходить с разных позиций. В нашем примере данные простые, и манипуляция над ними не является существенной частью метода. В противном случае важно продумать основные фрагменты данных до построения логики метода. Когда вы будете строить логику метода, структуры основных типов данных окажутся весьма полезны.

**Перекрестная ссылка** Об использовании переменных см. главы 10–13.

***Проверьте псевдокод*** Написав псевдокод и спроектировав данные, уделите минутку просмотру написанного. Задумайтесь, как бы вы объяснили это кому-то другому.

**Перекрестная ссылка** О методиках обзоров см. главу 21.

Попросите кого-нибудь прочитать написанное или выслушать ваше объяснение. Вам может показаться глупым просить коллегу посмотреть на какие-то 11 строк псевдокода, но результат вас удивит. Псевдокод более явно обозначит ваши ошибочные намерения, чем код на языке программирования. К тому же люди охотней просматривают несколько строк псевдокода своих коллег, чем 35 строк программы на C++ или Java.

Убедитесь, что вы имеете четкое представление о том, что и как делает метод. Если вы не понимаете его концептуально, на уровне псевдокода, какой же тогда у вас шанс разобраться в нем на уровне языка программирования? Если его не понимаете вы, кто его поймет?

**Перекрестная ссылка** Об итерациях см. раздел 34.8.

***Опишите несколько идей псевдокодом и выберите лучшую (пройдите по циклу)*** Прежде чем кодировать, реализуйте как можно больше своих идей в псевдокоде. Приступив к кодированию, вы эмоционально вовлекаетесь в этот процесс, и вам труднее отказаться от плохого проекта и начать заново.

Общая идея: раз за разом проходиться по псевдокоду, пока каждое его предложение не станет настолько простым, что под ним можно будет вставить строку программы, а псевдокод оставить в качестве документации. Часть псевдокода, написанного при первых проходах, может оказаться достаточно высокоуровневой и потребовать дальнейшей декомпозиции. Не забывайте это сделать. Если вам не понятно, как закодировать какой-то фрагмент, продолжайте работать с псевдокодом, пока это не прояснится. Продолжайте уточнение и декомпозицию, пока это не будет выглядеть как напрасная трата времени по сравнению с написанием настоящего кода.

## Кодирование метода

Спроектировав метод, приступайте к его конструированию. Конструирование можно производить в стандартном порядке, а при необходимости отступить от него (рис. 9-3).

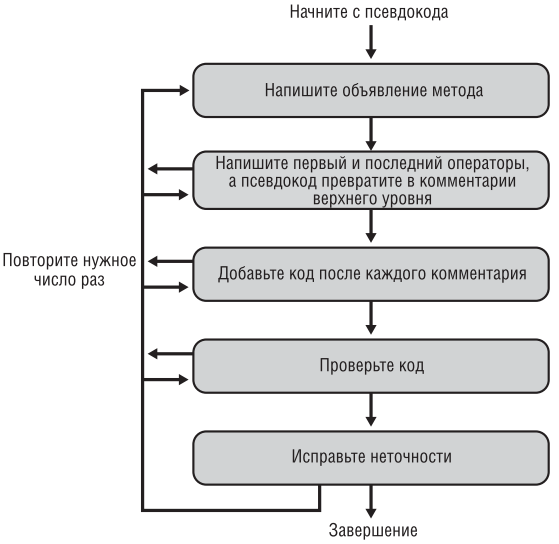



***Рис. 9-3.*** *Вы пройдете все эти этапы, но не обязательно именно в такой последовательности*

***Объявление метода*** Напишите интерфейсный оператор метода: объявление функции на C++, метода на Java, функции или подпрограммы на Microsoft Visual Basic и т. д. в зависимости от применяемого языка. Превратите существующий

заголовочный комментарий в комментарий соответствующего языка и оставьте его над уже написанным псевдокодом. Вот интерфейсный оператор и заголовок нашего примера на C++:

**Пример интерфейса метода и заголовка, добавленных к псевдокоду (C++)**

Это заголовочный комментарий, превращенный в комментарий C++.

```
/* Этот метод выводит сообщение об ошибке на основании кода ошибки, получаемого от
вызывающей программы. Способ вывода сообщения зависит от режима работы, который он
определяет сам. Он возвращает значение, указывающее на успешное завершение или сбой.
*/
```

Это интерфейсный оператор.

```
Status  ReportErrorMessage(
    ErrorCode errorToReport
    )
Установить статус по умолчанию в “сбой”.
Найти сообщение, соответствующее коду ошибки.

Если код ошибки корректен
    Если работа в интерактивном режиме, вывести сообщение
    и указать успешный статус.

    Если работа в режиме командной строки, запротоколировать
    сообщение об ошибке и указать успешный статус.

Если код ошибки некорректен, информировать пользователя
об обнаружении внутренней ошибки.

Вернуть статус.
```

***Изменение псевдокода на высокоуровневые комментарии*** Добавим первый и последний оператор: *{* и *}* на C++ и превратим псевдокод в комментарии:

**Пример первого и последнего оператора вокруг псевдокода (C++)**

```
/* Этот метод выводит сообщение об ошибке на основании кода ошибки, получаемого от
вызывающей программы. Способ вывода сообщения зависит от режима работы, который он
определяет сам. Он возвращает значение, указывающее на успешное завершение или сбой.
*/

Status  ReportErrorMessage(
    ErrorCode errorToReport
    ) {
```

С этого места предложения псевдокода заменены на комментарии C++.

```
    // Установить статус по умолчанию в “сбой”.
    // Найти сообщение, соответствующее коду ошибки.
    // Если код ошибки корректен
        // Если работа в интерактивном режиме, вывести сообщение
        // и указать успешный статус.
```

```
        // Если работа в режиме командной строки, запротоколировать
        // сообщение об ошибке и указать успешный статус.

    // Если код ошибки некорректен, информировать пользователя
    // об обнаружении внутренней ошибки.

    // Вернуть статус.
}
```

Теперь роль метода очевидна. Проектные работы закончены, и вы без всякого кода видите, как работает метод.

**Перекрестная ссылка** В этом случае подходит литературная метафора, о которой см. подраздел «Литературная метафора: написание кода» раздела 2.3.

***Напишите код под каждым комментарием*** Добавьте код под каждой строкой комментария. Это напоминает написание курсовой работы: сначала вы пишете план, а затем, под каждым его пунктом, — абзац текста. Каждый комментарий соответствует блоку или абзацу кода. Длина абзаца кода, как и длина абзаца литературного текста, зависит от высказываемой мысли, а его качество — от понимания автором сути.

### Пример добавления кода к комментариям (C++)

```
/* Этот метод выводит сообщение об ошибке на основании кода ошибки, получаемого от
вызывающей программы. Способ вывода сообщения зависит от режима работы, который он
определяет сам. Он возвращает значение, указывающее на успешное завершение или сбой.
*/

Status ReportErrorMessage(
    ErrorCode errorToReport
    ) {
    // Установить статус по умолчанию в "сбой".
```

Добавленный код.

```
    Status errorMessageStatus = Status_Failure;

    // Найти сообщение, соответствующее коду ошибки.
```

Новая переменная *errorMessage*.

```
    Message errorMessage = LookupErrorMessage( errorToReport );

    // Если код ошибки корректен.
        // Если работа в интерактивном режиме, вывести сообщение
        // и указать успешный статус.

        // Если работа в режиме командной строки, запротоколировать
        // сообщение об ошибке и указать успешный статус.

    // Если код ошибки некорректен, информировать пользователя
    // об обнаружении внутренней ошибки.

    // Вернуть статус.
}
```

Это только начало написания кода. Поскольку используется переменная *errorMessage*, ее нужно объявить. Если вы вносите комментарии после написания кода, двух строк комментария на две строки кода почти всегда будет достаточно. При данном же подходе важно семантическое содержание комментариев, а не число строк кода, к которым они относятся. Комментарии уже есть и описывают действия кода, так что оставьте их все.

Далее нужно добавить код ко всем оставшимся комментариям:

**Пример законченного метода, созданного посредством Процесса Программирования Псевдокода (C++)**

```
/* Этот метод выводит сообщение об ошибке на основании кода ошибки, получаемого от
вызывающей программы. Способ вывода сообщения зависит от режима работы, который он
определяет сам. Он возвращает значение, указывающее на успешное завершение или сбой.
*/

Status  ReportErrorMessage(
    ErrorCode errorToReport
    ) {
    // Установить статус по умолчанию в “сбой”.
    Status errorMessageStatus = Status_Failure;

    // Найти сообщение, соответствующее коду ошибки.
    Message errorMessage = LookupErrorMessage( errorToReport );

    // Если код ошибки корректен.
```

Отсюда начинаем добавлять код для каждого комментария.

```
    if ( errorMessage.ValidCode() ) {
        // Определяем метод обработки.
        ProcessingMethod errorProcessingMethod = CurrentProcessingMethod();

        // Если работа в интерактивном режиме, вывести сообщение
        // и указать успешный статус.
        if ( errorProcessingMethod == ProcessingMethod_Interactive ) {
            DisplayInteractiveMessage( errorMessage.Text() );
            errorMessageStatus = Status_Success;
        }

        // Если работа в режиме командной строки, запротоколировать
        // сообщение об ошибке и указать успешный статус.
```

Этот код — хороший кандидат стать новым методом: *DisplayCommandLineMessage*().

```
        else if ( errorProcessingMethod == ProcessingMethod_CommandLine ) {
            CommandLine messageLog;
            if ( messageLog.Status() == CommandLineStatus_Ok ) {
                messageLog.AddToMessageQueue( errorMessage.Text() );
                messageLog.FlushMessageQueue();
                errorMessageStatus = Status_Success;
            }
```

Эти код и комментарий новые и являются результатом развертывания оператора *if*.

```
        else {
            // Не можем ничего делать, так как процедура
            // сама занимается обработкой ошибки.
        }
```

Это тоже новый код и комментарий.

```
    else {
        // Не можем ничего делать, так как процедура
        // сама занимается обработкой ошибки.
    }
}

// Если код ошибки некорректен, извещаем пользователя
// об обнаружении внутренней ошибки.
else {
    DisplayInteractiveMessage(
        "Internal Error: Invalid error code in ReportErrorMessage()"
    );
}

// Вернуть статус.
return errorMessageStatus;
}
```

К каждому комментарию добавлена одна или несколько строк кода. Каждый блок кода выражает некоторое намерение, описанное комментариями. Все переменные объявлены и определены рядом с местом их первого использования. Каждый комментарий обычно разворачивается в 2–10 строк кода.

Теперь вернемся к спецификации и псевдокоду. Первоначальная спецификация из пяти предложений превратилась в 15 строк псевдокода, которые в свою очередь развернуты в метод размером в страницу. Хотя спецификация и была достаточно подробной, создание метода потребовало проектировочных работ при написании псевдокода и кодировании. Это низкоуровневое проектирование и есть одна из причин, по которой «кодирование» является нетривиальной задачей.

***Проверьте, не нужна ли дальнейшая декомпозиция кода*** В некоторых случаях вы увидите, что код, соответствующий одной изначальной строке псевдокода, существенно разросся. В таких ситуациях следует предпринять одно из следующих действий.

**Перекрестная ссылка** О рефакторинге см. главу 24.

- Преобразуйте код, соответствующий комментарию, в новый метод. Дайте методу имя и напишите код вызова этого метода. Если вы правильно применяли ППП, имя метода вы легко придумаете на основе псевдокода. Закончив работу с изначальным кодом, переходите к вновь созданным методам и применяйте ППП к ним.
- Применяйте ППП рекурсивно. Вместо того чтобы писать несколько десятков строк кода для одной строки псевдокода, разбейте эту строку псевдокода на несколько предложений и для каждой из них напишите код.

## Проверка кода

Третий шаг после проектирования и реализации метода — его проверка. Все ошибки, которые вы пропустите на этом этапе, вы сможете обнаружить лишь при позднейшем тестировании, что обойдется вам дороже.

Ошибка может не проявиться до окончательного кодирования по нескольким причинам. Ошибка в псевдокоде может стать заметнее при детальной реализации. Конструкция, выглядящая элегантно в псевдокоде, на языке программирования может стать топорной. Проработка детальной реализации может выявить ошибку архитектуры, проекта или требований. Наконец, код может содержать самые банальные ошибки программирования — никто не совершенен! По всем этим причинам пересмотрите код, прежде чем двигаться дальше.

**Перекрестная ссылка** О поиске ошибок при построении архитектуры и выработке требований см. главу 3.

***Умозрительно проверьте ошибки в методе*** Первая формальная проверка метода — умозрительная. Мысленно выполните все ветви метода. Сделать это непросто, что и является одной из причин писать короткие методы. Проверьте все возможные ветви и исключительные условия. Проделайте это сами и с коллегами.

Одно из основных различий между любителями и профессиональными программистами — различие, появляющееся при переходе от суеверия к пониманию. Под суеверием здесь я понимаю не иллюзию, что программа выдает больше ошибок в полнолуние, а замену «прочувствования» программы ее пониманием. Если вы часто обнаруживаете, что подозреваете компилятор или аппаратные средства в ошибке, вы в плену суеверий. Давнишние исследования показали, что только около 5% всех ошибок связано с аппаратурой, компиляторами или ОС (Ostrand and Weyuker, 1984). Сейчас этот процент, видимо, еще меньше. Программист, достигший сферы понимания, обращает внимание прежде всего на свое творение, являющееся потенциальным источником 95% ошибок. Нужно знать роль каждой строки своей программы. Ничто не может называться верным только потому, что выглядит работоспособным. Если вы не знаете, почему это работает, вероятно, оно и не работает на самом деле.

Итог: работающий метод — это еще не все. Если вы не знаете, как он работает, изучайте его, обсуждайте его, экспериментируйте с альтернативными вариантами, пока не добьетесь понимания.

***Компиляция метода*** Проверив метод, скомпилируйте его. Может показаться неэффективным так долго откладывать компиляцию. Вероятно, вы уменьшите себе работу, скомпилировав метод ранее и позволив компилятору проверить необъявленные переменные, обнаружить конфликты имен и т. д.

Между тем, отложив трансляцию на более поздний срок, вы получите ряд преимуществ. Основная причина в том, что, когда вы компилируете новый код, в вас начинают тикать внутренние часики. После первой трансляции появляется мысль: «Еще всего одна компиляция, и дело сделано». Этот синдром «еще всего одной компиляции» подвигает вас к скороспелым, чреватым ошибками изменениям, которые в долгосрочном плане увеличивают общее время работы. Не спешите и не компилируйте программу, пока не будете уверены, что она верна.

Вот несколько советов по эффективной компиляции.

- Установите наивысший уровень предупреждений компилятора. Вы можете отловить изрядное число ошибок, лишь позволив компилятору их обнаруживать.
- Применяйте проверяющие средства. Встроенные проверяющие средства компиляторов могут быть дополнены внешними, такими как lint для C. Даже некомпилируемый код, скажем, HTML и JavaScript, можно проверить соответствующими утилитами.
- Выясните причину всех сообщений об ошибках и предупреждений. Подумайте, что сообщение говорит о вашем коде. Многие предупреждения зачастую указывают на низкое качество кода, и вам следует попытаться понять смысл каждого предупреждения. На практике предупреждения, которые вы видите раз за разом, вызывают одну из двух реакций: вы или не обращаете на них внимания, и они скрывают от вашего взгляда более важные предупреждения, или они попросту вас раздражают. Как правило, проще и безболезненней переписать код, решив проблему, вызывающую предупреждения, и, таким образом, избавиться от них.

**Перекрестная ссылка** Подробности см. в главе 22 и подразделе «Создание лесов для тестирования отдельных классов» раздела 22.5.

***Пройдите по коду отладчиком*** Скомпилировав метод, запустите его в отладчике и пройдите по каждой строке кода. Убедитесь, что каждая строка выполняется так, как вы ожидаете. Следуя этому простому совету, вы сможете найти много ошибок.

***Протестируйте код*** Протестируйте код, используя тестовые примеры, которые вы запланировали или уже создали при разработке метода. Возможно, вы создали леса для поддержки тестирования, т. е. код, поддерживающий методы при тестировании и не включаемый в конечный продукт. Леса могут представлять собой методы — тестовые сбруи, которые вызывают ваш метод с тестовыми данными, или заглушки, вызываемые вашим методом.

**Перекрестная ссылка** Подробности см. в главе 23.

***Удалите ошибки из метода*** Обнаруженные ошибки нужно удалить. Если к этому моменту ваш метод работает нестабильно, велика вероятность, что он таким и останется. Обнаружив непонятное поведение метода, начните все заново. Не пытайтесь доводить его до ума — перепишите его. Изощренные переделки обычно говорят о неполном понимании и гарантируют возникновение ошибок как сейчас, так и в будущем. Полное перепроектирование нестабильного метода полностью оправданно. Мало что сравнится по эффективности с переписыванием проблемного метода — вы позабудете о бывших ошибках.

## Наведение глянца

Проверив код, оцените его с учетом общих критериев, описанных в этой книге. Чтобы гарантировать соответствие качества метода высоким стандартам, сделайте следующее.

- Проверьте интерфейс метода. Убедитесь, что применяются все входные и выходные данные и используются все параметры (см. раздел 7.5).

- Проверьте общее качество конструкции. Убедитесь, что метод выполняет единственную задачу и делает это хорошо, имея в виду его слабое сопряжение с другими методами и проектирование в соответствии с методикой защитного программирования (см. главу 7).
- Проверьте переменные метода: корректность их именования, неиспользуемые объекты, необъявленные переменные, неверно инициализированные объекты и т. д. (см. главы 10–13).
- Проверьте логику метода. Проанализируйте наличие ошибок занижения/завышения на 1, некорректной вложенности и утечки ресурсов (см. главы 14–19).
- Проверьте форматирование метода. Убедитесь в корректном использовании пробелов для структурирования метода, выражений и списка параметров (см. главу 31).
- Проверьте документирование метода. Убедитесь в корректности псевдокода, переведенного в комментарии. Проверьте описание алгоритма, документирование интерфейса, неочевидных зависимостей и нестандартных подходов (см. главу 32).
- Удалите лишние комментарии. Иногда комментарии, полученные из псевдокода, являются избыточными, особенно когда ППП применяется рекурсивно и комментарии лишь описывают вызов метода, назначение которого и так понятно из его имени.

### Повторите нужное число раз

Если качество метода неудовлетворительное, вернитесь к псевдокоду. Создание высококачественного ПО — итеративный процесс, так что без колебаний повторяйте весь цикл конструирования вновь и вновь.

## 9.4. Альтернативы ППП

Для меня ППП — идеальная методика создания классов и методов. Другие специалисты рекомендуют иные подходы. Вы можете применять их как альтернативу или дополнение ППП.

***Разработка с изначальными тестами*** Это популярный стиль разработки, при котором тестовые задания пишутся до самого кода (см. раздел 22.2). Есть хорошая книга Кента Бека на эту тему — «Test-Driven Development: By Example» (Beck, 2003).

***Рефакторинг*** Рефакторинг — это подход к разработке с усовершенствованием кода посредством последовательности семантически корректных преобразований. Программисты пользуются шаблонами плохого кода или «запахами» (smells) для выявления разделов кода, подлежащих усовершенствованию. Этот подход подробно описан в главе 24, а также в книге Мартина Фаулера «Refactoring: Improving the Design of Existing Code» (Fowler, 1999).

***Проектирование по контракту*** Это подход предполагает, что каждый метод имеет пред- и постусловия (см. раздел 8.2). Лучший источник информации на эту тему — книга Бертрана Мейера «Object-Oriented Software Construction» (Meyer, 1997).

***Бессистемное программирование*** Некоторые программисты лепят программу как попало, а не используют тот или иной систематический подход, например ППП. Если вы не понимаете, что делать дальше, это признак того, что надо переходить на ППП. Не было ли у вас такого, чтобы вы забывали написать часть класса или метода? Вряд ли такое могло случиться при применении ППП. Если вы глядите на экран и не знаете, с чего начать, пора начать ППП, который сделает вашу программистскую долю проще.

http://cc2e.com/0943

**Перекрестная ссылка** Назначение этого списка — проверить, применяете ли вы правильные методики при создании методов. Контрольный список по качеству методов как таковых см. в главе 7.

### Контрольный список: Процесс Программирования с Псевдокодом

- ❑ Проверили ли вы, что удовлетворяются предварительные условия?
- ❑ Определена ли проблема, которую решает класс?
- ❑ Достаточно ли понятна высокоуровневая конструкция, чтобы дать классам и методам адекватные имена?
- ❑ Продумали ли вы тестирование класса и каждого из его методов?
- ❑ Рассматривали ли вы эффективность с позиции стабильных интерфейсов и понятной реализации или с позиции соответствия ресурсам и бюджету?
- ❑ Проанализировали ли вы стандартные и другие библиотеки на предмет наличия подходящих методов и компонентов?
- ❑ Просмотрели ли вы литературу в поисках полезных алгоритмов?
- ❑ Проектировали ли вы каждый метод с использованием подробного псевдокода?
- ❑ Проверили ли вы псевдокод умозрительно? Легко ли его понять?
- ❑ Уделили ли вы внимание предупреждениям, которые указывают на необходимость перепроектирования (использование глобальных данных, операции, которые лучше перенести в другой класс или метод и т. д.)?
- ❑ Точно ли вы перевели псевдокод в код?
- ❑ Рекурсивно ли вы применяли ППП, разбивая методы на более мелкие при необходимости?
- ❑ Документировали ли вы принимаемые допущения?
- ❑ Убрали ли вы избыточные комментарии?
- ❑ Проделали ли вы несколько итераций или остановились после первой?
- ❑ Вполне ли вы понимаете свой код? Легко ли в нем разобраться?

## Ключевые моменты

- Конструирование классов и методов — процесс итеративный. Особенности, замечаемые при конструировании отдельных методов, заставляют возвращаться к проектированию класса.
- Написание хорошего псевдокода предполагает употребление понятного естественного языка без специфических особенностей конкретного языка программирования, а также формулировок на уровне намерений (описания сути конструкции, а не способов ее работы).

- Процесс Программирования с Псевдокодом — полезный инструмент детального проектирования, упрощающий кодирование. Псевдокод транслируется непосредственно в комментарии, гарантируя их адекватность и полезность.
- Не останавливайтесь на первой придуманной вами конструкции — испробуйте несколько подходов и выберите лучший, прежде чем писать код.
- Проверяйте свою работу на каждом шаге и просите об этом других. При этом вы отловите ошибки на наименее дорогостоящем уровне, когда вы вложили в работу меньше усилий.

Часть III

# ПЕРЕМЕННЫЕ

ГЛАВА 10

# Общие принципы использования переменных

http://cc2e.com/1085

**Содержание**



**Связанные темы**

- Именование переменных: глава 11
- Фундаментальные типы данных: глава 12
- Редкие типы данных: глава 13
- Размещение объявлений данных: одноименный подраздел раздела 31.5
- Документирование переменных: подраздел «Комментирование объявлений данных» раздела 32.5

Если при конструировании приходится заполнять небольшие пробелы в требованиях и архитектуре, это нормально и даже желательно. Проектирование программы вплоть до микроскопических деталей было бы неэффективным. В этой главе рассматривается один из низкоуровневых вопросов конструирования — использование переменных.

Эта глава будет особенно полезна опытным программистам. Довольно часто мы применяем рискованные подходы, не имея полного представления об альтернативах, а после используем их в силу привычки. Особый интерес для опытных программистов могут представлять разделы 10.6 и 10.8, посвященные соответствен-

но времени связывания и использованию переменных только с одной целью. Если вы не знаете, насколько «опытным» программистом вы являетесь, пройдите «Тест на знание типов данных» в разделе 10.1 и узнайте.

В этой главе я буду понимать под «переменными» и объекты, и встроенные типы данных, такие как целые числа и массивы. «Типами данных» я, как правило, буду называть встроенные типы данных, а просто «данными» — и встроенные типы данных, и объекты.

# 10.1. Что вы знаете о данных?

Создавая данные, вы должны в первую очередь понять, какие именно данные вам нужны. Обширные знания о разных типах данных — важнейший компонент в инструментарии любого программиста. Введения в типы данных в этой книге вы не найдете, но «Тест на знание данных» поможет вам определить, сколько еще вам нужно о них узнать.

## Тест на знание данных

Поставьте себе *1* балл за каждый термин, который вам известен. Если термин кажется знакомым, но вы не уверены в его значении, поставьте себе *0,5* балла. Выполнив тест, просуммируйте баллы и оцените результат по приведенному ниже описанию.

_____ abstract data type (абстрактный тип данных)
_____ array (массив)
_____ bitmap (растровое изображение)
_____ boolean variable (булева переменная)
_____ B-tree (B-дерево)
_____ character variable (символьная переменная)
_____ container class (класс-контейнер)
_____ double precision (двойная точность)
_____ elongated stream (удлиненный поток)
_____ enumerated type (перечисление)
_____ floating point (число с плавающей точкой)
_____ heap (куча)
_____ index (индекс)
_____ integer (целое число)
_____ linked list (связанный список)
_____ named constant (именованная константа)
_____ literal (литерал)
_____ local variable (локальная переменная)
_____ lookup table (таблица поиска)
_____ member data (данные-члены)
_____ pointer (указатель)
_____ private (закрытый)
_____ retroactive synapse (ретроактивный синапс)
_____ referential integrity (целостность ссылочных данных)
_____ stack (стек)
_____ string (строка)
_____ structured variable (структурная переменная)
_____ tree (дерево)
_____ typedef (синоним типа)
_____ union (объединение)
_____ value chain (цепочка начисления стоимости)
_____ variant (универсальный тип)

_____ **Общее число баллов**

Интерпретировать результаты можно примерно так.

0–14 Вы — начинающий программист; возможно, вы учитесь на первом курсе института или самостоятельно изучаете свой первый язык программирования. Вы можете многое узнать, прочитав одну из книг, перечисленных в следующем подразделе. Вернувшись к этой части книги потом, вы извлечете больше пользы, потому что этот материал адресован более опытным программистам.

15–19 Вы — программист среднего уровня или опытный программист, который многое забыл. Из книг, указанных чуть ниже, вы также сможете извлечь выгоду, хотя многие концепции будут вам знакомы.

20–24 Вы — эксперт в программировании. Вероятно, на вашей полке уже стоят книги, указанные ниже.

25–29 Вы знаете о типах данных больше, чем я. Может, напишете собственную книгу (пришлите мне экземпляр!)?

30–32 Вы — тщеславный мошенник. Термины «удлиненный поток», «ретроактивный синапс» и «цепочка начисления стоимости» не имеют никакого отношения к типам данных — я их выдумал! Прочитайте раздел «Профессиональная честность» в главе 33!

### Дополнительные ресурсы

Хорошими источниками информации о типах данных являются следующие книги:

Cormen, H. Thomas, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*. New York, NY: McGraw Hill. 1990.

Sedgewick, Robert. *Algorithms in C++, Parts 1-4*, 3d ed. Boston, MA: Addison-Wesley, 1998.

Sedgewick, Robert. *Algorithms in C++, Part 5*, 3d ed. Boston, MA: Addison-Wesley, 2002.

## 10.2. Грамотное объявление переменных

**Перекрестная ссылка** О форматировании объявлений переменных см. одноименный подраздел раздела 31.5, а о документировании — подраздел «Комментирование объявлений данных» раздела 32.5.

В этом разделе описаны способы оптимизации объявления переменных. Строго говоря, это не такая уж и крупная задача, и вы могли бы подумать, что она не заслуживает собственного раздела. И все же создавать переменные приходится очень часто, и приобретение правильных привычек поможет вам сэкономить время и исключить ненужные разочарования.

### Неявные объявления

Некоторые языки поддерживают неявное объявление переменных. Так, если, программируя на Microsoft Visual Basic, вы попытаетесь использовать необъявленную переменную, компилятор может автоматически объявить ее для вас (это зависит от параметров компилятора).

Неявное объявление переменных — одна из самых опасных возможностей языка. Если вы программировали на Visual Basic, то знаете, как жаль времени, потраченного на поиск причины неправильного значения *acctNo*, если в итоге обнаруживается, что вы по ошибке вызвали переменную *acctNum*, которая была инициализирована нулем. Если язык не заставляет объявлять переменные, подобную ошибку допустить очень легко.

Если язык требует объявления переменных, для столкновения с данной проблемой нужно сделать две ошибки: во-первых, использовать в теле метода и *acctNum*, и *acctNo*, ну, а во-вторых, объявить в методе обе эти переменные. Такую ошибку допустить сложнее, что практически устраняет проблему похожих имен переменных. По сути языки, требующие явного объявления переменных, заставляют более внимательно использовать данные, что является одним из важнейших достоинств таких языков. А если язык поддерживает неявные объявления? Несколько советов я привел ниже.

***Отключите неявные объявления*** Некоторые компиляторы позволяют запретить неявные объявления. Например, в Visual Basic для этого служит директива *Option Explicit On*, которая заставляет объявлять все используемые переменные.

***Объявляйте все переменные*** Печатая имя новой переменной, объявите ее, даже если компилятор этого не требует. Пусть не от всех, но от некоторых ошибок это вас избавит.

***Используйте конвенции именования*** Чтобы не использовать две переменные там, где предполагается одна, задайте конвенцию употребления популярных суффиксов в именах переменных. Скажем, конвенция может требовать применения директивы *Option Explicit On* и суффикса *No*.

**Перекрестная ссылка** О стандартизации сокращений имен см. подраздел «Общие советы по сокращению имен» раздела 11.6.

***Проверяйте имена переменных*** Просматривайте список перекрестных ссылок, сгенерированный компилятором или утилитой. Многие компиляторы составляют список всех переменных метода, что, например, позволяет узнать о неумышленном применении двух переменных с похожими именами. Кроме того, компиляторы могут указывать на объявленные, но неиспользованные переменные.

## 10.3. Принципы инициализации переменных

Неверная инициализация данных — один из самых плодородных источников ошибок в программировании. Эффективные способы предотвращения проблем с инициализацией могут значительно ускорить отладку.

При неверной инициализации проблемы объясняются тем, что переменная имеет не то первоначальное значение, которое вы ожидаете. Это может случиться по одной из следующих причин.

- Переменной не было присвоено значения. Она имеет то случайное значение, которое находилось в соответствующих ячейках памяти при запуске программы.
- Значение переменной устарело. Когда-то переменной было присвоено значение, но оно утратило свою актуальность.
- Одним частям переменной были присвоены значения, а другим нет.

**Перекрестная ссылка** О тестировании, основанном на шаблонах инициализации данных и их использования, см. подраздел «Тестирование, основанное на потоках данных» раздела 22.3.

Последняя причина имеет несколько вариаций. Вы можете инициализировать несколько членов объекта, но не все. Вы можете забыть выделить память и инициализировать «переменную», на которую указывает неинициализированный указатель. При этом на самом деле вы занесете некоторое значение в случайный блок памяти. Им может оказаться область памяти, содержащая данные. Им может ока-

заться область памяти, содержащая код. В этом блоке может находиться фрагмент ОС. Проблема с указателями может проявляться совершенно неожиданным образом, изменяющимся от случая к случаю, поэтому найти такие ошибки сложнее, чем любые другие.

Ниже описаны способы предотвращения проблем, связанных с инициализацией.

***Инициализируйте каждую переменную при ее объявлении*** Инициализация переменных при их объявлении — простая методика защитного программирования и хорошая страховка от ошибок инициализации. Так, следующий код позволяет гарантировать, что инициализация массива *studentGrades* будет выполняться при каждом вызове метода, содержащего массив:

**Пример инициализации массива при его объявлении (C++)**

```
float studentGrades[ MAX_STUDENTS ] = { 0.0 };
```

**Перекрестная ссылка** Проверка входных параметров является еще одной формой защитного программирования (см. главу 8).

***Инициализируйте каждую переменную там, где она используется в первый раз*** Visual Basic и некоторые другие языки не позволяют инициализировать переменные при их объявлении. В результате код может принимать вид, при котором сначала выполняется объявление нескольких переменных, а потом эти переменные инициализируются — и то и другое происходит вдали от места фактического использования переменных в первый раз:

**Пример плохой инициализации переменных (Visual Basic)**

```
‘ объявление всех переменных
Dim accountIndex As Integer
Dim total As Double
Dim done As Boolean

’ инициализация всех переменных
accountIndex = 0
total = 0.0
done = False
...

’ использование переменной accountIndex
...

’ использование переменной total
...

’ использование переменной done
While Not done
   ...
```

Лучше инициализировать каждую переменную как можно ближе к месту первого обращения к ней:

**Пример хорошей инициализации переменных (Visual Basic)**

```
Dim accountIndex As Integer
accountIndex = 0
' использование переменной accountIndex
...

Dim total As Double
```

Переменная *total* объявляется и инициализируется непосредственно перед ее использованием.

```
total = 0.0
' использование переменной total
...

Dim done As Boolean
```

Переменная *done* также объявляется и инициализируется непосредственно перед ее использованием.

```
done = False
' использование переменной done
While Not done
    ...
```

Второй вариант лучше первого по нескольким причинам. Пока выполнение первого примера дойдет до фрагмента, в котором используется переменная *done*, она может оказаться измененной. Даже если при написании программы это не так, нельзя гарантировать, что этого не произойдет после нескольких ее изменений. Кроме того, в первом примере все переменные инициализируются в одном месте, из-за чего создается впечатление, что все они используются на протяжении всего метода, тогда как на самом деле переменная *done* вызывается только в его конце. Наконец, в результате изменений программы (которые неизбежно придется вносить, и не только при отладке) код, использующий переменную *done*, может оказаться заключенным в цикл, при этом переменную каждый раз нужно будет инициализировать заново. Во второй пример в этом случае придется внести лишь небольшое изменение. Первый пример слабее защищен от досадных ошибок инициализации.

Два этих фрагмента иллюстрируют Принцип Близости: группируйте связанные действия вместе. Этот принцип предполагает также близость комментариев к описываемому ими коду, близость кода настройки цикла к самому циклу, группировку команд в линейных участках программы и т. д.

**Перекрестная ссылка** О группировке связанных действий см. раздел 10.4.

***В идеальном случае сразу объявляйте и определяйте каждую переменную непосредственно перед первым обращением к ней*** При объявлении переменной вы указываете ее тип. При определении вы присваиваете ей конкретное значение. Если язык позволяет (к таким языкам относятся, например, C++ и Java), переменную следует объявлять и определять перед фрагментом, в котором она используется впервые. В идеале каждую переменную следует определять при ее объявлении:

**Пример хорошей инициализации переменных (Java)**

```
int accountIndex = 0;
// использование переменной accountIndex
...

Переменная total инициализируется непосредственно перед ее использованием.

double total = 0.0;
// использование переменной total
...

Переменная done также инициализируется непосредственно перед ее использованием.

boolean done = false;
// использование переменной done
 while ( ! done ) {
   ...
```

**Перекрестная ссылка** О группировке связанных действий см. также раздел 14.2.

***Объявляйте переменные по мере возможности как* final *или* const** Объявив переменную как *final* в Java или *const* в C++, вы можете предотвратить изменение ее значения после инициализации. Ключевые слова *final* и *const* полезны для определения констант класса, исключительно входных параметров и любых локальных переменных, значения которых должны оставаться неизменными после инициализации.

***Уделяйте особое внимание счетчикам и аккумуляторам*** Переменные *i*, *j*, *k*, *sum* и *total* часто играют роль счетчиков или аккумуляторов. Нередко программисты забывают обнулить счетчик или аккумулятор перед его использованием в очередной раз.

***Инициализируйте данные-члены класса в его конструкторе*** Подобно переменным метода, которые следует инициализировать при вызове каждого метода, данные класса следует инициализировать в его конструкторе. Если в конструкторе выделяется память, в деструкторе ее следует освободить.

***Проверяйте необходимость повторной инициализации*** Спросите себя, нужно ли будет когда-нибудь инициализировать переменную повторно: например, для применения в цикле или для переустановки ее значения между вызовами метода. Если да, убедитесь, что команда инициализации входит в повторяющийся фрагмент кода.

***Инициализируйте именованные константы один раз; переменные инициализируйте в исполняемом коде*** Если переменные служат для имитации именованных констант, вполне допустимо инициализировать их один раз при запуске программы. Инициализируйте их в методе *Startup()*. Истинные переменные инициализируйте в исполняемом коде неподалеку от места их вызова. Очень часто метод, который первоначально применялся один раз, после изменения программы вызывается многократно. Переменные, которые инициализируются в методе *Startup()* уровня программы, не будут инициализироваться повторно.

***Настройте компилятор так, чтобы он автоматически инициализировал все переменные*** Если ваш компилятор поддерживает такую возможность, заставьте его автоматически инициализировать все переменные. Однако, полагаясь на компилятор, вы можете столкнуться с проблемами при переносе кода на другой компьютер или при использовании другого компилятора. Документируйте использование параметров компилятора — без такой документации предположения, основанные на конкретных параметрах компилятора, определить очень трудно.

***Внимательно изучайте предупреждения компилятора*** Многие компиляторы предупреждают об использовании неинициализированных переменных.

***Проверяйте корректность входных параметров*** Это еще один эффективный способ предотвращения ошибок инициализации. Прежде чем присвоить входные значения чему-либо, убедитесь, что они допустимы.

**Перекрестная ссылка** О проверке входных параметров см. главу 8, преимущественно раздел 8.1.

***Используйте утилиту проверки доступа к памяти для обнаружения неверно инициализированных указателей*** Некоторые ОС сами следят за корректностью обращений к памяти, выполняемых при помощи указателей, другие оставляют вас на произвол судьбы. Тогда можно приобрести инструмент проверки доступа к памяти и проконтролировать использование указателей в своей программе.

***Инициализируйте рабочую память при запуске программы*** Инициализация рабочей памяти известным значением облегчает поиск ошибок инициализации. Этого позволяют достичь описанные ниже подходы.

- Вы можете использовать специализированную утилиту для заполнения памяти определенным значением перед запуском программы. Для некоторых целей хорошо подходит значение 0, потому что оно гарантирует, что неинициализированные указатели будут указывать на нижнюю область памяти, благодаря чему их будет относительно легко найти. В случае процессоров с архитектурой Intel целесообразно заполнить память значением 0xCC, потому что оно соответствует машинному коду команды точки прерывания; если вы запустите код в отладчике и попытаетесь выполнить данные, а не код, вы потонете в точках прерывания. Еще одно достоинство значения 0xCC в том, что его легко заметить в дампах памяти; кроме того, оно редко используется. По этим же причинам Брайан Керниган и Роб Пайк предлагают заполнять память константой 0xDEADBEEF[1] (Kernighan and Pike, 1999).
- Если вы применяете утилиту заполнения памяти, можете время от времени изменять используемое ей значение. «Встряхивание» программы иногда позволяет обнаружить проблемы, которые остаются скрытыми, если среда никогда не изменяется.
- Вы можете сделать так, чтобы программа инициализировала свою рабочую память при запуске. Цель заполнения памяти до запуска программы — обнаружение дефектов, тогда как цель этого подхода — их сокрытие. Заполняя рабочую память каждый раз одинаковым значением, вы сможете гарантировать, что программа не будет зависеть от случайных изменений начальной конфигурации памяти.

---

[1] Букв. «мертвая корова». — *Прим. перев.*

# 10.4. Область видимости

Область видимости можно понимать как «известность» переменной в программе. Областью видимости называют фрагмент программы, в котором переменная известна и может быть использована. Переменная с ограниченной или небольшой областью видимости известна только в небольшом фрагменте программы: в качестве примера можно привести индекс, используемый в теле одного небольшого цикла. Переменная с большой областью видимости известна во многих местах программы: примером может служить таблица с данными о сотрудниках, используемая по всей программе.

В разных языках реализованы разные подходы к области видимости. В некоторых примитивных языках все переменные глобальны. В этом случае вы не имеете контроля над областью видимости переменных, что создает много проблем. В C++ и похожих языках переменная может иметь область видимости, соответствующую блоку (фрагменту кода, заключенному в фигурные скобки), методу, классу (возможно, и производным от него классам) или всей программе. В Java и C# переменная может также иметь область видимости, соответствующую пакету или пространству имен (набору классов).

Ниже я привел ряд советов, относящихся к области видимости.

## Локализуйте обращения к переменным

Код, расположенный между обращениями к переменной, является «окном уязвимости». Чем больше это окно, тем выше вероятность, что в его пределах будет добавлен новый код, искажающий значение переменной, и тем труднее следить за значением переменной при чтении кода. Поэтому обращения к переменной всегда целесообразно локализовать, группируя их вместе.

Идея локализации обращений к переменным самоочевидна, однако она допускает и формальную оценку. Одним из методов оценки степени сгруппированности обращений к переменной является определение «интервала» (span) между обращениями, например:

**Пример определения интервалов между обращениями к переменным (Java)**

```
a = 0;
b = 0;
c = 0;
a = b + c;
```

В данном случае между первым и вторым обращениями к *a* находятся две строки кода, поэтому и интервал равен 2. Между двумя обращениями к *b* — одна строка, что дает нам интервал, равный 1, ну а интервал между обращениями к *c* равен 0. Вот еще один пример:

**Пример интервалов, равных 1 и 0 (Java)**

```
a = 0;
b = 0;
c = 0;
b = a + 1;
b = b / c;
```

В этом примере между первым и вторым обращениями к *b* — одна строка кода, а между вторым и третьим обращениями строк нет, поэтому интервалы равны соответственно 1 и 0.

Средний интервал вычисляется путем усреднения отдельных интервалов. Так, во втором примере средний интервал между обращениями к *b* равен (1+0)/2, или 0,5. Локализовав обращения к переменным, вы позволите программисту, который будет читать ваш код, сосредоточиваться на меньшем фрагменте программы в каждый конкретный момент времени. Если обращения будут распределены по большему фрагменту кода, уследить за ними будет сложнее. Таким образом, главное преимущество локализации обращений к переменным в том, что оно облегчает чтение программы.

**Дополнительные сведения** Об интервалах между обращениями к переменным см. работу «Software Engineering Metrics and Models» (Conte, Dunsmore, and Shen, 1986).

## Делайте время жизни переменных как можно короче

С интервалом между обращениями к переменной тесно связано «время жизни» переменной — общее число строк, на протяжении которых переменная используется. Жизнь переменной начинается при первом обращении к ней, а заканчивается при последнем.

В отличие от интервала время жизни переменной не зависит от числа обращений к ней между первым и последним обращениями. Если переменная в первый раз вызывается в строке 1, а в последний — в строке 25, ее время жизни равно 25 строкам. Если переменная используется только в этих двух строках, средний интервал между обращениями к ней — 23 строки. Если бы между строками 1 и 25 переменная вызывалась в каждой строке, она имела бы средний интервал, равный 0, но время ее жизни по-прежнему равнялось бы 25 строкам. Связь интервалов между обращениями к переменной и времени ее жизни пояснена на рис. 10-1.

Как и интервал между обращениями к переменной, время ее жизни желательно делать как можно короче. Преимущество в обоих случаях одинаково: это уменьшает окно уязвимости, снижая вероятность неверного или неумышленного изменения переменной между действительно нужными обращениями к ней.

Второе преимущество короткого срока жизни: оно позволяет получить верное представление о коде. Если переменная изменяется в строке 10 и вызывается в строке 45, само пространство между двумя обращениями подразумевает, что переменная используется также между строками 10 и 45. Если переменная изменяется в строке 44 и вызывается в строке 45, других обращений к ней между этими строками быть не может, что позволяет вам сосредоточиться на меньшем фрагменте кода.

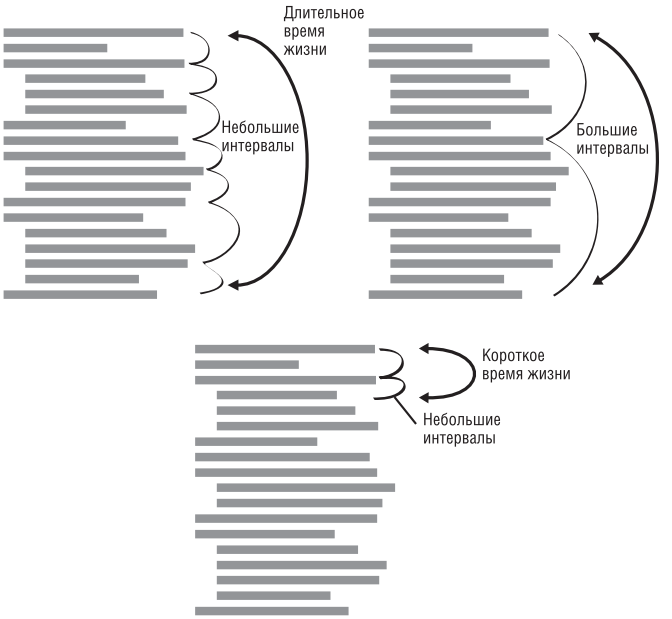



***Рис. 10-1.*** *«Длительное время жизни» подразумевает, что переменная используется в крупном фрагменте кода. При «коротком времени жизни» переменная используется лишь в небольшом фрагменте. «Интервал между обращениями» к переменной характеризует, насколько тесно сгруппированы обращения к переменной*

Короткое время жизни снижает вероятность ошибок инициализации. По мере изменения программы линейные участки кода имеют тенденцию превращаться в циклы, при этом программисты часто забывают про инициализацию переменных, выполненную вдали от цикла. Поддерживая код инициализации и код цикла в непосредственной близости, вы снизите вероятность того, что изменения приведут к ошибкам инициализации.

Кроме того, короткое время жизни облегчает чтение кода. Чем меньше строк кода нужно удерживать в уме в каждый конкретный момент времени, тем проще понять код. К тому же при небольшом времени жизни на экране помещаются сразу все обращения к переменной, что облегчает редактирование и отладку.

Наконец, короткое время жизни облегчает разделение крупного метода на меньшие. Если обращения к переменным сгруппированы в небольшом фрагменте, его проще выделить в отдельный метод.

### Оценка времени жизни переменной

Время жизни переменной можно формализовать, подсчитав число строк между первым и последним обращениями к ней (с учетом первой и последней строк). В следующем примере каждая из переменных обладает слишком долгим временем жизни:

#### Пример слишком долгого времени жизни переменных (Java)

```
1   // инициализация каждой переменной
2   recordIndex = 0;
3   total = 0;
```

```
4   done = false;
    ...
26  while ( recordIndex < recordCount ) {
27  ...
```

Последнее обращение к переменной *recordIndex*.

```
28      recordIndex = recordIndex + 1;
        ...

64  while ( !done ) {
        ...
```

Последнее обращение к переменной *total*.

```
69      if ( total > projectedTotal ) {
```

Последнее обращение к переменной *done*.

```
70          done = true;
```

Времена жизни переменных:

```
recordIndex  (строка 28 - строка 2 + 1) = 27
total  (строка 69 - строка 3 + 1) = 67
done  (строка 70 - строка 4 + 1) = 67
Среднее время жизни (27 + 67 + 67) / 3 »54
```

Следующий пример аналогичен предыдущему, только теперь обращения к переменным сгруппированы более тесно:

**Пример хорошего, короткого времени жизни переменных (Java)**

```
    ...
```

Инициализация переменной *recordIndex* ранее выполнялась в строке 3.

```
25  recordIndex = 0;
26  while ( recordIndex < recordCount ) {
27  ...
28      recordIndex = recordIndex + 1;
        ...
```

Инициализация переменных *total* и *done* ранее выполнялась в строках 4 и 5.

```
62  total = 0;
63  done = false;
64  while ( !done ) {
        ...
69      if ( total > projectedTotal ) {
70          done = true;
```

Теперь времена жизни переменных равны:

```
recordIndex  (строка 28 - строка 25 + 1) = 4
total  (строка 69 - строка 62 + 1) = 8
done  (строка 70 - строка 63 + 1) = 8
Среднее время жизни (4 + 8 + 8) / 3 »7
```

**Дополнительные сведения** О времени жизни переменных см. работу «Software Engineering Metrics and Models» (Conte, Dunsmore, and Shen, 1986).

Интуиция подсказывает, что второй вариант предпочтительнее, так как инициализация переменных выполняется ближе к месту их использования. Сравнение среднего времени жизни переменных — 54 и 7 — подкрепляет этот интуитивный вывод конкретными цифрами.

Какое время жизни считать приемлемым? А что можно сказать об интервале? Конкретных цифр у нас пока нет, но разумно предположить, что и интервал между обращениями к переменной, и время ее жизни следует пытаться свести к минимуму.

Если в этом ключе проанализировать глобальные переменные, окажется, что им соответствует огромный средний интервал между обращениями и такое же время жизни, — это одна из многих обоснованных причин избегать глобальных переменных.

## Общие советы по минимизации области видимости

Ниже даны конкретные рекомендации по минимизации области видимости.

**Перекрестная ссылка** Об инициализации переменных около места их использования см. раздел 10.3.

***Инициализируйте переменные, используемые в цикле, непосредственно перед циклом, а не в начале метода, содержащего цикл*** Следование этому совету снизит вероятность того, что при изменении цикла вы забудете изменить инициализацию используемых в нем переменных. Если же цикл будет вложен в новый цикл, переменные будут инициализироваться при каждой итерации нового цикла, а не только при первой.

**Перекрестная ссылка** Об этом стиле объявления и определения переменных см. подраздел «В идеальном случае сразу объявляйте и определяйте каждую переменную непосредственно перед первым обращением к ней» раздела 10.3.

***Не присваивайте переменной значение вплоть до его использования*** Вероятно, вы знаете, насколько трудно бывает найти строку, в которой переменной было присвоено ее значение. Чем больше вы сделаете для прояснения того, где переменная получает свое значение, тем лучше. Такие языки, как C++ и Java, позволяют инициализировать переменные следующим образом:

**Пример грамотного объявления и инициализации переменных (C++)**

```
int receiptIndex = 0;
float dailyReceipts = TodaysReceipts();
double totalReceipts = TotalReceipts( dailyReceipts );
```

**Перекрестная ссылка** О группировке связанных команд см. также раздел 14.2.

***Группируйте связанные команды*** В следующих фрагментах на примере метода, суммирующего дневную выручку, показано, как сгруппировать обращения к переменным, чтобы за ними было проще следить. В первом примере этот принцип нарушен:

**Пример запутанного использования двух наборов переменных (C++)**

```
void SummarizeData(...) {
   ...
```

Команды, в которых используются два набора переменных.

```
    GetOldData( oldData, &numOldData );
    GetNewData( newData, &numNewData );
    totalOldData = Sum( oldData, numOldData );
    totalNewData = Sum( newData, numNewData );
    PrintOldDataSummary( oldData, totalOldData, numOldData );
    PrintNewDataSummary( newData, totalNewData, numNewData );
    SaveOldDataSummary( totalOldData, numOldData );
    SaveNewDataSummary( totalNewData, numNewData );
    ...
}
```

Этот небольшой фрагмент заставляет следить сразу за шестью переменными: *oldData*, *newData*, *numOldData*, *numNewData*, *totalOldData* и *totalNewData*. В следующем примере благодаря разделению кода на два логических блока это число снижено до трех:

**Пример более понятного использования двух наборов переменных (C++)**

```
void SummarizeData( ... ) {
```

Команды, в которых используются «старые данные» (*oldData*).

```
    GetOldData( oldData, &numOldData );
    totalOldData = Sum( oldData, numOldData );
    PrintOldDataSummary( oldData, totalOldData, numOldData );
    SaveOldDataSummary( totalOldData, numOldData );
    ...
```

Команды, в которых используются «новые данные» (*newData*).

```
    GetNewData( newData, &numNewData );
    totalNewData = Sum( newData, numNewData );
    PrintNewDataSummary( newData, totalNewData, numNewData );
    SaveNewDataSummary( totalNewData, numNewData );
    ...
}
```

Каждый из двух блоков, полученных при разделении кода, короче, чем первоначальный блок, и содержит меньше переменных. Такой код легче понять, а если вам придется разбить его на отдельные методы, меньшие блоки, содержащие меньшее число переменных, позволят выполнять эту задачу эффективнее.

***Разбивайте группы связанных команд на отдельные методы*** При прочих равных условиях переменная из более короткого метода обычно характеризуется меньшим интервалом между обращениями и меньшим временем жизни, чем переменная из более крупного метода. Разбиение группы связанных команд на отдельные методы позволяет уменьшить область видимости, которую может иметь переменная.

***Начинайте с самой ограниченной области видимости и расширяйте ее только при необходимости*** Чтобы минимизировать область видимости переменной, постарайтесь сделать ее как можно более локальной. Область ви-

**Перекрестная ссылка** О глобальных переменных см. раздел 13.3.

димости гораздо сложнее сжать, чем расширить — иначе говоря, превратить глобальную переменную в переменную класса сложнее, чем наоборот. Защищенные данные-члены класса также сложнее превратить в закрытые, чем закрытые в защищенные. Так что, если сомневаетесь, выбирайте наименьшую возможную область видимости переменной: попытайтесь сделать переменную локальной для отдельного цикла, локальной для конкретного метода, затем — закрытой переменной класса, затем — защищенной, далее попробуйте включить ее в пакет (если ваш язык программирования поддерживает пакеты) и лишь в крайнем случае сделайте ее глобальной.

## Комментарии по поводу минимизации области видимости

Подход к минимизации области видимости переменных часто зависит от точки зрения на вопросы «удобства» и «интеллектуальной управляемости». Некоторые программисты делают многие переменные глобальными для того, чтобы облегчить доступ к ним и не беспокоиться о списках параметров и правилах области видимости. В их умах удобство доступа к глобальным переменным перевешивает связанную с этим опасность.

**Перекрестная ссылка** Идея минимизации области видимости связана с идеей сокрытия информации [см. подраздел «Скрывайте секреты (к вопросу о сокрытии информации)» раздела 5.3].

Другие предпочитают делать переменные как можно более локальными, потому что локальная область видимости способствует интеллектуальной управляемости. Чем больше информации вы скрыли, тем меньше вам нужно удерживать в уме в каждый конкретный момент времени и тем ниже вероятность того, что вы допустите ошибку, забыв одну из многих деталей, о которых нужно было помнить.

Разница между философией «удобства» и философией «интеллектуальной управляемости» сводится к различию между ориентацией на написание программы и ориентацией на ее чтение. Максимизация области видимости может облегчить написание программы, но программу, в которой каждый метод может вызвать любую переменную в любой момент времени, сложнее понять, чем код, основанный на грамотно организованных методах. Сделав данные глобальными, вы не сможете ограничиться пониманием работы одного метода: вы должны будете понимать работу всех других методов, которые вместе с ним используют те же глобальные данные. Подобные программы сложно читать, сложно отлаживать и сложно изменять.

**Перекрестная ссылка** О методах доступа см. подраздел «Использование методов доступа вместо глобальных данных» раздела 13.3.

Так что ограничивайте область видимости каждой переменной минимальным фрагментом кода. Можете ограничить ее одним циклом или одним методом — великолепно! Не получается — ограничьте область видимости методами одного класса. Если и это невозможно, создайте методы доступа, позволяющие использовать переменную совместно с другими классами. «Голые» глобальные данные требуются редко, если вообще такое бывает.

## 10.5. Персистентность

«Персистентность» — это еще одно слово, характеризующее длительность существования данных. Персистентность принимает несколько форм. Некоторые переменные «живут»:

- пока выполняется конкретный блок кода или метод: например, это переменные, объявленные внутри цикла *for* языка C++ или Java;
- столько, сколько вы им позволяете: в Java переменные, созданные при помощи оператора *new*, «живут» до сборки мусора; в C++ созданные аналогично переменные существуют, пока не будут уничтожены с помощью оператора *delete*;
- до завершения программы: этому описанию соответствуют глобальные переменные в большинстве языков, а также статические переменные в языках C++ и Java;
- всегда: такими переменными могут быть значения, которые вы храните в БД между запусками программы, — так, например, в случае интерактивной программы, позволяющей пользователям настраивать цвета экрана, вы могли бы хранить эти цвета в файле, считывая их при каждой загрузке программы.

Главная проблема персистентности возникает, когда вы предполагаете, что переменная существует дольше, чем есть на самом деле. Переменная похожа на пакет молока в холодильнике. Предполагается, что он может храниться неделю. Иногда он хранится месяц, и с молоком ничего не происходит, а иногда молоко скисает через пять дней. Переменная может быть столь же непредсказуема. Если вы попытаетесь использовать значение переменной после окончания нормальной длительности ее существования, получите ли вы ее прежнее значение? Иногда значение переменной «скисает», и вы получаете ошибку. В других случаях компилятор оставляет старое значение неизменным, позволяя вам воображать, что все нормально.

Проблем подобного рода можно избежать.

- Включайте в программу отладочный код или утверждения для проверки важных переменных на предмет допустимости их значений. Если значения недопустимы, отображайте предупреждение, рекомендующее приступить к поиску кода неверной инициализации.

**Перекрестная ссылка** Отладочный код легко включать в методы доступа (см. подраздел «Преимущества методов доступа» раздела 13.3).

- Завершив работу с переменными, присваивайте им «недопустимые значения». Скажем, после освобождения памяти при помощи оператора *delete* вы может установить указатель в *null*.
- Исходите из того, что данные не являются персистентными. Так, если при возврате из метода переменная имеет определенное значение, не предполагайте, что оно будет таким же при следующем вызове метода. Это правило неактуально, если вы используете специфические возможности языка, гарантирующие неизменность значения переменной, такие как ключевое слово *static* языков C++ и Java.
- Выработайте привычку объявлять и инициализировать все данные перед их использованием. Если видите обращение к данным, но не можете найти поблизости команду их инициализации, отнеситесь к этому с подозрением!

# 10.6. Время связывания

Одним из аспектов инициализации, серьезно влияющим на удобство сопровождения и изменения программы, является «время связывания» — момент, когда переменная и ее значение связываются вместе (Thimbleby, 1988). Связываются ли они при написании кода? При его компиляции? При загрузке? При выполнении программы? В другое время?

Иногда выгоднее использовать как можно более позднее время связывания. В целом чем позже вы выполняете связывание, тем более гибким будет ваш код. В следующем примере связывание выполняется максимально рано — при написании кода:

### Пример связывания во время написания кода (Java)

```
titleBar.color = 0xFF; // 0xFF – шестнадцатеричное значение синего цвета
```

Значение *0xFF* связывается с переменной *titleBar.color* во время написания кода, поскольку *0xFF* — литерал, жестко закодированный в программе. Как правило, это неудачное решение, потому что при изменении одного значения *0xFF* может утратиться его соответствие литералам *0xFF*, используемым в других фрагментах с той же целью.

В следующем примере связывание переменной выполняется чуть позднее, во время компиляции кода:

### Пример связывания во время компиляции (Java)

```
private static final int COLOR_BLUE = 0xFF;
private static final int TITLE_BAR_COLOR = COLOR_BLUE;
...
titleBar.color = TITLE_BAR_COLOR;
```

В данном случае *TITLE_BAR_COLOR* является именованной константой — выражением, вместо которого компилятор подставляет конкретное значение при компиляции. Этот подход почти всегда лучше, чем жесткое кодирование. Он облегчает чтение кода, потому что имя константы *TITLE_BAR_COLOR* лучше характеризует представляемое ей значение, чем *0xFF*. Он облегчает изменение цвета заголовка окна (title bar), так как изменение константы будет автоматически отражено во всех местах, где она используется. При этом он не приводит к снижению быстродействия программы в период выполнения.

Вот пример еще более позднего связывания — в период выполнения:

### Пример связывания в период выполнения (Java)

```
titleBar.color = ReadTitleBarColor();
```

*ReadTitleBarColor*() — это метод, который во время выполнения программы читает значение из реестра Microsoft Windows, файла свойств Java или подобного места.

Этот код более понятен и гибок, чем код с жестко закодированным значением. Чтобы изменить значение *titleBar.color*; вам не нужно изменять программу: достаточно просто изменить содержание файла, из которого метод *ReadTitleBarColor*()

читает значение цвета. Так часто делают при разработке интерактивных приложений, предоставляющих возможность настройки их параметров.

Кроме того, время связывания может определяться тем, когда вызывается метод *ReadTitleBarColor*(). Его можно вызывать при загрузке программы, при создании окна или при каждой перерисовке окна: каждый последующий вариант соответствует все более позднему времени связывания.

Итак, в нашем примере переменная может связываться со значением следующим образом (в других случаях детали могут быть несколько иными):

- при написании кода (с использованием магических чисел);
- при компиляции (с использованием именованной константы);
- при загрузке программы (путем чтения значения из внешнего источника, такого как реестр Windows или файл свойств Java);
- при создании объекта (например, путем чтения значения при каждом создании окна);
- по требованию (например, посредством чтения значения при каждой перерисовке окна).

В целом чем раньше время связывания, тем ниже гибкость и ниже сложность кода. Что до первых двух вариантов, то именованные константы по многим причинам предпочтительнее магических чисел, и вы можете получить гибкость, обеспечиваемую именованными константами, просто используя хорошие методики программирования. При дальнейшем повышении уровня гибкости повышается и сложность нужного для его поддержания кода, а вместе с ней и вероятность ошибок. Так как эффективность программирования зависит от минимизации сложности, опытный программист будет обеспечивать такой уровень гибкости, какой нужен для удовлетворения требований, но не более того.

## 10.7. Связь между типами данных и управляющими структурами

Между типами данных и управляющими структурами существуют четко определенные отношения, впервые описанные британским ученым Майклом Джексоном (Jackson, 1975). В этом разделе мы их вкратце обсудим.

Джексон проводит связи между тремя типами данных и соответствующими управляющими структурами.

***Последовательные данные соответствуют последовательности команд*** Последовательные данные (sequential data) — это набор блоков данных, используемых в определенном порядке (рис. 10-2). Если у вас есть пять команд подряд, обрабатывающих пять разных значений, они формируют последовательность команд. Если бы вам нужно было прочитать из файла последовательные данные (фамилию сотрудника, номер карточки социального обеспечения, адрес, телефон и возраст), вы включили бы в код последовательность команд.

**Перекрестная ссылка** О последовательном порядке выполнения команд см. главу 14.

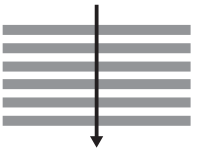


***Рис. 10-2.*** *Последовательными называются данные, обрабатываемые в определенном порядке*

**Перекрестная ссылка** Об условных операторах см. главу 15.

***Селективные данные соответствуют операторам* if *и* case** Вообще селективные данные (selective data) представляют собой набор, допускающий использование одного и только одного элемента данных в каждый конкретный момент времени (рис. 10-3). Соответствующими командами, выполняющими фактический выбор данных, являются операторы *if-then-else* или *case*. Так, программа расчета зарплаты должна была бы выполнять разные действия в зависимости от того, какой является оплата труда конкретного сотрудника: сдельной или повременной. Опять-таки шаблоны кода соответствуют шаблонам данных.

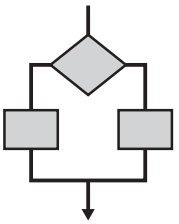


***Рис. 10-3.*** *Селективные данные допускают использование только одного из нескольких элементов*

**Перекрестная ссылка** О циклах см. главу 16.

***Итеративные данные соответствуют циклам*** Итеративные данные (iterative data) представляют собой данные одного типа, повторяющиеся более одного раза (рис. 10-4). Обычно они хранятся как элементы контейнера, записи файла или элементы массива. Скажем, вы могли бы хранить в файле список номеров карточек социального обеспечения, для чтения которого было бы разумно использовать соответствующий цикл.

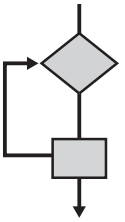


***Рис. 10-4.*** *Итеративные данные повторяются*

Реальные данные могут быть комбинацией последовательных, селективных и итеративных данных. Для описания сложных видов данных подойдет комбинация простых.

# 10.8. Единственность цели каждой переменной

Есть несколько тонких способов использования переменных более чем с одной целью, однако подобных тонкостей лучше избегать.

***Используйте каждую переменную только с одной целью*** Иногда есть соблазн вызвать одну переменную в двух разных местах для решения двух разных задач. Обычно в таких случаях переменной приходится присваивать неудачное имя, соответствующее одной из ее целей, или использовать для решения обеих задач «временную» переменную (как правило, с бесполезным именем *x* или *temp*). Следующий пример иллюстрирует использование временной переменной с двойной целью:

**Пример использования переменной с двойной целью — плохой подход (C++)**

```
// Вычисление корней квадратного уравнения.
// Предполагается, что дискриминант (b*b-4*a*c) неотрицателен.
temp = Sqrt( b*b - 4*a*c );
root[0] = ( -b + temp ) / ( 2 * a );
root[1] = ( -b - temp ) / ( 2 * a );
...

// корни меняются местами
temp = root[0];
root[0] = root[1];
root[1] = temp;
```

Вопрос: какие отношения связывают *temp* в первых строках кода и *temp* в последних? Ответ: никакие. Из-за использования в обоих случаях одной переменной создается впечатление, что две задачи связаны, хотя на самом деле это не так. Создание уникальных переменных для каждой цели делает код понятнее. Вот улучшенный вариант:

**Перекрестная ссылка** Параметры методов также должны иметь только одну цель. О параметрах методов см. раздел 7.5.

**Пример использования двух переменных для двух целей — хороший подход (C++)**

```
// Вычисление корней квадратного уравнения.
// Предполагается, что дискриминант (b*b-4*a*c) неотрицателен.
discriminant = Sqrt( b*b - 4*a*c );
root[0] = ( -b + discriminant ) / ( 2 * a );
root[1] = ( -b - discriminant ) / ( 2 * a );
...

// корни меняются местами
oldRoot = root[0];
root[0] = root[1];
root[1] = oldRoot;
```

***Избегайте переменных, имеющих скрытый смысл*** Другой способ использования переменной более чем с одной целью заключается в том, что разные значения переменной имеют разный смысл. Ниже я привел несколько примеров.

- Значение переменной *pageCount* представляет число отпечатанных страниц, однако, если оно равно -1, произошла ошибка.

- Если значение переменной *customerId* меньше 500 000, оно представляет номер заказчика, больше — вы вычитаете из него 500 000 для определения номера просроченного счета.
- Положительные значения переменной *bytesWritten* представляют число байт, записанных в выходной файл, а отрицательные — номер диска, используемого для вывода данных.

Избегайте подобных переменных, имеющих скрытый смысл. Формально это называется «гибридным сопряжением» (hybrid coupling) (Page-Jones, 1988). Переменная разрывается между двумя задачами, а это означает, что для решения одной из задач ее тип не подходит. В одном из наших примеров переменная *pageCount* в нормальной ситуации определяет число страниц — это целое число. Однако значение *-1* указывает на ошибку — целое число работает по совместительству булевой переменной!

Даже если применение переменных с двойной целью вам понятно, его не поймет кто-то другой. Дополнительная ясность, которой можно достигнуть благодаря использованию двух переменных для хранения двух видов данных, удивит вас. Никто не упрекнет вас в том, что вы впустую тратите ресурсы компьютера.

***Убеждайтесь в том, что используются все объявленные переменные*** Использование переменной с множественной целью имеет противоположность: переменную можно не использовать вообще. Кард, Черч и Агрести обнаружили, что наличие неиспользуемых переменных коррелировало с более высоким уровнем ошибок (Card, Church, and Agresti, 1986). Выработайте привычку проверять, что используются все объявленные переменные. Некоторые компиляторы и утилиты (такие как lint) предупреждают о наличии неиспользуемых переменных.

http://cc2e.com/1092

**Перекрестная ссылка** Контрольный список вопросов о специфических типах данных см. в главе 12, а контрольный список вопросов, касающихся именования переменных, — в главе 11.

### Контрольный список: общие вопросы использования данных

**Инициализация переменных**

- ❑ В каждом ли методе проверяется корректность входных параметров?
- ❑ Переменные объявляются около места их использования в первый раз?
- ❑ Инициализировали ли вы переменные при их объявлении, если такое возможно?
- ❑ Если переменные невозможно объявить и инициализировать одновременно, вы инициализировали их около места использования в первый раз?
- ❑ Правильно ли инициализируются счетчики и аккумуляторы? Выполняется ли их повторная инициализация, если она необходима?

- ❑ Осуществляется ли правильная повторная инициализация переменных в коде, который выполняется более одного раза?
- ❑ Код компилируется без предупреждений? (И задали ли вы самый строгий уровень диагностики?)
- ❑ Если язык поддерживает неявные объявления переменных, постарались ли вы предотвратить возможные проблемы?

**Другие общие вопросы использования данных**

- ❑ Все ли переменные имеют как можно меньшую область видимости?
- ❑ Являются ли обращения к переменным максимально сгруппированными как в плане интервала между обращениями, так и в плане общего времени жизни?
- ❑ Соответствуют ли управляющие структуры типам данных?
- ❑ Все ли объявленные переменные используются?
- ❑ Все ли переменные связываются в подходящее время, т. е. соблюдаете ли вы разумный баланс между гибкостью позднего связывания и соответствующей ему повышенной сложностью?
- ❑ Каждая ли переменная имеет одну и только одну цель?
- ❑ Не имеют ли какие-нибудь переменные скрытого смысла?

# Ключевые моменты

- Неграмотная инициализация данных часто приводит к ошибкам. Описанные в этой главе способы инициализации позволят избежать проблем, связанных с неожиданными первоначальными значениями переменных.
- Минимизируйте область видимости каждой переменной. Группируйте обращения к переменным. Старайтесь делать переменные локальными для методов или классов. Избегайте глобальных данных.
- Располагайте команды, использующие одни и те же переменные, как можно ближе друг к другу.
- Раннее связывание ограничивает гибкость, но минимизирует сложность программы. Позднее связывание повышает гибкость, но за это приходится расплачиваться повышенной сложностью.
- Используйте каждую переменную исключительно с одной целью.

ГЛАВА 11

# Сила имен переменных

http://cc2e.com/1184

**Содержание**



**Связанные темы**

- Имена методов: раздел 7.3
- Имена классов: раздел 6.2
- Общие принципы использования переменных: глава 10
- Размещение объявлений данных: одноименный подраздел раздела 31.5
- Документирование переменных: подраздел «Комментирование объявлений данных» раздела 32.5

Несмотря на всю важность выбора удачных имен для эффективного программирования, я не знаю ни одной книги, в которой эта тема обсуждается хотя бы с минимально приемлемым уровнем детальности. Многие авторы посвящают пару абзацев выбору аббревиатур, приводят несколько банальных примеров и ожидают, что вы сами о себе позаботитесь. Я рискую быть обвиненным в противоположном: вы получите настолько подробные сведения об именовании переменных, что никогда не сможете использовать их в полном объеме!

Советы, рассматриваемые в этой главе, касаются преимущественно именования переменных: объектов и элементарных типов данных. Однако их следует учитывать и при именовании классов, пакетов, файлов и других сущностей из мира программирования. Об именовании методов см. также раздел 7.3.

# 11.1. Общие принципы выбора имен переменных

Имя переменной нельзя выбирать, как кличку собаке: опираясь на его вычурность или звучание. В отличие от собаки и ее клички, которые являются разными сущностями, переменная и ее имя формируют по идее одну сущность. Поэтому и адекватность переменной во многом определяется ее именем. Выбирайте имена переменных со всей тщательностью.

В следующем примере переменные названы плохо:

**Пример неудачного именования переменных (Java)**

```
x = x - xx;
xxx = fido + SalesTax( fido );
x = x + LateFee( x1, x ) + xxx;
x = x + Interest( x1, x );
```

Что происходит в этом фрагменте кода? Что означают имена *x1*, *xx* и *xxx*? А *fido*? Допустим, кто-то сказал вам, что этот код подсчитывает общую сумму предъявляемого клиенту счета, опираясь на его долг и стоимость новых покупок. Какую переменную вы использовали бы для распечатки общей стоимости только новых покупок?

Взглянув на исправленный вариант того же кода, ответить на этот вопрос куда проще:

**Пример удачного именования переменных (Java)**

```
balance = balance - lastPayment;
monthlyTotal = newPurchases + SalesTax( newPurchases );
balance = balance + LateFee( customerID, balance ) + monthlyTotal;
balance = balance + Interest( customerID, balance );
```

Из сравнения этих фрагментов можно сделать вывод, что хорошее имя переменной адекватно ее характеризует, легко читается и хорошо запоминается. Чтобы облегчить себе достижение этих целей, соблюдайте несколько общих правил.

## Самый важный принцип именования переменных

Важнейший принцип именования переменных состоит в том, что имя должно полно и точно описывать сущность, представляемую переменной. Один эффективный способ выбора хорошего имени предполагает формулирование сути переменной в словах. Оптимальным именем переменной часто оказывается само это высказывание. Благодаря отсутствию загадочных сокращений оно удобочитаемо; к тому же оно однозначно. Так как оно является полным описанием сущности, его нельзя спутать с чем-либо другим. Наконец, такое имя легко запомнить, потому что оно похоже на исходную концепцию.

Переменную, представляющую число членов олимпийской команды США, можно было бы назвать *numberOfPeopleOnTheUsOlympicTeam*. Переменную, представ-

ляющую число мест на стадионе, — *numberOfSeatsInTheStadium*. Для хранения максимального числа очков, набранных спортсменами какой-то страны в современной Олимпиаде, можно было бы создать переменную *maximumNumberOfPointsInModernOlympics*. Переменную, определяющую текущую процентную ставку, лучше было бы назвать *rate* или *interestRate*, а не *r* или *x*. Думаю, идею вы поняли.

Обратите внимание на две характеристики этих имен. Во-первых, их легко расшифровать. Фактически их не нужно расшифровывать вообще: их можно просто прочитать. Ну, а во-вторых, некоторые имена велики — слишком велики, чтобы быть практичными. Длину имен переменных мы рассмотрим ниже.

Несколько примеров удачных и неудачных имен переменных я привел в табл. 11-1.

**Табл. 11-1. Примеры удачных и неудачных имен переменных**

| Суть переменной | Удачные имена, адекватное описание | Неудачные имена, неадекватное описание |
|---|---|---|
| Сумма, на которую на данный момент выписаны чеки | *runningTotal*, *checkTotal* | *written*, *ct*, *checks*, *CHKTTL*, *x*, *x1*, *x2* |
| Скорость поезда | *velocity*, *trainVelocity*, *velocityInMph* | *velt*, *v*, *tv*, *x*, *x1*, *x2*, *train* |
| Текущая дата | *currentDate*, *todaysDate* | *cd*, *current*, *c*, *x*, *x1*, *x2*, *date* |
| Число строк на странице | *linesPerPage* | *lpp*, *lines*, *l*, *x*, *x1*, *x2* |

Имена *currentDate* и *todaysDate* — хорошие имена, потому что полно и точно описывают идею «текущей даты». Фактически они составлены из слов с очевидным значением. Программисты иногда упускают из виду обычные слова, которые порой приводят к самому простому решению. Имена *cd* и *c* неудачны потому, что слишком коротки и «неописательны». Имя *current* тоже неудачно: оно не говорит, что именно является «текущим». Имя *date* кажется хорошим, но в итоге оно оказывается плохим, потому что мы имеем в виду не любую дату, а текущую; само по себе имя *date* об этом не говорит. Имена *x*, *x1* и *x2* заведомо неудачны: *x* традиционно представляет неизвестное количество чего-либо, и, если вы не хотите, чтобы ваши переменные были неизвестными величинами, подумайте о выборе других имен.

Имена должны быть максимально конкретны. Имена *x*, *temp*, *i* и другие, достаточно общие для того, чтобы их можно было использовать более чем с одной целью, не так информативны, как могли бы быть, и обычно являются плохими.

## Ориентация на проблему

Хорошее мнемоническое имя чаще всего описывает проблему, а не ее решение. Хорошее имя в большей степени выражает *что*, а не *как*. Если же имя описывает некоторый аспект вычислений, а не проблемы, имеет место обратное. Предпочитайте таким именам переменных имена, характеризующие саму проблему.

Запись данных о сотруднике можно было бы назвать *inputRec* или *employeeData*. Имя *inputRec* — компьютерный термин, выражающий идеи ввода данных и записи. Имя *employeeData* относится к проблемной области, а не к миру компьютеров. В случае битового поля, определяющего статус принтера, имя *bitFlag* более

компьютеризировано, чем *printerReady*, а в случае приложения бухгалтерского учета *calcVal* более компьютеризировано, чем *sum*.

## Оптимальная длина имени переменной

Оптимальная длина имени, наверное, лежит где-то между длинами имен *x* и *maximumNumberOfPointsInModernOlympics*. Слишком короткие страдают от недостатка смысла. Проблема с именами вроде *x1* и *x2* в том, что, даже узнав, что такое *x*, вы ничего не сможете сказать об отношении между *x1* и *x2*. Слишком длинные имена надоедает печатать, к тому же они могут сделать неясной визуальную структуру программы.

Горла, Бенандер и Бенандер обнаружили, что отладка программы требовала меньше всего усилий, если имена переменных состояли в среднем из 10–16 символов (Gorla, Benander, and Benander, 1990). Отладка программ с именами, состоящими в среднем из 8–20 символов, была почти столь же легкой. Это не значит, что следует присваивать всем переменным имена из 9–15 или 10–16 символов, — это значит, что, увидев в своем коде много более коротких имен, вы должны проверить их ясность.

Вопрос адекватности длины имен переменных поясняет табл. 11-2.

**Табл. 11-2. Слишком длинные, слишком короткие и оптимальные имена переменных**

| | |
|---|---|
| Слишком длинные имена: | *numberOfPeopleOnTheUsOlympicTeam* |
| | *numberOfSeatsInTheStadium* |
| | *maximumNumberOfPointsInModernOlympics* |
| Слишком короткие имена: | *n*, *np*, *ntm* |
| | *n, ns, nsisd* |
| | *m, mp, max, points* |
| То, что надо: | *numTeamMembers*, *teamMemberCount* |
| | *numSeatsInStadium, seatCount* |
| | *teamPointsMax, pointsRecord* |

## Имена переменных и область видимости

Всегда ли короткие имена переменных неудачны? Нет, не всегда. Если вы присваиваете переменной короткое имя, такое как *i*, сама длина имени говорит о том, что переменная является второстепенной и имеет ограниченную область действия.

**Перекрестная ссылка** Об области видимости см. раздел 10.4.

Программист, читающий код, сможет догадаться, что использование такой переменной ограничивается несколькими строками кода. Присваивая переменной имя *i*, вы говорите: «Эта переменная — самый обычный счетчик цикла/индекс массива, не играющий никакой роли вне этих нескольких строк».

У. Дж. Хансен (W. J. Hansen) обнаружил, что более длинные имена лучше присваивать редко используемым или глобальным переменным, а более короткие — локальным переменным или переменным, вызываемым в циклах (Shneiderman, 1980). Однако с короткими именами связано много проблем, и некоторые осмот-

рительные программисты, придерживающиеся политики защитного программирования, вообще избегают их.

***Дополняйте имена, относящиеся к глобальному пространству имен, спецификаторами*** Если у вас есть переменные, относящиеся к глобальному пространству имен (именованные константы, имена классов и т. д.), подумайте, принять ли конвенцию, разделяющую глобальное пространство имен на части и предотвращающую конфликты имен. В C++ и C# для разделения глобального пространства имен можно применить ключевое слово *namespace*:

**Пример разделения глобального пространства имен с помощью ключевого слова *namespace* (C++)**

```
namespace UserInterfaceSubsystem {
   ...
   // объявления переменных
   ...
}

namespace DatabaseSubsystem {
   ...
   // объявления переменных
   ...
}
```

Если класс *Employee* объявлен в обоих пространствах имен, вы можете указать нужное пространство имен, написав *UserInterfaceSubsystem::Employee* или *DatabaseSubsystem::Employee*. В Java с той же целью можно использовать пакеты.

Программируя на языке, не поддерживающем пространства имен или пакеты, вы все же можете принять конвенции именования для разделения глобального пространства имен. Скажем, вы можете дополнить глобальные классы префиксами, определяющими подсистему. Класс *Employee* из подсистемы пользовательского интерфейса можно было бы назвать *uiEmployee*, а тот же класс из подсистемы доступа к БД — *dbEmployee*. Это позволило бы свести к минимуму риск конфликтов в глобальном пространстве имен.

## Спецификаторы вычисляемых значений

Многие программы включают переменные, содержащие вычисляемые значения: суммы, средние величины, максимумы и т. д. Дополняя такое имя спецификатором вроде *Total*, *Sum*, *Average*, *Max*, *Min*, *Record*, *String* или *Pointer*, укажите его в конце имени.

У такого подхода несколько достоинств. Во-первых, при этом самая значимая часть имени переменной, определяющая наибольшую часть его смысла, располагается в самом начале имени, из-за чего становится более заметной и читается первой. Во-вторых, приняв эту конвенцию, вы предотвратите путаницу, возможную при наличии в одной программе имен *totalRevenue* и *revenueTotal*. Эти имена семантически эквивалентны, и конвенция не позволила бы использовать их как разные. В-третьих, набор имен вроде *revenueTotal*, *expenseTotal*, *revenueAverage* и *expenseAverage* обладает приятной глазу симметрией, тогда как набор имен *totalRevenue*,

*expenseTotal*, *revenueAverage* и *averageExpense* упорядоченным не кажется. Наконец, согласованность имен облегчает чтение и сопровождение программы.

Исключение из этого правила — позиция спецификатора *Num*. При расположении в начале имени спецификатор *Num* обозначает общее число: например, *numCustomers* — это общее число заказчиков. Если же он указан в конце имени, то определяет индекс: так, *customerNum* — это номер текущего заказчика. Другим признаком данного различия является буква *s* в конце имени *numCustomers*. Однако даже в этом случае спецификатор *Num* очень часто приводит к замешательству, поэтому лучше всего полностью исключить проблему, применив *Count* или *Total* для обозначения общего числа заказчиков и *Index* для ссылки на конкретного заказчика. Таким образом, переменные, определяющие общее число заказчиков и номер конкретного заказчика, получили бы имена *customerCount* и *customerIndex* соответственно.

### Антонимы, часто встречающиеся в именах переменных

Используйте антонимы последовательно. Это делает код более согласованным и облегчает его чтение. Пары вроде *begin/end* понять и запомнить легко. Пары, не соответствующие общепринятым антонимам, запоминаются сложнее и вызывают замешательство. В число антонимов, часто используемых в именах переменных, входят:

**Перекрестная ссылка** Аналогичный список антонимов, используемых в именах методов, см. в подразделе «Дисциплинированно используйте антонимы» раздела 7.3.

- begin/end;
- first/last;
- locked/unlocked;
- min/max;
- next/previous;
- old/new;
- opened/closed;
- visible/invisible;
- source/target;
- source/destination;
- up/down.

## 11.2. Именование конкретных типов данных

При именовании конкретных типов данных следует руководствоваться не только общими, но и специфическими соображениями. Ниже описаны принципы именования индексов циклов, переменных статуса, временных переменных, булевых переменных, перечислений и именованных констант.

### Именование индексов циклов

Принципы именования индексов циклов возникли потому, что циклы относятся к самым популярным конструкциям. Как правило, в качестве индексов циклов используют переменные *i*, *j* и *k*:

**Перекрестная ссылка** О циклах см. главу 16.

#### Пример простого имени индекса цикла (Java)

```
for ( i = firstItem; i < lastItem; i++ ) {
   data[ i ] = 0;
}
```

Если же переменную предполагается использовать вне цикла, ей следует присвоить более выразительное имя. Например, переменную, хранящую число записей, прочитанных из файла, можно было бы назвать *recordCount*:

#### Пример удачного описательного имени индекса цикла (Java)

```
recordCount = 0;
while ( moreScores() ) {
   score[ recordCount ] = GetNextScore();
   recordCount++;
}

// строки, в которых используется переменная recordCount
...
```

Если цикл длиннее нескольких строк, смысл переменной *i* легко забыть, поэтому в подобной ситуации лучше присвоить индексу цикла более выразительное имя. Так как код очень часто приходится изменять, модернизировать и копировать в другие программы, многие опытные программисты вообще не используют имена вроде *i*.

Одна из частых причин увеличения циклов — их вложение в другие циклы. Если у вас есть несколько вложенных циклов, присвойте индексам более длинные имена, чтобы сделать код более понятным:

#### Пример удачного именования индексов вложенных циклов (Java)

```
for ( teamIndex = 0; teamIndex < teamCount; teamIndex++ ) {
   for ( eventIndex = 0; eventIndex < eventCount[ teamIndex ]; eventIndex++ ) {
      score[ teamIndex ][ eventIndex ] = 0;
   }
}
```

Тщательный выбор имен индексов циклов позволяет избежать путаницы индексов — использования *i* вместо *j* и наоборот. Кроме того, это облегчает понимание операций над массивами: команда *score[ teamIndex ][ eventIndex ]* более информативна, чем *score[ i ][ j ]*.

Не присваивайте имена *i*, *j* и *k* ничему, кроме индексов простых циклов: нарушение этой традиции только запутает других программистов. Чтобы избежать подобных проблем, просто подумайте о более описательных именах, чем *i*, *j* и *k*.

## Именование переменных статуса

Переменные статуса характеризуют состояние программы. Ниже рассмотрен один принцип их именования.

***Старайтесь не присваивать переменной статуса имя* flag** Наоборот, рассматривайте флаги как переменные статуса. Имя флага не должно включать фрагмент *flag*, потому что он ничего не говорит о сути флага. Ради ясности флагам следует присваивать выразительные значения, которые лучше сравнивать со значениями перечислений, именованных констант или глобальных переменных, выступающих в роли именованных констант. Вот примеры неудачных имен флагов:

**Примеры загадочных флагов (C++)**

```
if ( flag ) ...
if ( statusFlag & 0x0F ) ...
if ( printFlag == 16 ) ...
if ( computeFlag == 0 ) ...

flag = 0x1;
statusFlag = 0x80;
printFlag = 16;
computeFlag = 0;
```

Команды вида *statusFlag = 0x80* будет совершенно непонятны, пока вы не поясните в коде или в документации, что такое *statusFlag* и *0x80*. Вот более понятный эквивалентный фрагмент:

**Примеры более грамотного использования переменных статуса (C++)**

```
if ( dataReady ) ...
if ( characterType & PRINTABLE_CHAR ) ...
if ( reportType == ReportType_Annual ) ...
if ( recalcNeeded == True ) ...

dataReady = true;
characterType = CONTROL_CHARACTER;
reportType = ReportType_Annual;
recalcNeeded = false;
```

Очевидно, что команда *characterType = CONTROL_CHARACTER* выразительнее, чем *statusFlag = 0x80*. Аналогично условие *if ( reportType == ReportType_Annual )* понятнее, чем *if ( printFlag == 16 )*. Второй фрагмент показывает, что данный подход применим к перечислениям и предопределенным именованным константам. Вот как с помощью именованных констант и перечислений можно было бы задать используемые в нашем примере значения:

**Объявление переменных статуса (C++)**

```
// возможные значения переменной CharacterType
const int LETTER = 0x01;
const int DIGIT = 0x02;
const int PUNCTUATION = 0x04;
const int LINE_DRAW = 0x08;
const int PRINTABLE_CHAR = ( LETTER | DIGIT | PUNCTUATION | LINE_DRAW );
```

```
const int CONTROL_CHARACTER = 0x80;

// возможные значения переменной ReportType
enum ReportType {
    ReportType_Daily,
    ReportType_Monthly,
    ReportType_Quarterly,
    ReportType_Annual,
    ReportType_All
};
```

Если вам трудно понять какой-то фрагмент кода, подумайте о переименовании переменных. В отличие от детективных романов код программ не должен содержать загадок. Его нужно просто читать.

## Именование временных переменных

Временные переменные служат для хранения промежуточных результатов вычислений и служебных значений программы. Обычно им присваивают имена *temp*, *x* или какие-нибудь другие столь же неопределенные и неописательные имена. В целом использование временных переменных говорит о том, что программист еще не полностью понял проблему. Кроме того, с переменными, официально получившими «временный» статус, программисты обычно обращаются небрежнее, чем с другими переменными, что повышает вероятность ошибок.

***Относитесь к «временным» переменным с подозрением*** Часто значение нужно на некоторое время сохранить. Однако в том или ином смысле временными являются почти все переменные. Называя переменную временной, подумайте, до конца ли вы понимаете ее реальную роль. Рассмотрим пример:

#### Пример неинформативного имени «временной» переменной (C++)

```
// Вычисление корней квадратного уравнения.
// Предполагается, что дискриминант (b^2-4*a*c) неотрицателен.
temp = sqrt( b^2 - 4*a*c );
root[0] = ( -b + temp ) / ( 2 * a );
root[1] = ( -b - temp ) / ( 2 * a );
```

Значение выражения *sqrt( b^2 - 4 * a * c )* вполне разумно сохранить в переменной, особенно если учесть, что оно используется позднее. Но имя *temp* ничего не говорит о роли переменной. Лучше поступить так:

#### Пример замены «временной» переменной на реальную переменную (C++)

```
// Вычисление корней квадратного уравнения.
// Предполагается, что дискриминант (b^2-4*a*c) неотрицателен.
discriminant = sqrt( b^2 - 4*a*c );
root[0] = ( -b + discriminant ) / ( 2 * a );
root[1] = ( -b - discriminant ) / ( 2 * a );
```

По сути это тот же код, только в нем использована переменная с точным описательным именем.

## Именование булевых переменных

Ниже я привел ряд рекомендаций по именованию булевых переменных.

***Помните типичные имена булевых переменных*** Вот некоторые наиболее полезные имена булевых переменных.

- ***done*** Используйте переменную *done* как признак завершения цикла или другой операции. Присвойте ей *false* до выполнения действия и установите ее в *true* после его завершения.
- ***error*** Используйте переменную *error* как признак ошибки. Присвойте ей значение *false*, если все в порядке, и *true* в противном случае.
- ***found*** Используйте переменную *found* для определения того, обнаружено ли некоторое значение. Установите ее в *false*, если значение не обнаружено, и в *true*, как только значение найдено. Используйте переменную *found* при поиске значения в массиве, идентификатора сотрудника в файле, определенного чека в списке чеков и т. д.
- ***success*** или ***ok*** Используйте переменную *success* или *ok* как признак успешного завершения операции. Присвойте ей *false*, если операция завершилась неудачей, и *true*, если операция выполнена успешно. Если можете, замените имя *success* на более определенное, ясно определяющее смысл «успеха». Если под «успехом» понимается завершение обработки данных, можете назвать переменную *processingComplete*. Если «успех» подразумевает обнаружение конкретного значения, можете использовать переменную *found*.

***Присваивайте булевым переменным имена, подразумевающие значение* true *или* false** Имена вроде *done* и *success* — хорошие имена булевых переменных, потому что они предполагают использование только значений *true* или *false*: что-то может быть или выполнено, или не выполнено, операция может завершиться или успехом, или неудачей. С другой стороны, имена вроде *status* и *sourceFile* не годятся, так как при этом значения *true* или *false* не имеют ясного смысла. Какой вывод можно сделать, если переменной *status* задано *true*? Означает ли это, что что-то имеет статус? Все имеет статус. Означает ли это, что что-то имеет статус «все в порядке»? Означает ли значение *false*, что никакое действие не было выполнено неверно? Если переменная имеет имя *status*, ничего определенного на сей счет сказать нельзя.

Поэтому имя *status* лучше заменить на имя вроде *error* или *statusOK*, а имя *sourceFile* — на *sourceFileAvailable*, *sourceFileFound* или подобное имя, соответствующее сути переменной.

Некоторые программисты любят дополнять имена булевых переменных префиксом *is*. В результате имя переменной превращается в вопрос: *isdone*? *isError*? *isFound*? *isProcessingComplete*? Ответ на этот вопрос сразу становится и значением переменной. Достоинство этого подхода в том, что он исключает использование неопределенных имен: вопрос *isStatus*? не имеет никакого смысла. Однако в то же время он затрудняет чтение логических выражений: например, условие *if* ( *isFound* ) менее понятно, чем *if* ( *found* ).

***Используйте утвердительные имена булевых переменных*** Имена, основанные на отрицании (такие как *notFound*, *notdone* и *notSuccessful*), при выполнении над переменной операции отрицания становятся куда менее понятны, например:

```
if not notFound
```

Подобные имена следует заменить на *found*, *done* и *processingComplete*, выполняя отрицание переменных в случае надобности. Так что для проверки нужного значения вы использовали бы выражение *found*, а не *not notFound*.

## Именование перечислений

**Перекрестная ссылка** О перечислениях см. раздел 12.6.

Принадлежность переменных к тому или иному перечислению можно пояснить, дополнив их имена префиксами, такими как *Color_*, *Planet_* или *Month_*:

**Пример дополнения элементов перечислений префиксами (Visual Basic)**

```
Public Enum Color
   Color_Red
   Color_Green
   Color_Blue
End Enum

Public Enum Planet
   Planet_Earth
   Planet_Mars
   Planet_Venus
End Enum

Public Enum Month
   Month_January
   Month_February
   ...
   Month_December
End Enum
```

Кроме того, сами перечисления (*Color*, *Planet* или *Month*) можно идентифицировать разными способами: например, используя в их именах только заглавные буквы или дополняя их имена префиксами (*e_Color*, *e_Planet* или *e_Month*). Кое-кто мог бы сказать, что перечисление по сути является типом, определяемым пользователем, поэтому имена перечислений надо форматировать так же, как имена классов и других пользовательских типов. С другой стороны, члены перечислений являются константами, поэтому имена перечислений следует форматировать как имена констант. В этой книге я придерживаюсь конвенции, предусматривающей применение в именах перечислений букв обоих регистров.

В некоторых языках перечисления рассматриваются скорее как классы, а именам членов перечисления всегда предшествует имя самого перечисления, например, *Color.Color_Red* или *Planet.Planet_Earth*. Если вы используете подобный язык, повторять префикс не имеет смысла, так что вы можете считать префиксом само имя перечисления и сократить имена до *Color.Red* и *Planet.Earth*.

### Именование констант

Имя константы должно характеризовать абстрактную сущность, представляемую константой, а не конкретное значение. Имя *FIVE* — плохое имя константы (независимо от того, имеет ли она значение *5.0*). *CYCLES_NEEDED* — хорошее имя. *CYCLES_NEEDED* может иметь значение *5.0*, *6.0* и любое другое. Выражение *FIVE = 6.0* было бы странным. Аналогично *BAKERS_DOZEN* — плохое имя константы, а *DONUTS_MAX* — вполне подходящее.

**Перекрестная ссылка** Об именованных константах см. раздел 12.7.

## 11.3. Сила конвенций именования

Программисты предпочитают порой не использовать стандарты и конвенции по вполне разумной причине. Некоторые стандарты и конвенции, слишком жесткие и неэффективные, подавляют творчество и снижают качество программы. Это печально, так как эффективные стандарты — один из мощнейших инструментов. В этом разделе мы обсудим, почему, когда и как создавать собственные стандарты именования переменных.

### Зачем нужны конвенции?

Конвенции обеспечивают несколько преимуществ.

- Они позволяют больше принимать как данное. Приняв одно общее решение вместо нескольких более узких, вы сможете сосредоточиться на более важных аспектах кода.
- Они помогают использовать знания, полученные при работе над предыдущими проектами. Сходство имен облегчает понимание незнакомых переменных.
- Они ускоряют изучение кода нового проекта. Вместо изучения особенностей кода Аниты, Джулии и Кристин вы сможете работать с более согласованным кодом.
- Они подавляют «размножение» имен. Не применяя конвенцию именования, вы легко можете присвоить одной сущности два разных имени. Скажем, вы можете назвать общее число баллов и *pointTotal*, и *totalPoints*. Возможно, при написании программы вам все будет понятно, но другой программист, который попытается понять такой код позднее, может столкнуться с серьезными проблемами.
- Они компенсируют слабости языка. Вы можете использовать конвенции для имитации именованных констант и перечислений. Конвенции позволяют провести грань между локальными данными, данными класса и глобальными данными, а также охарактеризовать типы, не поддерживаемые компилятором.
- Они подчеркивают отношения между связанными элементами. Если вы используете данные объектов, компилятор заботится об этом автоматически. Если язык не поддерживает объекты, вы можете свести этот недостаток к минимуму при помощи конвенции именования. Так, имена *address*, *phone* и *name* не говорят о том, что эти переменные связаны между собой. Но если вы решите дополнять все переменные, хранящие данные о сотрудниках, префиксом *Employee*, эта связь будет ясно выражена в итоговых именах *employeeAddress*, *employeePhone* и *employeeName*. Конвенции программирования могут устранить недостатки используемого вами языка.

Суть сказанного в том, что наличие хоть какой-то конвенции обычно предпочтительнее, чем ее отсутствие. Конвенция может быть произвольной. Сила конвенций именования объясняется не конкретными аспектами, а самим фактом их использования, обеспечивающим структурирование кода и уменьшающим количество поводов для беспокойства.

### Когда следует использовать конвенцию именования?

Непреложных правил на этот счет нет, однако некоторые рекомендации дать можно. Итак, используйте конвенцию именования, если:

- над проектом работают несколько программистов;
- программу будут изменять и сопровождать другие программисты (что имеет место почти всегда);
- обзор программы выполняют другие программисты из вашей компании;
- программа так велика, что вы не можете полностью охватить ее умом, а вынуждены рассматривать по частям;
- программа будет использоваться длительное время, из-за чего вам, возможно, придется вернуться к ней через несколько недель или месяцев;
- прикладная область имеет необычную терминологию и вы хотите стандартизовать применение терминов или аббревиатур в коде.

Использовать конвенции именования всегда выгодно, а мои советы помогут вам определить оптимальную детальность конвенции для конкретного проекта.

### Степень формальности конвенций

**Перекрестная ссылка** О различиях формальности при работе над небольшими и крупными проектами см. главу 27.

Конвенциям именования могут соответствовать разные степени формальности. Неформальная конвенция может быть совсем простой: «Используйте выразительные имена». Другие неформальные конвенции рассматриваются в следующем разделе. В общем, оптимальная степень формальности конвенции определяется числом людей, работающих над программой, размером программы и ожидаемым временем ее использования. При работе над крошечными проектами, подлежащими выбросу, следование строгой конвенции, наверное, окажется пустой тратой сил. В случае более крупных проектов, реализуемых с участием нескольких программистов, формальная конвенция — важнейшее средство улучшения удобочитаемости программы.

## 11.4. Неформальные конвенции именования

В большинстве проектов используются относительно неформальные конвенции именования, подобные тем, что описываются в этом разделе.

### Конвенция, не зависящая от языка

Ниже приведены некоторые советы по созданию конвенции, не зависящей от языка.

***Проведите различие между именами переменных и именами методов*** Конвенция, используемая в этой книге, подразумевает, что имена переменных и

объектов начинаются со строчной буквы, а имена методов — с прописной: *variableName*, но *RoutineName*().

***Проведите различие между классами и объектами*** Соответствие между именами классов и объектов (или между именами типов и переменных этих типов) может быть довольно тонким. Некоторые стандартные способы проведения различия между ними иллюстрирует следующий фрагмент:

```
Вариант 1: имена типов отличаются от имен переменных
регистром первой буквы

Widget widget;
LongerWidget longerWidget;

Вариант 2: имена типов отличаются от имен переменных регистром всех букв

WIDGET widget;
LONGERWIDGET longerWidget

Вариант 3: имена типов дополняются префиксом «t_»

t_Widget Widget;
t_LongerWidget LongerWidget;

Вариант 4: имена переменных дополняются префиксом «a»

Widget aWidget;
LongerWidget aLongerWidget;

Вариант 5: имена переменных более конкретны, чем имена типов

Widget employeeWidget;
LongerWidget fullEmployeeWidget;
```

Каждый из этих вариантов имеет свои плюсы и минусы. Вариант 1 часто используется при программировании на C++, Java и других языках, чувствительных к регистру букв, но некоторые программисты считают, что различать имена только по регистру первой буквы неудобно. Действительно, имена, отличающиеся только регистром первой буквы, имеют слишком малое психологическое и визуальное различие.

Вариант 1 не удастся согласованно использовать при программировании на нескольких языках, если хотя бы в одном из них регистр букв не имеет значения. Так, при компиляции команды *Dim widget as Widget* компилятор Microsoft Visual Basic сообщит о синтаксической ошибке, потому что *widget* и *Widget* покажутся ему одним и тем же элементом.

Вариант 2 проводит более очевидное различие между именами типов и переменных. Однако по историческим причинам в C++ и Java верхний регистр служит для определения констант, к тому же при разработке программы с использованием нескольких языков этот подход приводит к тем же проблемам, что и вариант 1.

Вариант 3 поддерживается всеми языками, но некоторым программистам префиксы не нравятся по эстетическим причинам.

Вариант 4 иногда используют как альтернативу варианту 3, но вместо изменения имени одного класса он требует модификации имени каждого экземпляра класса.

Вариант 5 заставляет тщательно обдумывать имя каждой переменной. Обычно результатом этого является более понятный код. Но иногда *widget* (приспособление) на самом деле — всего лишь общее «приспособление», и в этих случаях вы должны будете придумывать менее ясные имена вроде *genericWidget*, которые, несомненно, читаются хуже.

Короче, каждый из вариантов связан с компромиссами. В этой книге я использую вариант 5, потому что он наиболее понятен, если человеку, читающему код, неизвестны конвенции именования.

***Идентифицируйте глобальные переменные*** Одной из частых проблем программирования является неверное использование глобальных переменных. Если вы присвоите всем глобальным переменным имена, начинающиеся, скажем, с префикса *g_*, программист, увидевший переменную *g_RunningTotal*, сразу поймет, что это глобальная переменная, и будет обращаться с ней должным образом.

***Идентифицируйте переменные-члены*** Идентифицируйте данные-члены класса. Ясно покажите, что переменная-член не является ни локальной, ни глобальной переменной. Идентифицировать переменные-члены класса можно, например, при помощи префикса *m_*.

***Идентифицируйте определения типов*** Конвенции именования типов играют две роли: они явно показывают, что имя является именем типа, и предотвращают конфликты имен типов и переменных. Для этого вполне годится префикс (суффикс). В C++ для именования типов обычно используют только заглавные буквы: например, *COLOR* и *MENU*. (Это справедливо для имен типов, определяемых с помощью директив *typedef*, и имен структур, но не классов.) Однако при этом можно спутать типы с именованными константами препроцессора. Для предотвращения путаницы можно дополнять имена типов префиксом *t_*, что дает нам такие имена, как *t_Color* и *t_Menu*.

***Идентифицируйте именованные константы*** Именованные константы нужно идентифицировать, чтобы вы могли определить, присваиваете ли вы переменной значение другой переменной (которое может изменяться) или именованной константы. В случае Visual Basic эти два варианта можно также спутать с присваиванием переменной значения, возвращаемого функцией. Visual Basic не требует применения скобок при вызове функции, не принимающей параметров, тогда как в C++ скобки нужно указывать при вызове любой функции.

Одним из подходов к именованию констант является применение префикса, например *c_*. Это дает нам такие имена, как *c_RecsMax* или *c_LinesPerPageMax*. В случае C++ и Java конвенция подразумевает использование только заглавных букв без разделения слов или с разделением слов символами подчеркивания: *RECSMAX* или *RECS_ MAX* и *LINESPERPAGEMAX* или *LINES_PER_PAGE_ MAX*.

***Идентифицируйте элементы перечислений*** Элементы перечислений следует идентифицировать по той же причине, что и именованные константы: чтобы элемент перечисления можно было легко отличить от переменной, именованной константы или вызова функции. Стандартный подход предполагает применение в имени перечисления только заглавных букв или дополнение имени префиксом *e_* или *E_*; что касается имен элементов, то они дополняются префиксом, основанным на имени конкретного перечисления, скажем, *Color_* или *Planet_*.

***Идентифицируйте неизменяемые параметры, если язык не требует их явного определения*** Иногда программисты случайно изменяют входные параметры. C++, Visual Basic и некоторые другие языки заставляют явно указывать, хотите ли вы, чтобы изменения параметров внутри метода были доступны в остальном коде. Для этого служат спецификаторы *, & и *const* в C++ и *ByRef/ByVal* в Visual Basic.

В случае других языков изменение входной переменной в методе отражается в остальном коде, хотите вы того или нет. Это особенно верно при передаче объектов. Например, в Java все объекты передаются в методы «значением», поэтому, передавая объект в метод, будьте готовы к тому, что состояние объекта может измениться[1] (Arnold, Gosling, Holmes, 2000).

Если, программируя на таком языке, вы следуете конвенции именования, согласно которой исключительно входные (неизменяемые) параметры нужно дополнять префиксом *const* (или *final*, или *nonmodifiable*, или каким-то аналогичным), то, увидев что-то с префиксом *const* слева от знака равенства, вы будете знать, что произошла ошибка. Если вы увидите вызов *constMax.SetNewMax( … )*, вы также по префиксу *const* поймете, что это ошибка.

**Перекрестная ссылка** Дополнение языка конвенцией именования, компенсирующей ограничения самого языка, является примером программирования с использованием языка вместо простого программирования на языке (см. раздел 34.4).

***Форматируйте имена так, чтобы их было легко читать*** Для повышения удобочитаемости кода слова в именах переменных часто разделяют заглавными буквами или символами-разделителями. Например, имя *GYMNASTICSPOINTTOTAL* читается хуже, чем *gymnasticsPointTotal* или *gymnastics_point_total*. C++, Java, Visual Basic и другие языки позволяют использовать оба этих подхода.

Старайтесь не смешивать эти способы, так как это осложняет чтение кода. Если же вы будете согласованно использовать один из подходов, код станет более понятным. Программисты уже давно спорят по поводу того, делать ли заглавной первую букву имени (*TotalPoints* или *totalPoints*), но если все участвующие в проекте программисты будут поступать согласованно, подобные мелочи не будут играть особой роли. В данной книге имена переменных начинаются с буквы нижнего регистра по той причине, что этот подход принят в языке Java, а также для поддержания сходства стилей между разными языками.

## Конвенции, специфические для конкретных языков

Соблюдайте конвенции именования, принятые в используемом вами языке. Книги по стилю программирования можно найти почти для любого языка. Советы, относящиеся к языкам C, C++, Java и Visual Basic, даны в следующих подразделах.

### Конвенции C

Конвенции именования, используемые при программировании на C, предполагают, что:

- имена символьных переменных дополняются префиксом *c* или *ch*;
- целочисленным индексам присваиваются имена *i* и *j*;

[1] Значением передается ссылка на объект, который и может быть изменен. — *Прим. перев.*

**Дополнительные сведения** Классической книгой о стиле программирования на C является «C Programming Guidelines» (Plum, 1984).

- имена переменных, хранящих количество чего-либо, дополняются префиксом *n*;
- имена указателей дополняются префиксом *p*;
- имена строк начинаются с префикса *s*;
- имена макросов препроцессора включают *ТОЛЬКО_ЗАГЛАВНЫЕ_БУКВЫ*; обычно это правило распространяется и на имена типов, определяемых при помощи директивы *typedef*;
- имена переменных и методов включают *только_строчные_буквы*;
- для разделения слов служит символ подчеркивания (_): *имена_такого_вида* читаются легче, чем *именатакоговида*.

Эти правила справедливы для программирования на C в общем, а также для сред UNIX и Linux, однако в разных средах конвенции имеют свои особенности. Программисты на C, разрабатывающие программы для Microsoft Windows, предпочитают применять для именования переменных ту или иную форму венгерской нотации и буквы верхнего и нижнего регистров. Программисты, разрабатывающие ПО для платформы Macintosh, обычно используют для именования методов смешанный регистр, потому что инструментарий Macintosh и методы ОС были изначально разработаны в соответствии с интерфейсом Pascal.

### Конвенции C++

**Дополнительные сведения** О стиле программирования на C++ см. книгу «The Elements of C++ Style» (Misfeldt, Bumgardner, and Gray, 2004).

С программированием на C++ связаны такие конвенции:

- целочисленным индексам присваиваются имена *i* и *j*;
- имена указателей дополняются префиксом *p*;
- имена констант, типов, определяемых с помощью директивы *typedef*, и макросов препроцессора включают *ТОЛЬКО_ЗАГЛАВНЫЕ_БУКВЫ*;
- имена классов и других типов содержат *БуквыОбоихРегистров*;
- первое слово в именах переменных и методов начинается со строчной буквы, а все последующие слова — с заглавной: *имяПеременнойИлиМетода*;
- символ подчеркивания используется только в именах, состоящих полностью из заглавных букв, и после некоторых префиксов (например, после префикса, служащего для идентификации глобальных переменных).

Как и в случае языка C, некоторые аспекты этой конвенции могут зависеть от конкретной среды.

### Конвенции Java

**Дополнительные сведения** О стиле программирования на Java см. книгу «The Elements of Java Style, 2d ed.» (Vermeulen et al., 2000).

В отличие от C и C++ конвенции стиля программирования на Java были сформулированы уже на ранних этапах развития этого языка:

- *i* и *j* — имена целочисленных индексов;
- имена констант включают *ТОЛЬКО_ЗАГЛАВНЫЕ_БУКВЫ*, а слова разделяются символами подчеркивания;

- все слова в именах классов и интерфейсов начинаются с заглавной буквы: *ИмяКлассаИлиИнтерфейса*;
- в именах переменных и методов с заглавной буквы начинаются все слова, кроме первого: *имяПеременойИлиМетода*;
- символ подчеркивания служит разделителем только в именах, полностью состоящих из заглавных букв;
- имена методов доступа начинаются с префикса *get* или *set*.

#### Конвенции Visual Basic

Устойчивых конвенций стиля программирования на Visual Basic не существует. Чуть ниже я приведу один из возможных вариантов.

## Программирование с использованием нескольких языков

Если вы программируете, используя несколько языков, сформулируйте конвенцию именования (а также форматирования, документирования и т. д.) так, чтобы она способствовала общей согласованности и удобочитаемости кода, даже если для этого придется отступить от конвенции, принятой в одном из языков.

Например, в этой книге все имена переменных начинаются со строчной буквы, что соответствует конвенции Java и некоторым, но не всем конвенциям C++. Все имена методов начинаются с заглавной буквы, что согласуется с конвенцией C++. Согласно конвенции Java имена методов должны были бы начинаться с буквы нижнего регистра, но ради общей удобочитаемости я решил независимо от языка начинать их с заглавной буквы.

## Примеры конвенций именования

В стандартных конвенциях, описанных выше, не отражены некоторые важные аспекты, в том числе область видимости переменных (закрытая, класс или глобальная), различия имен классов и объектов, методов и переменных и т. д.

Советы по именованию могут показаться сложными, если они сконцентрированы на нескольких страницах. Однако на самом деле они могут быть вполне простыми, и вы можете адаптировать их к своим потребностям. Имена переменных должны включать информацию трех видов:

- суть переменной (то, что переменная представляет);
- тип данных (именованная константа; элементарная переменная; тип, определенный пользователем, или класс);
- область видимости переменной (закрытая, класс, пакет или глобальная область видимости).

В табл. 11-3, 11-4 и 11-5 описаны конвенции именования для языков C, C++, Java и Visual Basic, основанные на уже известных вам принципах. Использовать именно эти конвенции не обязательно, однако они помогут вам понять, что может включать неформальная конвенция именования.

**Табл. 11-3. Пример конвенции именования для языков C++ и Java**

| Сущность | Описание |
|---|---|
| *ClassName* | Имена классов начинаются с заглавной буквы и включают буквы обоих регистров. |
| *TypeName* | Имена типов, в том числе перечислений и типов, определяемых при помощи директив *typedef*, начинаются с заглавной буквы и включают буквы обоих регистров. |
| *EnumeratedTypes* | Кроме предыдущего правила, имена перечислений всегда имеют форму множественного числа. |
| *localVariable* | Имена локальных переменных начинаются со строчной буквы и включают буквы обоих регистров. Имя должно характеризовать сущность, представляемую переменной, и не должно зависеть от фактического типа переменной. |
| *routineParameter* | Имена параметров методов форматируются так же, как имена локальных переменных. |
| *RoutineName()* | Имена методов включают буквы обоих регистров (об удачных именах методов см. раздел 7.3). |
| *m_ClassVariable* | Имена переменных-членов, доступных только методам класса, дополняются префиксом *m_*. |
| *g_GlobalVariable* | Имена глобальных переменных дополняются префиксом g_. |
| *CONSTANT* | Имена именованных констант включают *ТОЛЬКО_ЗАГЛАВНЫЕ_БУКВЫ*. |
| *MACRO* | Имена макросов включают *ТОЛЬКО_ЗАГЛАВНЫЕ_БУКВЫ*. |
| *Base_EnumeratedType* | Имена элементов перечислений дополняются именем самого перечисления в единственном числе: *Color_Red, Color_Blue*. |

**Табл. 11-4. Пример конвенции именования для языка C**

| Сущность | Описание |
|---|---|
| *TypeName* | Имена типов начинаются с заглавной буквы и включают буквы обоих регистров. |
| *GlobalRoutineName()* | Имена открытых методов включают буквы обоих регистров. |
| *f_FileRoutineName()* | Имена методов, видимых в одном модуле (файле), дополняются префиксом *f_*. |
| *LocalVariable* | Имена локальных переменных включают буквы обоих регистров. Имя должно характеризовать сущность, представляемую переменной, и не должно зависеть от фактического типа переменной. |
| *RoutineParameter* | Имена параметров методов форматируются так же, как имена локальных переменных. |
| *f_FileStaticVariable* | Имена переменных, видимых в одном модуле (файле), дополняются префиксом *f_*. |
| *G_GLOBAL_GlobalVariable* | Имена глобальных переменных дополняются префиксом *G_* и обозначением модуля (файла), в котором определена переменная; обозначение модуля (файла) включает только заглавные буквы: *SCREEN_Dimensions*. |

**Табл. 11-4.** (*окончание*)

| Сущность | Описание |
|---|---|
| *LOCAL_CONSTANT* | Имена именованных констант, видимых в одном методе или модуле (файле), включают только заглавные буквы: *ROWS_MAX*. |
| *G_GLOBALCONSTANT* | Имена глобальных именованных констант включают только заглавные буквы и дополняются префиксом *G_* и обозначением модуля (файла), в котором определена именованная константа; обозначение модуля (файла) включает только заглавные буквы: *G_SCREEN_ROWS_MAX*. |
| *LOCALMACRO()* | Имена макросов, видимых в одном методе или модуле (файле), включают только заглавные буквы. |
| *G_GLOBAL_MACRO()* | Имена глобальных макросов включают только заглавные буквы и дополняются префиксом *G_* и обозначением модуля (файла), в котором определен макрос; обозначение модуля (файла) включает только заглавные буквы: *G_SCREEN_LOCATION()*. |

Так как Visual Basic безразличен к регистру букв, для различения имен типов и переменных приходится применять специфические правила (табл. 11-5).

**Табл. 11-5. Пример конвенции именования для языка Visual Basic**

| Сущность | Описание |
|---|---|
| *C_ClassName* | Имена классов дополняются префиксом *C_*, начинаются с заглавной буквы и включают буквы обоих регистров. |
| *T_TypeName* | Имена типов дополняются префиксом T_, начинаются с заглавной буквы и включают буквы обоих регистров. |
| *T_EnumeratedTypes* | Кроме предыдущего правила, имена перечислений всегда имеют форму множественного числа. |
| *localVariable* | Имена локальных переменных начинаются со строчной буквы и включают буквы обоих регистров. Имя должно характеризовать сущность, представляемую переменной, и не должно зависеть от фактического типа переменной. |
| *routineParameter* | Имена параметров методов форматируются так же, как имена локальных переменных. |
| *RoutineName()* | Имена методов включают буквы обоих регистров (об удачных именах методов см. раздел 7.3). |
| *m_ClassVariable* | Имена переменных-членов, доступных только методам класса, дополняются префиксом *m_*. |
| *g_GlobalVariable* | Имена глобальных переменных дополняются префиксом *g_*. |
| *CONSTANT* | Имена именованных констант включают *ТОЛЬКО_ЗАГЛАВНЫЕ_БУКВЫ*. |
| *Base_EnumeratedType* | Имена элементов перечислений дополняются именем самого перечисления в единственном числе: *Color_Red, Color_Blue*. |

# 11.5. Стандартизованные префиксы

**Дополнительные сведения** О венгерской нотации см. статью «The Hungarian Revolution» (Simonyi and Heller, 1991).

Стандартизация префиксов обеспечивает лаконичный, но в то же время согласованный и понятный способ именования данных. Самая известная схема стандартизации префиксов — венгерская нотация — представляет собой набор детальных принципов именования переменных и методов (а не жителей Венгрии!) , который когда-то широко применялся при программировании для ОС Microsoft Windows. Сейчас венгерскую нотацию используют редко, но ее суть — создание стандартизованного набора лаконичных точных аббревиатур — от этого не становится менее полезной.

Стандартизованный префикс состоит из двух частей: аббревиатуры типа, определенного пользователем (user-defined type, UDT), и семантического префикса.

## Аббревиатура типа, определенного пользователем

Аббревиатура UDT обозначает тип объекта или переменной. Как правило, аббревиатуры UDT служат для описания таких сущностей, как окна, области экрана и шрифты, но не предопределенных типов данных конкретного языка программирования.

Типы UDT описываются краткими кодами, которые вы создаете и стандартизируете для конкретной программы. Коды — это мнемонические обозначения, такие как *wn* в случае окна и *scr* в случае области экрана. В табл. 11-6 приведены примеры UDT, которые можно было бы задействовать в текстовом редакторе.

**Табл. 11-6. Примеры UDT текстового редактора**

| Аббревиатура UDT | Значение |
|---|---|
| *ch* | Символ (тип данных, используемый для представления символа документа, а не символ C++) |
| *doc* | Документ |
| *pa* | Абзац (paragraph) |
| *scr* | Область экрана |
| *sel* | Выбранный текст |
| *wn* | Окно |

При работе с UDT следует также определить типы данных с именами, соответствующими аббревиатурам UDT. Таким образом, при использовании UDT из табл. 11-6 у вас получились бы подобные объявления данных:

```
CH    chCursorPosition;
SCR   scrUserWorkspace;
DOC   docActive
PA    firstPaActiveDocument;
PA    lastPaActiveDocument;
WN    wnMain;
```

Разумеется, аббревиатуры следует создавать для тех UDT, которые чаще всего встречаются в конкретной среде.

## Семантические префиксы

Семантические префиксы дополняют аббревиатуры UDT, характеризуя использование переменной или объекта. В отличие от аббревиатур UDT, зависимых от конкретного проекта, семантические префиксы являются в некотором смысле стандартными (табл. 11-7).

**Табл. 11-7. Семантические префиксы**

| Семантический префикс | Значение |
|---|---|
| *c* | Количество (записей, символов и т. д.). |
| *first* | Элемент массива, обрабатываемый первым. Префикс *first* аналогичен префиксу *min*, но связан с текущей операцией, а не с самим массивом. |
| *g* | Глобальная переменная. |
| *i* | Индекс массива. |
| *last* | Элемент массива, обрабатываемый последним. Префикс *last* дополняет префикс *first*. |
| *lim* | Верхняя граница обрабатываемого массива. Значение с префиксом *lim* уже не является допустимым индексом. Как и *last*, префикс *lim* дополняет префикс *first*. В отличие от префикса *last*, используемого для представления последнего допустимого элемента, значение с *lim* выходит за пределы массива. В общем, *lim* равняется *last* + 1. |
| *m* | Переменная уровня класса. |
| *max* | Индекс последнего элемента массива или другого списка. Префикс *max* связан с самим массивом, а не с выполняемыми над массивом операциями. |
| *min* | Индекс первого элемента массива или другого списка. |
| *p* | Указатель. |

Семантические префиксы включают строчные буквы или буквы обоих регистров и по мере необходимости объединяются с аббревиатурами UDT и другими семантическими префиксами. Например, имя переменной, определяющей первый абзац документа, включило бы аббревиатуру *pa*, говорящую о том, что это абзац, и префикс *first*, показывающий, что это первый абзац. В итоге мы получили бы имя *firstPa*. Индекс набора абзацев был бы назван *iPa*; счетчик или число абзацев — *cPa*, а первый и последний абзацы текущего активного документа — *firstPaActiveDocument* и *lastPaActiveDocument* соответственно.

## Достоинства стандартизованных префиксов

Стандартизованные префиксы обеспечивают все общие преимущества конвенций именования, а также некоторые дополнительные. Стандартизация имен снижает число имен элементов программы или класса, которые нужно помнить.

Стандартизованные префиксы позволяют уточнить имена, которые без этого часто оказываются неточными. Особенно полезны точные различия между префиксами *min*, *first*, *last* и *max*.

Стандартизованные префиксы делают имена более компактными. Так, переменной, определяющей число абзацев, можно присвоить имя *cpa*, а не *totalParagraphs*. Индекс массива абзацев можно назвать *ipa*, а не *indexParagraphs* или *paragraphsIndex*.

Наконец, стандартизованные префиксы облегчают проверку правильности использования абстрактных типов данных, когда компилятор оказывается беспомощным. Так, выражение *paReformat = docReformat* скорее всего ошибочно, потому что аббревиатуры *pa* и *doc* соответствуют разным UDT.

Главная ловушка при использовании стандартизованных префиксов — отказ от дополнения префикса выразительным именем переменной. Так, если имя *ipa* однозначно определяет индекс массива абзацев, есть соблазн не присваивать переменной более выразительное имя, такое как *ipaActiveDocument*. Помните про удобочитаемость кода и присваивайте переменным описательные имена.

# 11.6. Грамотное сокращение имен переменных

Стремление к сокращению имен переменных в некотором смысле стало пережитком. В более старых языках, таких как ассемблер, обычный Basic и Fortran, имена переменных были ограничены 2–8 символами. Кроме того, раньше программирование было более тесно связано с математикой, что побуждало использовать в уравнениях и других выражениях переменные с «математическими» именами *i*, *j*, *k* и т. п. C++, Java, Visual Basic и другие современные языки позволяют создавать имена почти любой длины, поэтому сокращение выразительных имен уже не имеет под собой практически никаких оснований.

Если обстоятельства требуют создания коротких имен, помните, что некоторые способы сокращения имен лучше других. Удачные короткие имена можно создать, устранив ненужные слова, выбрав более короткие синонимы и использовав какую-нибудь из нескольких стратегий сокращения. Целесообразно знать несколько методик сокращения имен, потому что ни одна из них не является одинаково эффективной во всех случаях.

## Общие советы по сокращению имен

Ниже я привел несколько рекомендаций по сокращению имен. Некоторые из них противоречат другим, так что не пытайтесь использовать все советы сразу. Итак:

- используйте стандартные аббревиатуры (общепринятые, которые можно найти в словаре);
- удаляйте все гласные, не являющиеся первыми буквами имен (*computer — cmptr*, *screen — scrn*, *apple — appl*, *integer — intgr*);
- удаляйте артикли и союзы, такие как *and*, *or*, *the* и т. д.;
- сохраняйте одну или несколько первых букв каждого слова;
- «обрезайте» слова согласованно: после первой, второй или третьей буквы (выбирайте вариант, уместный в конкретном случае);
- сохраняйте первую и последнюю буквы каждого слова;
- сохраняйте до трех выразительных слов;
- удаляйте бесполезные суффиксы: *ing*, *ed* и т. д.

- сохраняйте наиболее выразительный звук каждого слога;
- проверяйте, чтобы смысл имени переменной в ходе сокращения не искажался;
- используйте эти способы, пока не сократите имя каждой переменной до 8–20 символов или до верхнего предела, ограничивающего длину имен в конкретном языке.

## Фонетические аббревиатуры

Некоторые люди сокращают слова, опираясь на их звучание, а не написание. В результате *skating* превращается в *sk8ing*, *highlight* — в *hilite*, *before* — в *b4*, *execute* — в *xqt* и т. д. Этот способ не отличается понятностью, поэтому я рекомендую забыть про него. Попробуйте, например, догадаться, что означают имена:

*ILV2SK8* *XMEQWK* *S2DTM8O* *NXTC* *TRMN8R*

## Комментарии по поводу сокращения имен

Сокращая имена, вы можете попасть в одну из нескольких ловушек. Ниже описаны некоторые правила, позволяющие их избежать.

***Не сокращайте слова только на один символ*** Напечатать лишний символ не так уж трудно, и экономия одного символа едва ли может оправдать ухудшение удобочитаемости кода. При этом имена становятся похожими на названия месяцев в календаре. Нужно очень уж сильно торопиться, чтобы написать «Июн» вместо «Июнь». Обычно после сокращения слов на один символ потом трудно вспомнить, действительно ли вы удалили этот символ. Или удаляйте более одного символа, или пишите все слово.

***Сокращайте имена согласованно*** Всегда используйте один и тот же вариант сокращения — например, только *Num* или только *No*, но не оба варианта. Аналогично не сокращайте слово только в некоторых именах. Если в одних именах вы использовали слово *Number*, не сокращайте его в других именах до *Num* и наоборот.

***Сокращайте имена так, чтобы их можно было произнести*** Используйте имена *xPos* и *needsComp*, а не *xPstn* и *ndsCmptg*. Возьмите на заметку телефонный тест: если вы не можете прочитать код другому человеку по телефону, присвойте переменным более членораздельные имена (Kernighan and Plauger, 1978).

***Избегайте комбинаций, допускающих неверное прочтение или произношение имени*** Если вам нужно как-то обозначить конец *B*, назовите переменную *ENDB*, а не *BEND*. Если вы грамотно разделяете части имен, этот совет вам не понадобится, так как сочетания *B-END*, *BEnd* или *b_end* нельзя произнести неправильно.

***Обращайтесь к словарю для разрешения конфликтов имен*** При сокращении некоторых имен в итоге получается одна и та же аббревиатура. Так, если длина имени ограничена тремя символами и вам нужно использовать в одной части программы элементы *fired* и *full revenue disbursal*, вы можете по неосторожности сократить оба варианта до *frd*.

Предотвратить конфликт имен позволяют синонимы, и тут на помощь приходит словарь. В нашем примере *fired* можно было бы заменить на синоним *dismissed*, а *full revenue disbursal* — на *complete revenue disbursal*. В итоге получаются аббревиатуры *dsm* и *crd*, что устраняет конфликт имен.

***Документируйте очень короткие имена прямо в коде при помощи таблиц*** Если язык позволяет использовать только очень короткие имена, включайте в код таблицу, характеризующую суть переменных. Включайте ее как комментарий перед соответствующим блоком кода, например:

**Пример хорошей таблицы преобразования (Fortran)**

```
C   ****************************************************************
C     Translation Table
C
C     Variable    Meaning
C     ----        ----
C     XPOS        x-Coordinate Position (in meters)
C     YPOS        Y-Coordinate Position (in meters)
C     NDSCMP      Needs Computing (=0 if no computation is needed;
C                                  =1 if computation is needed)
C     PTGTTL      Point Grand Total
C     PTVLMX      Point Value Maximum
C     PSCRMX      Possible Score Maximum
C   ****************************************************************
```

Может показаться, что этот способ устарел, но не далее чем в середине 2003 г. я работал с клиентом, который использовал программу на языке RPG, включавшую несколько сотен тысяч строк кода. Длина имен переменных в ней была ограничена 6 символами. Подобные проблемы все еще всплывают время от времени.

***Указывайте все сокращения в проектном документе «Стандартные аббревиатуры»*** Применение аббревиатур сопряжено с двумя распространенными факторами риска:

- аббревиатура может оказаться непонятной программисту, читающему код;
- программисты могут сократить одно и то же имя по-разному, что вызывает ненужное замешательство.

Для предотвращения этих потенциальных проблем вы можете создать документ «Стандартные аббревиатуры», описывающий все аббревиатуры конкретного проекта. Сам документ может быть файлом текстового редактора или электронной таблицей. При работе над очень крупным проектом им может быть база данных. Этот документ следует зарегистрировать в системе управления версиями и изменять его каждый раз, когда кто-то создает новую аббревиатуру. Элементы документа должны быть сортированы по полным словам, а не по аббревиатурам.

Может показаться, что эта методика связана с большим объемом дополнительной работы, но на самом деле она требует небольших усилий на начальном этапе, предоставляя взамен механизм, помогающий эффективно использовать аббревиатуры. Она устраняет первый из двух факторов риска, заставляя документировать все используемые аббревиатуры. Если программист не может создать новую аббревиатуру, не проверив документ «Стандартные аббревиатуры», не изменив его и снова не зарегистрировав в системе управления версиями, — *это хорошо*. Это значит, что программисты будут создавать аббревиатуры, только когда будут чувствовать, что выгода от их создания стоит того, чтобы преодолеть все барьеры, связанные с их документированием.

Вероятность создания избыточных аббревиатур также снижается. Программист, желающий сократить какое-то слово, должен будет проверить документ и ввести новую аббревиатуру. Если это слово уже было сокращено, программист заметит это и будет использовать существующую аббревиатуру.

Этот совет иллюстрирует различие между удобством написания кода и удобством его чтения. Очевидно, что рассмотренный подход делает написание кода *менее удобным*, однако на протяжении срока службы системы программисты проводят гораздо больше времени именно за чтением кода, которое благодаря этому подходу облегчается. Когда вся пыль осядет, вполне может оказаться, что этот подход облегчил и написание кода.

***Помните, что имена создаются в первую очередь для программистов, читающих код*** Прочитайте собственный код, который вы не видели минимум шесть месяцев, и обратите внимание на те имена, суть которых вы не можете понять с первого взгляда. Измените подходы, подталкивающие к выбору таких имен.

## 11.7. Имена, которых следует избегать

Ниже я описал некоторые типы имен, которых следует избегать.

***Избегайте обманчивых имен или аббревиатур*** Убедитесь в том, что имя не является двусмысленным. Скажем, *FALSE* обычно является противоположностью *TRUE*, и использовать такое имя как сокращение фразы «Fig and Almond Season» было бы глупо.

***Избегайте имен, имеющих похожие значения*** Если есть вероятность, что вы можете спутать имена двух переменных и это не приведет к ошибке компиляции, переименуйте обе переменных. Например, пары имен *input* и *inputValue*, *recordNum* и *numRecords* или *fileNumber* и *fileIndex* так похожи с семантической точки зрения, что если вы будете использовать их в одном фрагменте кода, то сможете легко их спутать, внеся в код неуловимые ошибки.

***Избегайте переменных, имеющих разную суть, но похожие имена*** Если у вас есть две таких переменных, попытайтесь переименовать одну из них или изменить аббревиатуры. Избегайте имен вроде *clientRecs* и *clientReps*. Они различаются только одной буквой, и это трудно заметить. Выбирайте имена, различающиеся хотя бы двумя буквами или первой/последней буквой. Имена *clientRecords* и *clientReports* лучше, чем первоначальные имена.

**Перекрестная ссылка** Технически подобное различие между сходными именами переменных называется «психологической дистанцией» (см. подраздел «Психологическая дистанция» раздела 23.4).

***Избегайте имен, имеющих похожее звучание, таких как wrap и rap*** Когда вы пытаетесь обсудить код с другими людьми, в разговор иногда вмешиваются омонимы. Так, одним из самых забавных аспектов экстремального программирования (Beck, 2000) является слишком хитрое использование терминов «Goal Donor» и «Gold Owner»[1], которые звучат практически одинаково. В итоге разговор может принять подобный оборот:

[1] Что буквально переводится как «донор цели» и «владелец золота». В экстремальном программировании так называют роли людей, соответственно ставящих перед разработчиками задачу и финансирующих проект. — *Прим. перев.*

— Я только что разговаривал с Goal Donor.
— Что ты сказал? «Gold Owner» или «Goal Donor»?
— Я сказал «Goal Donor».
— Что?
— GOAL - - - DONOR!
— Ясно, Goal Donor. Не нужно кричать, черт возьми (Goll' Darn it).
— Какое еще «золотое кольцо» (Gold Donut)?

Как и в случае непроизносимых аббревиатур, используйте для исключения подобных ситуаций телефонный тест.

***Избегайте имен, включающих цифры*** Если наличие цифр в именах действительно имеет смысл, используйте вместо отдельных переменных массив. Если этот вариант неуместен, цифры в именах еще более неуместны. Например, избегайте имен *file1* и *file2* или *total1* и *total2*. Почти всегда можно найти более разумный способ проведения различия между двумя переменными, чем дополнение их имен цифрами. В то же время я не могу сказать, что цифры *нельзя* использовать вообще. Некоторые сущности реального мира (такие как шоссе 66) изначально включают цифры. И все же перед созданием подобного имени подумайте, есть ли лучшие варианты.

***Избегайте орфографических ошибок*** Вспомнить правильные имена переменных и так довольно трудно. Требовать запоминания «правильных» орфографических ошибок — это уж слишком. Например, если вы решите сэкономить три буквы и замените слово *highlight* на *hilite*, программисту, читающему код, будет неописуемо трудно вспомнить, на что же вы его заменили. На *higblite*? *hilite*? *hiligbt*? *hilit*? *jai-a-lai-t*? Кто его знает.

***Избегайте слов, при написании которых люди часто допускают ошибки*** *Absense*, *acummulate*, *acsend*, *calender*, *concieve*, *defferred*, *definate*, *independance*, *occassionally*, *prefered*, *reciept*, *superseed* и многие другие орфографические ошибки весьма распространены в англоязычном мире. Список подобных слов можно найти в большинстве справочников по английскому языку. Не используйте такие слова в именах переменных.

***Проводите различие между именами не только по регистру букв*** Если вы программируете на C++ или другом языке, в котором регистр букв играет роль, вы можете поддастся соблазну сократить понятия *fired*, *final review duty* и *full revenue disbursal* соответственно до *frd*, *FRD* и *Frd*. Избегайте этого подхода. Хотя эти имена уникальны, их связь с конкретными значениями произвольна и непонятна. Имя *Frd* может с тем же успехом обозначать *final review duty*, а *FRD — full revenue disbursal*, и никакое логическое правило не поможет вам или кому-то другому запомнить, что есть что.

***Избегайте смешения естественных языков*** Если в проекте участвуют программисты разных национальностей, обяжите их именовать все элементы программы, используя один естественный язык. Понять код другого программиста непросто; а код, написанный на юго-восточном диалекте марсианского языка, — невозможно.

Еще более тонкая проблема связана с наличием разных диалектов английского языка. Если в проекте участвуют представители разных англоязычных стран, сде-

лайте стандартом тот или иной вариант английского языка, чтобы вам не пришлось вспоминать, как называется конкретная переменная: «color» или «colour», «check» или «cheque» и т. д.

***Избегайте имен стандартных типов, переменных и методов*** Все языки программирования имеют зарезервированные и предопределенные имена. Просматривайте время от времени списки таких имен, чтобы не вторгаться во владения используемого языка. Так, следующий фрагмент вполне допустим при программировании на PL/I, но написать ТАКОЕ может только идиот со справкой:

```
if if = then then
   then = else;
else else = if;
```

***Не используйте имена, которые совершенно не связаны с тем, что представляют переменные*** Использование имен вроде *margaret* и *pookie* практически гарантирует, что никто другой их не поймет. Не называйте переменные в честь девушки, жены, любимого сорта пива и т. д., если только девушка, жена или сорт пива не являются представляемыми в программе «сущностями». Но даже тогда вы должны понимать, что все в мире изменяется, поэтому имена *девушка*, *жена* и *любимыйСортПива* гораздо лучше!

***Избегайте имен, содержащих символы, которые можно спутать с другими символами*** Помните, что некоторые символы выглядят очень похоже. Если два имени различаются только одним таким символом, вы можете столкнуться с проблемами. Попробуйте, например, определить, какое из имен является лишним в каждой тройке:

```
eyeChart1       eyeChartI       eyeChart1
TTLCONFUSION    TTLCONFUSION    TTLCONFUSION
hard2Read       hardZRead       hard2Read
GRANDTOTAL      GRANDTOTAL      6RANDTOTAL
ttl5            ttlS            ttlS
```

В число пар символов, которые трудно различить, входят пары (1 и l), (1 и I), (. и ,), (0 и O), (2 и Z), (; и :), (S и 5) и (G и 6).

Действительно ли важны подобные детали? Да! Джеральд Вайнберг пишет, что в 1970-х из-за того, что в команде *FORMAT* языка Fortran вместо точки была использована запятая, ученые неверно рассчитали траекторию космического корабля и потеряли космический зонд стоимостью 1,6 млрд долларов (Weinberg, 1983).

**Перекрестная ссылка** Вопросы, касающиеся использования данных, приведены в контрольном списке в главе 10.

**Контрольный список: именование переменных**

http://cc2e.com/1191

**Общие принципы именования переменных**

- ❑ Описывает ли имя представляемую переменной сущность полно и точно?
- ❑ Характеризует ли имя проблему реального мира, а не ее решение на языке программирования?

- ❑ Имеет ли имя длину, достаточную для того, чтобы над ним не нужно было ломать голову?
- ❑ Спецификаторы вычисляемых значений находятся в конце имен?
- ❑ Используются ли в именах спецификаторы *Count* или *Index* вместо *Num*?

**Именование конкретных видов данных**

- ❑ Выразительные ли имена присвоены индексам циклов (более ясные, чем *i*, *j* и *k*, если цикл содержит более одной-двух строк или является вложенным)?
- ❑ Всем ли «временным» переменным присвоены выразительные имена?
- ❑ Можно ли по именам булевых переменных понять, какой смысл имеют значения «истина» и «ложь»?
- ❑ Включают ли имена элементов перечислений префикс или суффикс, определяющий принадлежность элемента к перечислению — например, префикс *Color_* в случае элементов *Color_Red*, *Color_Green*, *Color_Blue* и т. д.?
- ❑ Именованные константы названы в соответствии с представляемыми абстрактными сущностями, а не конкретными числами?

**Конвенции именования**

- ❑ Проводит ли конвенция различие между локальными данными, данными класса и глобальными данными?
- ❑ Проводит ли конвенция различие между именами типов, именованных констант, перечислений и переменных?
- ❑ Идентифицировали ли вы исключительно входные параметры методов, если язык не навязывает их идентификацию?
- ❑ Постарались ли вы сделать конвенцию как можно более совместимой со стандартными конвенциями конкретного языка?
- ❑ Способствует ли форматирование имен удобству их чтения?

**Короткие имена**

- ❑ Стараетесь ли вы не сокращать имена без необходимости?
- ❑ Избегаете ли вы сокращения имен только на одну букву?
- ❑ Все ли слова вы сокращаете согласованно?
- ❑ Легко ли произнести выбранные имена?
- ❑ Избегаете ли вы имен, допускающих неверное прочтение или произношение?
- ❑ Документируете ли вы короткие имена при помощи таблиц преобразования?

**Распространенные проблемы именования. Избежали ли вы...**

- ❑ ...имен, которые вводят в заблуждение?
- ❑ ...имен с похожими значениями?
- ❑ ...имен, различающихся только одним или двумя символами?
- ❑ ...имен, имеющих похожее звучание?
- ❑ ...имен, включающих цифры?
- ❑ ...имен, намеренно написанных с ошибками с целью сокращения?
- ❑ ...имен, при написании которых люди часто допускают ошибки?
- ❑ ...имен, конфликтующих с именами методов из стандартных библиотек или предопределенными именами переменных?
- ❑ ...совершенно произвольных имен?
- ❑ ...символов, которые можно спутать с другими символами?

# Ключевые моменты

- Выбор хороших имен переменных — одно из главных условий понятности программы. С отдельными типами переменных — например, с индексами циклов и переменными статуса — связаны свои принципы именования.
- Имена должны быть максимально конкретны. Имена, которые из-за невыразительности или обобщенности можно использовать более чем с одной целью, обычно являются плохими.
- Конвенции именования позволяют провести различие между локальными данными, данными класса и глобальными данными, а также между именами типов, именованных констант, перечислений и переменных.
- Над каким бы проектом вы ни работали, вам следует принять конвенцию именования переменных. При выборе типа конвенции следует учитывать размер программы и число работающих над ней программистов.
- Современные языки программирования позволяют отказаться от сокращения имен. Если вы все-таки сокращаете имена, регистрируйте аббревиатуры в словаре проекта или используйте стандартизированные префиксы.
- За чтением кода программисты проводят гораздо больше времени, чем за его написанием. Выбирайте имена так, чтобы они облегчали чтение кода, пусть даже за счет удобства его написания.

ГЛАВА 12

# Основные типы данных

http://cc2e.com/1278

**Содержание**



**Связанные темы**

- Присвоение имен данным: глава 11
- Нестандартные типы данных: глава 13
- Общие вопросы использования переменных: глава 10
- Описание форматов данных: подраздел «Размещение объявлений данных» раздела 31.5
- Документирование переменных: подраздел «Комментирование объявлений данных» раздела 32.5
- Создание классов: глава 6

Основные типы данных являются базовыми блоками для построения остальных типов данных. Эта глава содержит советы по применению чисел вообще, целых чисел, чисел с плавающей запятой, символов и строк, логических переменных, перечислимых типов, именованных констант и массивов. В заключительном разделе этой главы рассказано о создании собственных типов данных.

Эта глава содержит рекомендации по предупреждению главных ошибок в применении основных типов данных. Если вы уже ознакомились с основами, переходите к концу главы, к обзору перечня допускаемых ошибок, а также к обсуждению нестандартных типов данных в главе 13.

# 12.1. Числа в общем

Далее дано несколько рекомендаций, позволяющих сократить число ошибок при использовании чисел.

***Избегайте «магических чисел»*** Магические числа — это обычные числа, такие как *100* или *47524*, которые появляются в программе без объяснений. Если вы программируете на языке, поддерживающем именованные константы, используйте их вместо магических чисел. Если вы не можете применить именованные константы, применяйте глобальные переменные, когда это возможно.

**Перекрестная ссылка** О применении именованных констант вместо магических чисел см. раздел 12.7.

Исключение магических чисел дает три преимущества.

- Изменения можно сделать более надежно. Если вы используете именованные константы, нет нужды искать каждое из чисел *100*, и вы не измените по ошибке те из этих чисел, что ссылаются на что-либо иное.
- Изменения сделать проще. Когда максимальное число элементов меняется со *100* на *200*, используя магические числа, вы должны найти каждое число *100* и изменить его на *200*. Если вы используете *100+1* или *100-1*, вы также должны найти и изменить все числа *101* и *99* на *201* и *199* соответственно. При использовании именованных констант вы просто меняете определение константы со *100* на *200* в одном месте.
- Ваша программа лучше читается. Конечно, в выражении:

  ```
  for i = 0 to 99 do ...
  ```

  можно предположить, что *99* определяет максимальное число элементов. А вот выражение:

  ```
  for  i = 0 to MAX_ENTRIES-1 do ...
  ```

  не оставляет на этот счет сомнений. Даже если вы уверены, что это число никогда не изменится, применяя именованные константы, вы получите более читабельную программу.

***Применяйте жестко заданные нули и единицы по необходимости*** Значения *0* и *1* используются для инкремента, декремента, а также в начале циклов при нумерации первого элемента массива. *0* в конструкции:

```
for i = 0 to CONSTANT do ...
```

вполне приемлем, так же как *1* в выражении:

```
total = total + 1
```

Вот хорошее правило: используйте в программе как константы только *0* и *1*, а любые другие числа определите как литералы с понятными именами.

***Ошибки деления на ноль*** Каждый раз, когда вы пользуетесь символом деления (/ в большинстве языков), думайте о том, может ли в знаменателе оказаться *0*. Если такая возможность существует, напишите код, предупреждающий появление ошибки деления на 0.

***Выполняйте преобразования типов понятно*** Убедитесь, что кто-нибудь, читая вашу программу, поймет преобразования между разными типами данных, которые в ней встречаются. На языке C++ вы могли бы написать:

```
y = x + (float) i
```

а на Microsoft Visual Basic:

```
y = x + CSng( i )
```

Эта практика поможет обеспечить однозначность ваших преобразований — разные компиляторы по-разному конвертируют, а при таком подходе вы гарантированно получите то, что ожидали.

**Перекрестная ссылка** Вариант этого примера см. в разделе 12.3.

***Избегайте сравнений разных типов*** Если $x$ — число с плавающей запятой, а $i$ — целое, проверка:

```
if ( i = x ) then ...
```

почти гарантированно не сработает. К тому времени, когда компилятор определит каждый тип, который он хочет задействовать для сравнения, преобразует один из типов в другой, произведет ряд округлений и вычислит ответ, вы будете рады, если ваша программа вообще работает. Сделайте преобразования вручную, так чтобы компилятор мог сравнить два числа одного и того же типа, и точно знайте, что нужно сравнивать.

***Обращайте внимание на предупреждения вашего компилятора*** Многие современные компиляторы сообщают о наличии разных типов чисел в одном выражении. Обращайте на это внимание! Каждый программист рано или поздно просит кого-нибудь помочь выследить надоедливую ошибку, а выясняется, что о ней все время предупреждал компилятор. Профессионалы высокого класса пишут свои программы так, чтобы исключить все предупреждения компилятора. Легче предоставить работу компилятору, чем выполнять ее самому.

## 12.2. Целые числа

Учитывайте следующие рекомендации при применении целых чисел.

***Проверяйте целочисленность операций деления*** Когда используются целые числа, выражение 7/10 не равно 0,7. Оно обычно равно 0 или минус бесконечности, или ближайшему целому, или... ну, вы понимаете. Результат зависит от выбранного языка. Это же относится и к промежуточным результатам. В реальном мире 10 * (7/10) = (10*7) / 10 = 7. Но не в мире целочисленной арифметики. 10 * (7/10) равно 0, потому что целочисленное деление (7/10) равно 0. Простейший способ исправить положение — преобразовать его так, чтобы операции деления выполнялись последними: (10*7) / 10.

***Проверяйте переполнение целых чисел*** При выполнении умножения или сложения необходимо принимать во внимание наибольшие возможные значения целых чисел. Для целого числа без знака это обычно $2^{32}-1$, а иногда и $2^{16}-1$, или 65 535. Проблема возникает, когда вы умножаете два числа, в результате чего получается

число большее, чем максимально возможное целое. Скажем, если вы умножаете 250 * 300, правильным ответом будет 75 000. Но если максимальное целое — 65 535, то, возможно, из-за переполнения вы получите 9464 (75 000 – 65 536 = 9464). Вот интервалы значений для часто встречающихся целых типов (табл. 12-1):

**Табл. 12-1. Интервалы значений некоторых целых типов**

| Целый тип | Интервал |
|---|---|
| 8-битный со знаком | От –128 до 127 |
| 8-битный без знака | От 0 до 255 |
| 16-битный со знаком | От –32 768 до 32 767 |
| 16-битный без знака | От 0 до 65 535 |
| 32-битный со знаком | От –2 147 483,648 до 2 147 483 647 |
| 32-битный без знака | От 0 до 4 294 967 295 |
| 64-битный со знаком | От –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 |
| 64-битный без знака | От 0 до 18 446 744 073 709 551 615 |

Простейший способ предотвращения целочисленного переполнения — просмотр каждого члена арифметического выражения с целью представить наибольшее возможное значение, которое он может принимать. Так, если в целочисленном выражении $m = j * k$, наибольшим значением для $j$ будет 200, а для $k$ — 25, то максимальным значением для $m$ будет $200 * 25 = 5\ 000$. Для 32-разрядных машин это вполне допустимо, так как максимальным целым является 2 147 483 647. С другой стороны, если максимально возможное значение для $j$ — это 200 000, а для $k$ — 100 000, то значение $m$ может достигать $200\ 000 * 100\ 000 = 20\ 000\ 000\ 000$. Это уже неприемлемо, так как 20 000 000 000 больше, чем 2 147 483 647. В этом случае для размещения наибольшего возможного значения $m$ вам придется использовать 64-битные целые или числа с плавающей запятой.

Кроме того, учитывайте будущее развитие программы. Если $m$ никогда не будет больше 5 000 — отлично. Но если ожидается, что $m$ будет постоянно расти на протяжении нескольких лет, примите это во внимание.

***Проверяйте на переполнение промежуточные результаты*** Число, получаемое в конце вычислений, — не единственное, о котором следует беспокоиться. Представьте, что у вас есть такой код:

**Пример переполнения промежуточных результатов (Java)**

```
int termA = 1000000;
int termB = 1000000;
int product = termA * termB / 1000000;
System.out.println( "( " + termA + " * " + termB + " ) / 1000000 = " + product );
```

Вы можете подумать, что значение *Product* вычисляется как *(100 000*100 000) / 100 000* и поэтому равно *100 000*. Но программе приходится вычислять промежуточное значение *100 000*100 000* до того, как будет выполнено деление на *100 000*, а это значит, что нужно хранить такое большое число, как *1 000 000 000 000*. Угадайте, что получится? Вот результат:

```
( 1000000 * 1000000 ) / 1000000 = -727
```

Если значение целых чисел в вашей системе не превышает 2 147 483 647, промежуточный результат слишком велик для целого типа данных. В такой ситуации промежуточный результат, который должен быть равен *1 000 000 000 000*, на самом деле равен *727 379 968*, поэтому, когда вы делите его на *100 000*, вы получаете *-727* вместо *100 000*.

Вы можете решить проблему переполнения промежуточных результатов так же, как и в случае целочисленного переполнения: изменив тип на длинное целое или число с плавающей запятой.

## 12.3. Числа с плавающей запятой

Главная особенность применения чисел с плавающей запятой в том, что многие дробные десятичные числа не могут быть точно представлены с помощью нулей и единиц, используемых в цифровом компьютере. В бесконечных десятичных дробях, таких как 1/3 или 1/7, обычно сохраняется только 7 или 15 цифр после запятой. В моей версии Microsoft Visual Basic 32-битное представление дроби 1/3 в виде числа с плавающей запятой равно 0,33333330. То есть точность ограничена 7 цифрами. Такая точность достаточна для большинства случаев, но все же способна иногда вводить в заблуждение.

Вот несколько рекомендаций по использованию чисел с плавающей запятой.

**Перекрестная ссылка** Книги, содержащие алгоритмы решения этих проблем, см. в подразделе «Дополнительные ресурсы» раздела 10.1.

***Избегайте сложения и вычитания слишком разных по размеру чисел*** Для 32-битной переменной с плавающей запятой сумма 1 000 000,00 + 0,1, вероятно, будет равна 1 000 000,00, так как в 32 битах недостаточно значимых цифр, чтобы охватить интервал между 1 000 000 и 0,1. Аналогично 5 000 000,02 – 5 000 000,01, вероятно, равно 0,0.

Решение? Если вам нужно складывать настолько разные по величине числа, сначала отсортируйте их, а затем складывайте, начиная с самых маленьких значений. Аналогично, если вам надо сложить бесконечный ряд значений, начните с наименьшего члена, т. е. суммируйте члены в обратном порядке. Это не решит проблемы округления, но минимизирует их. Многие алгоритмические книги предлагают решения для таких случаев.

1 равен 2 для достаточно больших значений 1.

*Аноним*

***Избегайте сравнений на равенство*** Числа с плавающей запятой, которые должны быть равны, на самом деле равны не всегда. Главная проблема в том, что два разных способа получить одно и то же число не всегда приводят к одинаковому результату. Так, если 10 раз сложить 0,1, то 1,0 получается только в редких случаях. Следующий пример содержит две переменных (*nominal* и *sum*), которые должны быть равны, но это не так.

**Пример неправильного сравнения чисел с плавающей точкой (Java)**

Переменная *nominal* — 64-битное вещественное число.

```
double nominal = 1.0;
double sum = 0.0;
```

```
for ( int i = 0; i < 10; i++ ) {
```

*sum* вычисляется как 10*0,1. Она должна быть равна 1,0.

```
    sum += 0.1;
}
```

Здесь неправильное сравнение.

```
if ( nominal == sum ) {
    System.out.println( "Numbers are the same." );
}
else {
    System.out.println( "Numbers are different." );
}
```

Как вы, наверное, догадались, программа выводит:

```
Numbers are different.
```

Вывод каждого значения *sum* в цикле *for* выглядит так:

```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
```

Таким образом, хорошей идеей будет найти альтернативу операции сравнения на равенство для чисел с плавающей запятой. Один эффективный подход состоит в том, чтобы определить приемлемый интервал точности, а затем использовать логические функции для выяснения, достаточно ли близки сравниваемые значения. Для этого обычно пишется функция *Equals()*, которая возвращает *true*, если значения попадают в этот интервал, и *false* — в противном случае. На языке Java такая функция может выглядеть так:

### Пример метода для сравнения чисел с плавающей запятой (Java)

```
final double ACCEPTABLE_DELTA = 0.00001;
boolean Equals( double Term1, double Term2 ) {
   if ( Math.abs( Term1 - Term2 ) < ACCEPTABLE_DELTA ) {
      return true;
   }
   else {
      return false;
   }
}
```

**Перекрестная ссылка** Этот пример — доказательство того, что из каждого правила есть исключения. Переменные здесь содержат цифры в именах. Правило против использования цифр в именах переменных см. в разделе 11.7.

Если код в примере «неправильного сравнения чисел с плавающей запятой» изменить так, чтобы для сравнения использовался этот метод, новое выражение получит следующий вид:

```
if ( Equals( Nominal, Sum ) ) ...
```

При запуске теста программа выведет сообщение:

```
Numbers are the same.
```

В зависимости от требований вашего приложения использование жестко закодированного значения *ACCEPTABLE_DELTA* может быть недопустимо. Возможно, придется вычислять *ACCEPTABLE_DELTA* на основании размера двух сравниваемых чисел.

***Предупреждайте ошибки округления*** Проблемы с ошибками округления сходны с проблемами слишком разных по размеру чисел. У них одинаковые причины и похожие методики решения. Кроме того, далее перечислены способы решения проблем округления.

- Измените тип переменной на тип с большей точностью. Если вы используете числа с одинарной точностью, замените их числами с двойной точностью и т. д.
- Используйте двоично-десятичные переменные (binary coded decimal, BCD). BCD-числа обычно работают медленнее и требуют больше памяти для хранения, но предотвращают множество ошибок округления. Это особенно важно, если используемые переменные представляют собой доллары и центы или другие величины, которые должны точно балансироваться.

**Перекрестная ссылка** Как правило, конвертация в тип BCD минимально влияет на производительность. Если вы озабочены проблемой производительности, см. раздел 25.6.

- Измените тип с плавающей запятой на целые значения. Это такая самодельная замена BCD-переменных. Возможно, вам придется использовать 64-битные целые, чтобы получить нужную точность. Этот способ предполагает, что вы сами будете отслеживать дробные части чисел. Допустим, изначально вы вели учет денежных сумм, применяя числа с плавающей запятой, при этом центы указывались как дробная часть. Это обычный способ обработки долларов и центов. Когда вы переключаетесь на целые числа, вам нужно вести учет центов с помощью целых, а долларов — с помощью чисел, кратных 100 центам. Иначе говоря, вы умножаете сумму в долларах на 100 и храните центы в этой переменной в интервале от 0 до 99. Такое решение может показаться абсурдным, но оно эффективно и с точки зрения скорости, и с точки зрения точности. Вы можете упростить эти манипуляции, создав класс *DollarsAndCents*, скрывающий целое представление чисел и предоставляющий необходимые числовые операции.

***Проверяйте поддержку специальных типов данных в языке и дополнительных библиотеках*** Некоторые языки, включая Visual Basic, предоставляют такие типы данных, как *Currency*, предназначенные для данных, чувствительных к ошибкам округления. Если ваш язык содержит встроенный тип данных, предоставляющий такую функциональность, используйте его!

# 12.4. Символы и строки

Этот раздел предлагает несколько советов по использованию строк. Первый относится к строкам во всех языках.

***Избегайте магических символов и строк*** Магические символы — это литеральные символы (например, '*A*'), а магические строки — это литеральные строки (например, *"Gigamatic Accounting Program"*), которые разбросаны по всей программе. Если ваш язык программирования поддерживает применение именованных констант, то лучше задействуйте их. В противном случае используйте глобальные переменные. Далее перечислено несколько причин, по которым надо избегать литеральных строк.

**Перекрестная ссылка** Вопросы использования магических символов и строк аналогичны вопросам применения магических чисел (см. раздел 12.1).

- Для таких часто встречающихся строк, как имя программы, названия команд, заголовки отчетов и т. п., вам может понадобиться поменять содержимое. Например, *"Gigamatic Accounting Program"* в более поздней версии может измениться на *"New and Improved! Gigamatic Accounting Program"*.
- Все большее значение приобретают международные рынки, и строки, сгруппированные в файле ресурсов переводить гораздо легче, чем раскиданные по всей программе.
- Строковые литералы обычно занимают много места. Они используются для меню, сообщений, экранов помощи, форм ввода и т. д. Если их слишком много, они выходят из-под контроля и вызывают проблемы с памятью. Во многих системах объем памяти, занимаемый строками, не является причиной для беспокойства. Однако при программировании встроенных систем и других приложений, в которых каждый байт на счету, проблему хранения строк легче решить, если эти строки относительно независимы от кода.
- Символьные и строковые литералы могут быть загадочными. Комментарии или именованные константы проясняют ваши намерения. В следующем примере смысл *0x1B* неясен. Константа *ESCAPE* делает значение более понятным.

**Пример сравнений с использованием строк (C++)**

```
Плохо!
if ( input_char == 0x1B ) ...

Лучше!
if ( input_char == ESCAPE ) ...
```

***Следите за ошибками завышения/занижения на единицу*** Поскольку подстроки могут индексироваться аналогично массивам, не забывайте об ошибках завышения/занижения на 1, которые приводят к чтению или записи за концом строки.

***Узнайте, как ваш язык и система поддерживают Unicode*** В некоторых языках, например в Java, все строки хранятся в формате Unicode. В других — таких, как C и C++ — работа со строками в Unicode требует применения отдельного набора функций. Преобразование между Unicode и другими наборами символов часто необходимо для взаимодействия со стандартными библиотеками и библиотеками сторонних про-

http://cc2e.com/1285

изводителей. Если часть строк не будет поддерживать Unicode (скажем, в C или C++), как можно раньше решите, стоит ли вообще использовать символы Unicode. Если вы решились на это, подумайте, где и когда будете это делать.

***Разработайте стратегию интернационализации/локализации в ранний период жизни программы*** Вопросы, связанные с интернационализацией, относятся к разряду ключевых. Решите, будут ли все строки храниться во внешних ресурсах и будет ли создаваться отдельный вариант программы для каждого языка или конкретный язык будет определяться во время выполнения.

http://cc2e.com/1292

***Если вам известно, что нужно поддерживать только один алфавит, рассмотрите вариант использования набора символов ISO 8859*** Для приложений, использующих только один алфавит (например, английский), которым не надо поддерживать несколько языков или какой-либо идеографический язык (такой как письменный китайский), расширенный ASCII-набор стандарта ISO 8859 — хорошая альтернатива символам Unicode.

***Если вам необходимо поддерживать несколько языков, используйте Unicode*** Unicode обеспечивает более полную поддержку международных наборов символов, чем ISO 8859 или другие стандарты.

***Выберите целостную стратегию преобразования строковых типов*** Если вы используете несколько строковых типов, общим подходом, помогающим хранить строковые типы в порядке, будет хранение всех строк программы в одном формате и преобразование их в другой формат как можно ближе к операциям ввода и вывода.

## Строки в языке C

Строковый класс в стандартной библиотеке шаблонов C++ решил большинство проблем со строками языка C. А тот, кто напрямую работает с C-строками, ниже узнает о способах избежать часто встречающихся ошибок.

***Различайте строковые указатели и символьные массивы*** Проблемы со строковыми указателями и символьными массивами возникают из-за способа обработки строк в C. Учитывайте различия между ними в двух случаях.

- Относитесь с недоверием к строковым выражениям, содержащим знак равенства. Строковые операции в C практически всегда выполняются с помощью *strcmp()*, *strcpy()*, *strlen()* и аналогичных функций. Знаки равенства часто сигнализируют о каких-то ошибках в указателях. Присваивание в C не копирует строковые константы в строковые переменные. Допустим, у нас есть выражение:

  ```
  StringPtr = "Some Text String";
  ```

  В этом случае *"Some Text String"* — указатель на литеральную текстовую строку, и это присваивание просто присвоит указателю *StringPtr* адрес данной строки. Операция присваивания не копирует содержимое в *StringPtr*.
- Используйте соглашения по именованию, чтобы различать переменные — массивы символов и указатели на строки. Одно из общепринятых соглашений — использование *ps* как префикса для обозначения указателя на строку, и *ach* — как префикса для символьного массива. И хотя они не всегда ошибочны, от-

носитесь все-таки с подозрением к выражениям, включающим переменные с обоими префиксами.

***Объявляйте для строк в стиле С длину, равную КОНСТАНТА+1*** В C и C++ ошибки завышения на 1 в С-строках — обычное явление, потому что очень легко забыть, что строка длины *n* требует для хранения *n + 1* байт, и не выделить место для нулевого терминатора (байта в конце строки, установленного в 0). Эффективный способ решения этой проблемы — использовать именованные константы при объявлении всех строк. Суть в том, что именованные константы применяются всегда одинаково: Сначала длина строки объявляется как *КОНСТАНТА+1*, а затем *КОНСТАНТА* используется для обозначения длины строки во всем остальном коде. Вот пример:

**Пример правильных объявлений строк (C)**

```
/* Объявляем строку длиной «константа+1».
   Во всех остальных местах программы используем «константа»,
   а не «константа +1». */

Эта строка объявлена с длиной NAME_LENGTH +1.
char name[ NAME_LENGTH + 1 ] = { 0 }; /* Длина строки – NAME_LENGTH */

...
/* Пример 1: Заполняем строку символами 'A', используя константу NAME_LENGTH
   для определения количества символов 'A', которые можно скопировать.
   Заметьте: используется NAME_LENGTH, а не NAME_LENGTH + 1. */

В действиях со строкой NAME_LENGTH используется здесь...
for ( i = 0; i < NAME_LENGTH; i++ )
   name[ i ] = 'A';
...

/* Пример 2: Копируем другую строку в первую, используя константу
   для определения максимальной длины, которую можно копировать. */

...и здесь.
strncpy( name, some_other_name, NAME_LENGTH );
```

Если у вас не будет соглашения по этому поводу, иногда вы будете объявлять строки длиной *NAME_LENGTH*, а в операциях использовать *NAME_ LENGTH-1*; а иногда вы будете объявлять строки длиной *NAME_LENGTH+1* и работать с *NAME_LENGTH*. Каждый раз при использовании строки вам придется вспоминать, как вы ее объявили.

Если же вы всегда одинаково объявляете строки, думать, как работать с каждой из них, не надо, и вы избежите ошибок из-за того, что забыли особенность объявления данной строки. Выработка соглашения минимизирует умственную перегрузку и ошибки при программировании.

***Инициализируйте строки нулем во избежание строк бесконечной длины*** Язык С определяет конец строки путем поиска нулевого терминатора — байта в конце строки, установленного в 0. Какой предполагалась длина строки, зна-

**Перекрестная ссылка** Подробнее об инициализации данных см. раздел 10.3.

чения не имеет: C никогда не найдет ее конец, если не найдет нулевой байт. Если вы забыли поместить нулевой байт в конец строки, строковые операции могут работать не так, как вы ожидаете.

Вы можете предупредить появление бесконечных строк двумя способами. Во-первых, при объявлении инициализируйте символьные массивы *0*:

**Пример правильного объявления символьного массива (C)**

```
char EventName[ MAX_NAME_LENGTH + 1 ] = { 0 };
```

Во-вторых, при динамическом создании строк инициализируйте их *0*, используя функцию *calloc()* вместо *malloc()*. Функция *calloc()* выделяет память и инициализирует ее *0*. *malloc()* выделяет память без инициализации, поэтому вы рискуете, используя память, выделенную с помощью *malloc()*.

**Перекрестная ссылка** О массивах см. раздел 12.8.

***Используйте в C массивы символов вместо указателей*** Если объем занимаемой памяти некритичен (а часто так и есть), объявляйте все строковые переменные как массивы символов. Это поможет избежать проблем с указателями, а компилятор будет выдавать больше предупреждений в случае неправильных действий.

***Используйте* strncpy() *вместо* strcpy() *во избежание строк бесконечной длины*** Строковые функции в C существуют в опасной и безопасной версиях. Более опасные функции, такие как *strcpy()* и *strcmp()*, продолжают работу до обнаружения нулевого терминатора. Их более безобидные спутники — *strncpy()* и *strncmp()* — принимают максимальную длину в качестве параметра, так что, даже если строки будут бесконечными, ваши вызовы функций не зациклятся.

## 12.5. Логические переменные

Логические или булевы переменные сложно использовать неправильно, а их вдумчивое применение сделает вашу программу аккуратней.

**Перекрестная ссылка** Об использовании комментариев для документирования программы см. главу 32.

***Используйте логические переменные для документирования программы*** Вместо простой проверки логического выражения вы можете присвоить его значение переменной, которая сделает смысл теста очевидным. Например, в этом фрагменте из условия *if* не ясно, выполняется ли проверка завершения, ошибочной ситуации или чего-то еще:

**Перекрестная ссылка** Пример использования логической функции для документирования программы см. в подразделе «Упрощение сложных выражений» раздела 19.1.

**Пример логического условия, чье назначение неочевидно (Java)**

```
if ( ( elementIndex < 0 ) || ( MAX_ELEMENTS < elementIndex ) ||
   ( elementIndex == lastElementIndex )
   ) {
   ...
}
```

В следующем фрагменте применение логических переменных делает назначение *if*-проверки яснее:

**Пример логического условия, чье назначение понятно (Java)**

```
finished = ( ( elementIndex < 0 ) || ( MAX_ELEMENTS < elementIndex ) );
repeatedEntry = ( elementIndex == lastElementIndex );
if ( finished || repeatedEntry ) {
   ...
}
```

***Используйте логические переменные для упрощения сложных условий***
Чтобы правильно закодировать сложное условие, часто приходится делать несколько попыток. Когда через некоторое время нужно модифицировать это условие, бывает сложно разобраться, что же оно проверяет. Логические переменные могут упростить проверку. В предыдущем примере программа на самом деле проверяет два условия: завершено ли выполнение метода и выполняется ли этот метод повторно. Создав логические переменные *finished* и *repeatedEntry*, вы упрощаете *if*-проверку: теперь ее легче читать, легче изменять, и она меньше подвержена ошибкам.

Вот другой пример сложного условия:

**Пример сложного условия (Visual Basic)**

```
If ( ( document.AtEndOfStream() ) And ( Not inputError ) ) And _
   ( ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES ) ) And _
   ( Not ErrorProcessing() ) Then
   ' делаем что-то
   ...
End If
```

Условие в примере достаточно запутанное, но все же часто встречающееся. Оно налагает на читателя тяжелую умственную нагрузку. Могу предположить, что вы даже не попытаетесь разобраться в этой *if*-проверке, а посмотрите и скажете: «Разберусь с этим позже, если и впрямь понадобится». Обратите внимание на эту мысль, ведь это абсолютно то же самое, что сделают другие люди, читая ваш код, содержащий подобные условия.

А вот как переписать этот код, используя логические переменные, добавленные для упрощения условия:

**Пример упрощенного условия (Visual Basic)**

```
allDataRead = ( document.AtEndOfStream() ) And ( Not inputError )
legalLineCount = ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES )

Вот упрощенная проверка.
If ( allDataRead ) And ( legalLineCount ) And ( Not ErrorProcessing() ) Then
   ' делаем что-то
   ...
End If
```

Вторая версия проще. Думаю, вы легко прочитаете логическое выражение в *if*-проверке.

***Создайте свой логический тип в случае необходимости*** Некоторые языки, такие как C++, Java и Visual Basic, имеют предопределенный логический тип, другие — скажем, C — не имеют. В языках, подобных C, вы можете определить свой логический тип. В C это можно сделать так:

**Пример определения типа *BOOLEAN* с помощью обычного *typedef* на языке C**

```
typedef int BOOLEAN;
```

Или можно это сделать, определив дополнительно значения *true* и *false*:

**Пример определения типа *Boolean* с помощью *Enum* на языке C**

```
enum Boolean {
   True=1,
   False=(!True)
};
```

Объявление переменных как *BOOLEAN*, а не *int* делает их последующее использование более очевидным, а вашу программу — самодокументируемой.

## 12.6. Перечислимые типы

Перечислимым называется тип данных, который позволяет описать на естественном языке каждый элемент класса или объекта. Перечислимые типы реализованы в C++ и Visual Basic и обычно используются, когда вы знаете все возможные значения переменной и хотите выразить их словами. Вот примеры перечислимых типов на Visual Basic:

**Примеры перечислимых типов (Visual Basic)**

```
Public Enum Color
   Color_Red
   Color_Green
   Color_Blue
End Enum

Public Enum Country
   Country_China
   Country_England
   Country_France
   Country_Germany
   Country_India
   Country_Japan
   Country_Usa
End Enum

Public Enum Output
   Output_Screen
   Output_Printer
```

```
   Output_File
End Enum
```

Перечислимые типы — мощная альтернатива устаревшим схемам, в которых вы подробно расписываете: «1 — это красный, 2 — зеленый, 3 — голубой...». В связи с этим можно предложить следующие принципы применения таких типов.

***Используйте перечислимые типы для читабельности*** Вместо выражений вроде:

```
if chosenColor = 1
```

вы можете написать более читабельную фразу, например:

```
if chosenColor = Color_Red
```

Каждый раз, когда вы видите числовую константу, подумайте: не заменить ли ее перечислимым типом.

Перечислимые типы особенно полезны для определения параметров методов. Кто поймет, что означают параметры этой функции?

**Пример вызова функции, в котором стоило бы использовать перечислимые типы (C++)**

```
int result = RetrievePayrollData( data, true, false, false, true );
```

А параметры в этом вызове функции гораздо понятнее:

**Пример вызова функции, использующей перечислимые типы для читабельности (C++)**

```
int result = RetrievePayrollData(
   data,
    EmploymentStatus_CurrentEmployee,
    PayrollType_Salaried,
    SavingsPlan_NoDeduction,
    MedicalCoverage_IncludeDependents
);
```

***Используйте перечислимые типы для надежности*** В некоторых языках (особенно в Ada) перечислимые типы заставляют компилятор выполнять более тщательную проверку типов, чем при работе с целыми значениями и константами. При использовании именованных констант компилятору неоткуда узнать, что единственные возможные значения переменной — это *Color_Red, Color_Green* и *Color_Blue*. Компилятор не будет возражать, встретив выражения вроде *color = Country_England* или *country = Output_Printer*. Если вы используете перечислимые типы, то при объявлении переменной типа *Color* компилятор позволит присвоить ей только значения *Color_Red, Color_Green* или *Color_Blue*.

***Используйте перечислимые типы для модифицируемости*** Перечислимые типы позволяют вам легко модифицировать код. Обнаружив пробел в своей схеме «1 — это красный, 2 — зеленый, 3 — голубой», придется пройти по коду и изменить все вхождения *1*, *2*, *3* и т. д. А используя перечислимые типы, вы можете

продолжить добавление элементов в список, просто поместив их в определение типа и перекомпилировав программу.

***Используйте перечислимые типы как альтернативу логическим переменным*** Очень часто логической переменной недостаточно для представления необходимых значений. Представьте, что у вас есть метод, возвращающий *True*, если он успешно выполнил свою задачу, и *False* в противном случае. Позднее вы можете обнаружить, что на самом деле у вас есть два варианта *False*. Первый означает, что задание не выполнено, но это повлияло только на сам метод; второй — что невыполнение задания привело к фатальной ошибке, которую надо передать в остальную часть программы. В этом случае перечислимый тип со значениями *Status_Success, Status_Warning* и *Status_FatalError* подходит больше, чем логический тип со значениями *true* и *false*. Эту схему очень легко расширять новыми вариантами успешного или неудачного выполнения.

***Проверяйте некорректные значения*** Когда вы используете перечислимый тип в условиях *if* или *case*, проверяйте появление недопустимых значений. В переключателе *case* для перехвата неправильных значений применяется оператор *else*:

**Хороший пример проверки некорректных значений в перечислимом типе (Visual Basic)**

```
Select Case screenColor
   Case Color_Red
      ...
   Case Color_Blue
      ...
   Case Color_Green
      ...
```

Здесь выполняется проверка неправильного значения.

```
   Case Else
      DisplayInternalError( False, "Internal Error 752: Invalid color." )
End Select
```

***Настройте первый и последний элемент перечислимого типа для использования в качестве границ циклов*** Определение первого и последнего элемента в перечислении в виде *Color_First, Color_Last, Country_First, Country_Last* и т. д., позволяет писать циклы, проходящие по всем элементам типа. Вы настраиваете свой перечислимый тип, используя явные значения:

**Пример установки значений *First* и *Last* перечислимого типа (Visual Basic)**

```
Public Enum Country
   Country_First = 0
   Country_China = 0
   Country_England = 1
   Country_France = 2
   Country_Germany = 3
   Country_India = 4
   Country_Japan = 5
```

```
   Country_Usa = 6
   Country_Last = 6
End Enum
```

Теперь элементы *Country_First* и *Country_Last* могут служить как границы цикла:

**Хороший пример циклического вызова элементов перечисления (Visual Basic)**

```
' Вычисляем курсы валют по отношению к долларам США.
Dim usaCurrencyConversionRate( Country_Last ) As Single
Dim iCountry As Country
For iCountry = Country_First To Country_Last
    usaCurrencyConversionRate( iCountry ) = ConversionRate( Country_Usa, iCountry )
Next
```

***Зарезервируйте первый элемент перечислимого типа как недопустимый*** При объявлении перечислимого типа зарезервируйте его первый элемент в качестве недопустимого значения. Большинство компиляторов присваивает первому элементу перечисления значение *0*. Объявление некорректным элемента, приравненного *0*, поможет выявить неправильно проинициализированные переменные, так как они вероятнее всего будут равны *0*, чем любому другому неправильному значению.

Вот как при этом подходе будет выглядеть перечислимый тип *Country*:

**Пример объявления первого элемента перечисления недопустимым (Visual Basic)**

```
Public Enum Country
   Country_InvalidFirst = 0
   Country_First = 1
   Country_China = 1
   Country_England = 2
   Country_France = 3
   Country_Germany = 4
   Country_India = 5
   Country_Japan = 6
   Country_Usa = 7
   Country_Last = 7
End Enum
```

***Точно укажите в стандартах кодирования для проекта, как должны использоваться первый и последний элементы, и неукоснительно придерживайтесь этого*** Использование в перечислениях элементов *InvalidFirst, First* и *Last* может сделать объявления массивов и операторы циклов читабельнее. Но есть вероятность неразберихи в таких вопросах: с чего начинаются значимые элементы перечисления — с *0* или *1* и является ли допустимыми первый и последний элементы? Если используется эта технология, стандарты кодирования для проекта в целях уменьшения количества ошибок должны требовать применения элементов *InvalidFirst, First* и *Last* для всех без исключения перечислений.

***Помните о подводных камнях в присваивании явных значений элементам перечисления*** Некоторые языки позволяют присваивать конкретные значения элементам перечисления:

**Пример явного присваивания значений элементам перечисления (C++)**

```
enum Color {
   Color_InvalidFirst = 0,
   Color_First = 1,
   Color_Red = 1,
   Color_Green = 2,
   Color_Blue = 4,
   Color_Black = 8,
   Color_Last = 8
};
```

Если в этом примере вы объявите переменную цикла типа *Color* и попробуете пройти по всем элементам *Color*, то этой переменной наряду с допустимыми значениями 1, 2, 4 и 8 будут присваиваться и недопустимые — 3, 5, 6 и 7.

**Перекрестная ссылка** На момент написания этой книги Java не поддерживает перечислимых типов. Но, возможно, уже делает это к моменту, когда вы это читаете.

## Если ваш язык не поддерживает перечислимые типы

Если в вашем языке нет перечислимых типов, их можно имитировать, используя глобальные переменные или классы. Например, такие объявления можно использовать в языке Java:

**Пример имитации перечислимых типов (Java)**

```
// Создание перечислимого типа Country.
class Country {
   private Country() {}
   public static final Country China = new Country();
   public static final Country England = new Country();
   public static final Country France = new Country();
   public static final Country Germany = new Country();
   public static final Country India = new Country();
   public static final Country Japan = new Country();
}

// Создание перечислимого типа Output.

class Output {
   private Output() {}
   public static final Output Screen = new Output();
   public static final Output Printer = new Output();
   public static final Output File = new Output();
}
```

Такие перечислимые типы делают вашу программу читабельнее, поскольку вы можете использовать открытые члены класса, например, *Country.England* и *Output.Screen,* вместо именованных констант. Именно такой способ создания перечислимых типов еще и обеспечивает безопасность типов, так как каждый тип объявлен как класс и компилятор будет проверять некорректные присваивания вроде *Output output = Country.England* (Bloch, 2001).

В языках, не поддерживающих классы, такого же эффекта можно достичь, аккуратно используя глобальные переменные для каждого элемента перечисления.

## 12.7. Именованные константы

Именованные константы аналогичны переменным за исключением того, что вы не можете изменить значение константы после ее инициализации. Именованные константы позволяют ссылаться на постоянные величины, например, максимальное количество работников, используя имя, а не число — *MAXIMUM_EMPLOYEES*, а не *1000*.

Применение именованных констант — это способ «параметризации» программы, т. е. размещение некоторой характеристики, которая может измениться, в параметре. В результате вам потребуется изменить его только в одном месте, а не по всей программе. Если вы когда-нибудь объявляли массив такой длины, какой, по-вашему, должно хватить для любых ситуаций, а потом выяснялось, что массив слишком мал, вы оцените значение именованных констант. Когда размер массива изменяется, вы меняете только определение константы, используемой для объявления массива. Такое «централизованное управление» имеет большое значение для того, чтобы сделать работу с ПО действительно приятной: упростить его поддержку и модификацию.

***Используйте именованные константы в объявлениях данных*** Именованные константы повышают читабельность и удобство сопровождения объявлений данных и выражений, которым необходимо знать размеры обрабатываемых данных. В следующем примере для описания длины телефонных номеров работников лучше использовать *LOCAL_NUMBER_LENGTH*, а не число 7.

**Хороший пример применения именованных констант в объявлениях данных (Visual Basic)**

```
Const AREA_CODE_LENGTH = 3

LOCAL_NUMBER_LEN GTH здесь объявляется как константа.
Const LOCAL_NUMBER_LENGTH = 7
...
Type PHONE_NUMBER
   areaCode( AREA_CODE_LENGTH ) As String

А здесь используется.
   localNumber( LOCAL_NUMBER_LENGTH ) As String
End Type
...
```

```
' Убедимся, что все символы в телефонном номере – это цифры.
```

И здесь тоже используется.

```
For iDigit = 1 To LOCAL_NUMBER_LENGTH
    If ( phoneNumber.localNumber( iDigit ) < "0" ) Or _
       ( "9" < phoneNumber.localNumber( iDigit ) ) Then
       ' Выполняем обработку ошибок.
       ...
```

Это простой пример, но вы вполне можете представить программу, в которой сведения о длине телефонных номеров требуются во многих местах.

На момент создания программы все работники живут в одной стране, поэтому вам нужно только семь цифр для их телефонных номеров. По мере расширения компании ее филиалы открываются в разных странах, и вам понадобятся более длинные телефонные номера. Если вы параметризовали эту длину, вам надо сделать изменение только в одном месте — в определении именованной константы *LOCAL_NUMBER_LENGTH*.

**Дополнительные сведения** О значении централизованного управления см. стр. 57–60 в книге «Software Conflict» (Glass, 1991).

Как вы, наверное, поняли, именованные константы делают сопровождение программы удобнее. Как правило, любая технология, централизующая управление объектами, подверженными изменениям, — это хороший способ уменьшить затраты на сопровождение (Glass, 1991).

***Избегайте литеральных значений, даже «безопасных»*** Как вы думаете, что в следующем цикле означает число *12*?

**Пример непонятного кода (Visual Basic)**

```
For i = 1 To 12
   profit( i ) = revenue( i ) - expense( i )
Next
```

Исходя из специфического содержимого кода, можно предположить, что выполняется цикл по 12 месяцам в году. Но вы *уверены*? Вы поставите на это свое собрание «Монти Пайтон»?

В этом случае вам не нужно использовать именованные константы для поддержки расширяемости: вряд ли число месяцев в году изменится в ближайшем будущем. Но если при написании кода остается хотя бы тень сомнения в его предназначении, развейте ее с помощью хорошо названной именованной константы, например, так:

**Пример более понятного кода (Visual Basic)**

```
For i = 1 To NUM_MONTHS_IN_YEAR
   profit( i ) = revenue( i ) - expense( i )
Next
```

Это уже лучше, но для завершения примера индекс цикла тоже нужно назвать более информативно:

#### Пример еще более понятного кода (Visual Basic)

```
For month = 1 To NUM_MONTHS_IN_YEAR
   profit( month ) = revenue( month ) - expense( month )
Next
```

Этот пример выглядит весьма неплохо, но мы можем сделать еще один шаг вперед, применив перечислимый тип:

#### Пример очень понятного кода (Visual Basic)

```
For month = Month_January To Month_December
   profit( month ) = revenue( month ) - expense( month )
Next
```

В последнем примере не может возникнуть никаких сомнений относительно назначения цикла. Даже если вы считаете, что литеральное значение безопасно, используйте вместо него именованную константу. Фанатично искорените литералы из вашего кода. С помощью текстового редактора выполните поиск *2, 3, 4, 5, 6, 7, 8* и *9*, чтобы убедиться, что вы не используете их случайно.

***Имитируйте именованные константы с помощью переменных или классов правильной области видимости*** Если ваш язык не поддерживает именованные константы, их можно создать. Подход, аналогичный приведенному выше Java-примеру, имитирующему перечислимые типы, позволяет получить преимущества использования именованных констант. Старайтесь применять обычные правила области видимости: отдавайте предпочтение локальной, классовой или глобальной области видимости именно в таком порядке.

**Перекрестная ссылка** Об имитации перечислимых типов см. подраздел «Если ваш язык не поддерживает перечислимые типы» раздела 12.6.

***Последовательно используйте именованные константы*** Опасно использовать для представления одной сущности именованные константы в одном месте и литералы в другом. Некоторые приемы программирования напрашиваются на ошибки, а этот просто доставляет вам ошибки на дом. Если значение именованной константы нужно изменить, вы сделаете это и подумаете, что выполнили все необходимые изменения. Вы не обратите внимания на жестко закодированные литералы, и ваша программа будет демонстрировать таинственные дефекты. Их устранение может потребовать так много усилий, что захочется схватить телефонную трубку и молить о помощи.

## 12.8. Массивы

Массивы — простейшие и наиболее часто используемые типы структурированных данных. В некоторых языках это единственный вид структурированных данных. Массивы состоят из группы элементов одинакового типа, доступ к которым осуществляется напрямую по индексу.

***Убедитесь, что все значения индексов массива не выходят за его границы*** Все проблемы с массивами так или иначе связаны с тем, что доступ к их элементам может осуществляться произвольно. Наиболее часто

возникающая проблема объясняется попыткой доступа к элементу по индексу, выходящему за пределы массива. В некоторых языках при этом генерируется ошибка, а в других — получаются причудливые и неожиданные результаты.

***Обдумайте применение контейнеров вместо массивов или рассматривайте массивы как последовательные структуры*** Некоторые именитые в компьютерной науке люди предлагали запретить произвольный доступ к массиву, заменив его последовательным (Mills and Linger, 1986). Аргументтруют они это тем, что произвольный доступ к массиву похож на случайные операторы *goto* в программе: их применение приводит к неаккуратному, подверженному ошибкам коду, в корректности которого сложно быть уверенным. Поэтому вместо массивов предлагается использовать множества, стеки и очереди, доступ к элементам которых выполняется последовательно.

Проведя небольшой эксперимент, Миллз (Mills) и Линджер (Linger) выяснили, что разработанный таким образом проект потребовал использования меньшего числа переменных и меньшего числа ссылок на эти переменные. То есть проект был относительно эффективнее, что привело к созданию более надежного ПО.

Рассмотрите вопрос использования контейнерных классов с последовательным доступом — наборов, стеков, очередей и т. п. — как альтернативу прежде, чем выбрать массив.

**Перекрестная ссылка** Вопросы применения массивов и циклов имеют много общего. Подробнее о циклах см. главу 16.

***Проверяйте конечные точки массивов*** Как бывает полезно продумать применение конечных точек в операторе цикла, так и вы сможете обнаружить немало ошибок, проверив крайние элементы массивов. Задайтесь вопросом, правильно ли выполняется доступ к первому элементу массива или случайно используется элемент перед ним либо после него. А что с последним элементом? Нет ли в коде ошибки потери единицы? И, наконец, спросите себя, правильно ли код обращается к элементам в середине массива.

***В многомерном массиве убедитесь, что его индексы используются в правильном порядке*** Очень легко написать *Array[ i ][ j ]*, имея в виду *Array[ j ][ i ]*, так что не жалейте времени для проверки правильного порядка индексов. Попробуйте использовать более значимые имена, чем *i* и *j*, когда их назначение не вполне очевидно.

***Остерегайтесь пересечения индексов*** При использовании вложенных циклов легко написать *Array[ j ]*, имея в виду *Array[ i ]*. Перемена мест индексов называется «пересечением индексов» (index cross-talk). Проверьте эту возможность. Опять же, используйте более значимые имена индексов, чем *i* и *j*, чтобы ошибки пересечения изначально сложнее было совершить.

***В языке C для работы с массивами используйте макрос ARRAY_LENGTH()*** Вы можете добавить гибкости вашей работе с массивами, определив макрос *ARRAY_LENGTH()*:

**Пример определения макроса *ARRAY_LENGTH()* на языке C**

```
#define ARRAY_LENGTH( x )   (sizeof(x)/sizeof(x[0]))
```

При выполнении операций над массивами для указания верхней границы используйте макрос *ARRAY_LENGTH()* вместо именованной константы. Например:

**Пример использования макроса *ARRAY_LENGTH()* для операций с массивами на языке C**

```
ConsistencyRatios[] =
   { 0.0, 0.0, 0.58, 0.90, 1.12,
   1.24, 1.32, 1.41, 1.45, 1.49,
   1.51, 1.48, 1.56, 1.57, 1.59 };
   ...

Вот здесь используется макрос.
for ( ratioIdx = 0; ratioIdx < ARRAY_LENGTH( ConsistencyRatios ); ratioIdx++ );
   ...
```

Этот способ особенно полезен для массивов неопределенного размера, как в этом примере. Если вы добавляете или удаляете элементы, вам не надо помнить об изменении именованной константы, определяющей размер массива. Разумеется, эта технология работает и с массивами заданного размера, но, используя этот подход, вам не всегда надо будет создавать дополнительные именованные константы для объявления массивов.

## 12.9. Создание собственных типов данных (псевдонимы)

Типы данных, определяемые программистом, — одна из наиболее мощных возможностей, позволяющих наиболее четко обозначить ваше понимание программы. Они защищают программу от непредвиденных изменений и упрощают ее прочтение, и все это — без необходимости проектировать, разрабатывать или тестировать новые классы. Если вы программируете на C, C++ или других языках, поддерживающих такие типы, задействуйте это преимущество!

Чтобы оценить возможности создания типов, представьте, что вы пишете программу для преобразования координат из системы *x, y, z* в широту, долготу и высоту. Вам кажется, что могут потребоваться числа с плавающей запятой двойной точности, но пока вы абсолютно в этом не уверены, предпочитаете писать программу, используя числа с одинарной точностью. Вы можете создать новый тип данных специально для координат, применив оператор *typedef* в C или C++ или его эквивалент в другом языке. Вот как вы определите такой тип в C++:

**Перекрестная ссылка** Во многих случаях лучше создавать класс, чем простой тип данных. Подробнее см. главу 6.

**Пример создания типа (C++)**

```
typedef float Coordinate;  // для координатных переменных
```

Это определение объявляет новый тип *Coordinate*, функционально идентичный типу *float*. Чтобы задействовать этот тип, вы просто объявляете с ним переменные точно так же, как и с любым предопределенным типом вроде *float*. Пример:

### Пример использования созданного типа (C++)

```
Routine1( ... ) {
   Coordinate latitude;      // широта в градусах
   Coordinate longitude;     // долгота в градусах
   Coordinate elevation;     // высота в метрах от центра Земли
   ...
}
...

Routine2( ... ) {
   Coordinate x;   // координата x в метрах
   Coordinate y;   // координата y в метрах
   Coordinate z;   // координата z в метрах
   ...
}
```

Здесь все переменные *latitude, longitude, elevation, x, y* и *z* объявлены с типом *Coordinate*.

Теперь допустим, что программа изменилась и вы выяснили, что все-таки нужны переменные с двойной точностью. Поскольку вы создали тип специально для координатных данных, все, что вам нужно изменить, — это определение типа. И сделать это вам необходимо только в одном месте — в выражении *typedef*. Вот как выглядит новое определение типа:

### Пример измененного определения типа (C++)

Первоначальный тип *float* заменен на *double*.

```
typedef double Coordinate;  // для координатных переменных
```

Вот еще один пример — теперь на языке Pascal. Представьте, что вы разрабатываете систему расчета заработной платы, в которой длина имен работников не превышает 30 символов. Пользователи сказали вам, что *ни у кого* нет имени длинней 30 символов. Закодируете ли вы число *30* по всей программе? Если да, то вы доверяете вашим пользователям гораздо больше, чем я — своим. Лучший подход состоит в определении типа для имен работников:

### Пример создания типа для имен работников (Pascal)

```
Type
   employeeName = array[ 1..30 ] of char;
```

Когда речь идет о строке или массиве, обычно разумно определить именованную константу, содержащую длину строки или массива, а затем задействовать ее в определении типа. Вы найдете в своей программе много мест, в которых стоит использовать константу, и это — первое из них. Вот как это выглядит:

### Пример лучшего создания типа (Pascal)

```
Const
```

```
Вот объявление именованной константы.
   NAME_LENGTH = 30;
   ...
Type
Здесь эта именованная константа используется.
   employeeName = array[ 1..NAME_LENGTH ] of char;
```

Еще более усовершенствованный пример может комбинировать идею создания собственных типов с технологией сокрытия информации. Порой сведения, которые вы хотите скрыть, и есть информация о типе данных.

Пример с координатами на C++ частично удовлетворяет принципу сокрытия информации. Если вы всегда будете использовать *Coordinate* вместо *float* или *double*, вы эффективно спрячете исходный тип данных. В C++ это практически все возможное сокрытие информации, которое язык позволяет сделать разработчику. Все последующие пользователи вашего кода должны соблюдать дисциплину и не смотреть на определение *Coordinate*. C++ предоставляет скорее фигуральную, а не буквальную возможность сокрытия информации.

Другие языки, например Ada, делают шаг вперед и поддерживают буквальное сокрытие информации. Вот как фрагмент кода для типа *Coordinate* будет выглядеть в модуле Ada, где он был объявлен:

**Пример сокрытия деталей реализации типа внутри модуля (Ada)**

```
package Transformation is
Это выражение объявляет Coordinate скрытым в данном модуле.
   type Coordinate is private;
   ...
```

Вот как тип *Coordinate* будет выглядеть в другом модуле, где он используется:

**Пример использования типа из другого модуля (Ada)**

```
with Transformation;
...
procedure Routine1(...) ...
   latitude:  Coordinate;
   longitude: Coordinate;
begin
   -- операторы, использующие широту и долготу
   ...
end Routine1;
```

Заметьте: тип *Coordinate* объявлен в модуле как *private*. Это значит, что единственная часть программы, которая знает определение типа *Coordinate*, — это закрытая часть модуля *Transformation*. При групповой разработке проекта вы можете распространить только спецификацию модуля, что затруднит программисту, работающему с другим модулем, просмотр исходного типа *Coordinate*. Информация будет

буквально спрятана. Такие языки, как C++, которые требуют распространять определение типа *Coordinate* в заголовочном файле, подрывают идею реального сокрытия информации.

Следующие примеры иллюстрируют несколько причин для создания собственных типов.

- **Упростить модификацию кода** Сделать новый тип легко, а это дает вам большую гибкость.
- **Избежать излишнего распространения информации** Явная типизация распространяет сведения о типе данных по всей программе вместо их централизации в одном месте. Это пример принципа сокрытия информации с целью достижения централизации, (см. раздел 6.2).
- **Увеличить надежность** В Ada вы можете объявлять типы как *type Age is range 0..99*. После этого компилятор генерирует проверки времени выполнения, чтобы удостовериться, что значение любой переменной типа *Age* всегда попадает в диапазон *0..99*.
- **Замаскировать недостатки языка** Если ваш язык не содержит необходимого предопределенного типа, вы можете создать его сами. Например, в C нет булева или логического типа. Этот недостаток легко исправить, создав тип:

```
typedef int Boolean;
```

## Почему приведены примеры создания типов на языках Pascal и Ada?

Языки Pascal и Ada сейчас подобны динозаврам, а языки, заменившие их, в основном гораздо практичнее. Однако в области определения простых типов мне кажется, что C++, Java и Visual Basic представляют случай трех шагов вперед и одного шага назад. В Ada такое объявление, как:

```
currentTemperature: INTEGER range 0..212;
```

содержит важную семантическую информацию, которую объявление:

```
int temperature;
```

не содержит. Если посмотреть глубже, то определение:

```
type Temperature is range 0..212;
...
currentTemperature:  Temperature;
```

позволяет компилятору удостовериться, что *currentTemperature* присваивается только другим переменным типа *Temperature*, и такая дополнительная прослойка безопасности требует минимального кодирования.

Естественно, программист может создать класс *Temperature*, чтобы реализовать те же семантические правила, автоматически предоставляемые в Ada, но между созданием простого типа данных в одну строку и созданием класса дистанция огромного размера. Зачастую программист будет использовать простой тип данных, но не станет делать дополнительных усилий для создания класса.

## Основные принципы создания собственных типов

Имейте в виду следующие принципы, когда решите создавать собственный тип.

***Создавайте типы с именами, отражающими их функциональность*** Избегайте имен типов, которые ссылаются на данные, лежащие в основе этих типов. Используйте имена, которые отражают те элементы реальной задачи, которые этот тип представляет. Определения из предыдущих примеров — понятно названные типы для координат и имен работников — это реальные сущности. Точно так же вы можете создавать типы для валюты, кодов платежей, возрастов и т. д., а именно для аспектов действительно существующих задач.

**Перекрестная ссылка** В каждом случае следует решать, не лучше ли использовать класс, а не простой тип данных. Подробнее см. главу 6.

Будьте осторожны, создавая имена типов, ссылающиеся на предопределенные типы. Такие имена, как *BigInteger* или *LongString*, описывают компьютерные данные, а не конкретную задачу. Большое преимущество создания собственных типов данных состоит в том, что добавляется слой, изолирующий программу от языка разработки. Имена типов, ссылающиеся на типы языка, лежащие в их основе, нарушают эту изоляцию. Они не дают вам большого преимущества по сравнению с применением предопределенных типов. Проблемно-ориентированные имена, с другой стороны, облегчают процесс внесения изменений и предоставляют самодокументируемые объявления типов.

***Избегайте предопределенных типов*** Если есть хоть малейшая возможность, что тип может измениться, избегайте применения предопределенных типов везде, кроме определений *typedef* или *type*. Легко создать новые функционально-ориентированные типы — менять же данные в программе, использующей жестко закодированные типы, гораздо сложней. Более того, функционально-ориентированные типы частично документируют объявленные с ними переменные. Объявление *Coordinate x* сообщит вам об *x* гораздо больше, чем объявление *float x*. Используйте собственные типы везде, где только можно.

***Не переопределяйте предопределенные типы*** Изменение определения стандартного типа может вызвать путаницу. Например, если в вашем языке есть предопределенный тип *Integer*, не создавайте свой тип с именем *Integer*. Читающие ваш код могут забыть, что вы его переопределили, и будут считать, что видят тот же *Integer*, который привыкли видеть.

***Определите подстановки для переносимости*** В отличие от совета не изменять определение стандартных типов вы можете создать для этих типов подстановки, так что на разных платформах переменные будут представлены одними и теми же сущностями. Так, вы можете определить тип *INT32* и использовать его вместо *int* или тип *LONG64* вместо *long*. Изначально единственной разницей между двумя типами будет применение заглавных букв. Но при переходе на другую платформу вы сможете переопределить варианты с большими буквами так, чтобы они совпадали с типами для данных аппаратных средств.

Не создавайте типы, которые легко перепутать с предопределенными. Существует возможность определить *INT* вместо *INT32*, но лучше сделать явное различие между типами, созданными вами, и типами, предоставленными языком программирования.

***Рассмотрите вопрос создания класса вместо использования* typedef** Простые операторы *typedef* позволяют проделать большой путь в сторону сокрытия информации об исходном типе переменной. Однако иногда вам может потребоваться дополнительная гибкость и управляемость, которой позволяют добиться классы. Подробнее см. главу 6.

http://cc2e.com/1206

**Перекрестная ссылка** Список вопросов, затрагивающих данные вообще, без подразделения на конкретные типы, см. в контрольном списке главы 10. Список вопросов по вариантам именования см. в контрольном списке главы 11.

## Контрольный список: основные данные

**Числа в общем**

- ❑ Не содержит ли код магические числа?
- ❑ Предупреждаются ли в коде ошибки деления на ноль?
- ❑ Очевидны ли преобразования типов?
- ❑ Если переменные двух разных типов используются в одном выражении, будет ли оно вычислено так, как вы это предполагаете?
- ❑ Не происходит ли сравнение переменных разных типов?
- ❑ Компилируется ли программа без предупреждений компилятора?

**Целые числа**

- ❑ Работают ли выражения, содержащие целочисленное деление так, как это предполагалось?
- ❑ Предупреждаются ли в целочисленных выражениях проблемы целочисленного переполнения?

**Числа с плавающей запятой**

- ❑ Не содержит ли код операции сложения и вычитания слишком разных по величине чисел?
- ❑ Предупреждаются ли в коде ошибки округления?
- ❑ Не выполняется сравнение на равенство чисел с плавающей запятой?

**Символы и строки**

- ❑ Не содержит ли код магических символов и строк?
- ❑ Свободны ли операции со строками от ошибки потери единицы?
- ❑ Различаются ли в коде на C строковые указатели и массивы символов?
- ❑ Соблюдается ли в коде на C соглашение об объявлении строк с длиной *CONSTANT+1*?
- ❑ Используются ли в C массивы символов вместо указателей там, где это допустимо?
- ❑ Инициализируются ли в C строки с помощью *NULL* во избежание бесконечных строк?
- ❑ Используются ли в коде на C *strncpy()* вместо *strcpy()*? А *strncat()* и *strncmp()*?

**Логические переменные**

- ❑ Используются ли в программе дополнительные логические переменные для документирования проверок условия?
- ❑ Используются ли в программе дополнительные логические переменные для упрощения проверок условия?

**Перечислимые типы**

- ❑ Используются ли в программе перечислимые типы вместо именованных констант ради их улучшенной читабельности, надежности и модифицируемости?
- ❑ Используются ли перечислимые типы вместо логических переменных, если все значения переменной не могут быть переданы с помощью *true* и *false*?
- ❑ Проверяются ли некорректные значения перечислимых типов в условных операторах?
- ❑ Зарезервирован ли первый элемент перечислимого типа как недопустимый?

**Перечислимые константы**

- ❑ Используются ли в программе именованные константы вместо магических чисел для объявления данных и границ циклов?
- ❑ Используются ли именованные константы последовательно, чтобы одно значение не представлялось в одном месте константой, а в другом — литералом?

**Массивы**

- ❑ Находятся ли все индексы массива в его границах?
- ❑ Свободны ли ссылки на массив от ошибок потери единицы?
- ❑ Указаны ли все индексы многомерных массивов в правильном порядке?
- ❑ В правильном ли порядке используются переменные-индексы во вложенных циклах, не происходит ли пересечения индексов?

**Создание типов**

- ❑ Используются ли в программе отдельные типы для каждого вида данных, который может измениться?
- ❑ Ориентируются ли имена типов на реальные сущности, которые эти типы представляют, а не на типы языка программирования?
- ❑ Достаточно ли наглядны имена типов, чтобы помочь документированию объявлений данных?
- ❑ Не произошло ли переопределение предопределенных типов?
- ❑ Рассматривался ли вопрос создания нового класса вместо простого переопределения типа?

# Ключевые моменты

- Работа с определенными типами данных требует запоминания множества правил для каждого из них. Используйте список контрольных вопросов из этой главы, чтобы убедиться, что вы учли основные проблемы с ними.
- Создание собственных типов, если ваш язык это позволяет, упрощает модификацию вашей программы и делает ее более самодокументируемой.
- Прежде чем создавать простой тип с помощью *typedef* или его эквивалента, подумайте, не следует ли создать вместо него новый класс.

ГЛАВА 13

# Нестандартные типы данных

http://cc2e.com/1378

**Содержание**



**Связанные темы**

- Фундаментальные типы данных: глава 12
- Защитное программирование: глава 8
- Нестандартные управляющие структуры: глава 17
- Сложность в разработке ПО: раздел 5.2

Некоторые языки программирования поддерживают экзотические виды данных в дополнение к типам, обсуждавшимся в главе 12. В разделе 13.1 рассказывается, при каких обстоятельствах вы могли бы использовать структуры вместо классов. В разделе 13.2 описываются детали использования указателей. Если у вас возникали проблемы с использованием глобальных данных, из раздела 13.3 вы узнаете, как их избежать. Если вы думаете, что типы данных, описанные в этой главе, — это не те типы, о которых вы обычно читаете в современных книгах по объектно-ориентированному программированию, то вы абсолютно правы. Поэтому эта глава и называется «*Нестандартные* типы данных».

## 13.1. Структуры

Термин «структура» относится к типу данных, построенному на основе других типов. Так как массивы — особый случай, они рассматриваются отдельно в главе 12. В этом разделе обсуждаются структурированные данные, созданные пользователем: *structs* в C/C++ и *Structures* в Microsoft Visual Basic. В Java и C++ классы тоже иногда выглядят, как структуры (когда они состоят только из открытых членов данных и не содержат открытые методы).

Чаще всего вы предпочтете создавать классы, а не структуры, чтобы задействовать преимущества закрытости и функциональности, предлагаемой классами, в

дополнение к открытым данным, поддерживаемым структурами. Но иногда прямое манипулирование блоками данных может быть полезно.

***Используйте структуры для прояснения взаимоотношений между данными*** Структуры объединяют группы взаимосвязанных элементов. Иногда труднее всего понять, какие данные в программе используются вместе. Это похоже на прогулку по маленькому городку с вопросами о том, кто с кем в родстве. Вы выясняете, что каждый кому-то кем-то приходится, но вы не уверены в этом, и никогда не можете получить внятного ответа.

Если данные хорошо структурированы, выяснение, что с чем связано, сильно упрощается. Вот пример данных, которые не были структурированы:

**Пример неструктурированных, вводящих в заблуждение переменных (Visual Basic)**

```
name = inputName
address = inputAddress
phone = inputPhone
title = inputTitle
department = inputDepartment
bonus = inputBonus
```

Так как данные не структурированы, кажется, что эти операторы присваивания объединены. На самом деле, переменные *name*, *address*, и *phone* относятся к рядовому служащему, а *title*, *department* и *bonus* — к менеджеру. В этом фрагменте нет подсказок о том, что используется два вида данных. В следующем фрагменте применение структур делает взаимоотношения яснее:

**Пример более информативных, структурированных переменных (Visual Basic)**

```
employee.name = inputName
employee.address = inputAddress
employee.phone = inputPhone

supervisor.title = inputTitle
supervisor.department = inputDepartment
supervisor.bonus = inputBonus
```

В этом коде, содержащем структурированные переменные, очевидно, что часть данных относится к работнику, а остальные — к менеджеру.

***Используйте структуры для упрощения операций с блоками данных*** Вы можете объединить взаимосвязанные элементы в структуру и выполнять операции над ней. Проще обрабатывать структуру целиком, чем выполнять те же действия над каждым элементом. Это надежней и требует меньше строк кода.

Допустим, у вас есть группа взаимосвязанных элементов данных — скажем, информация о работнике в базе данных персонала. Если данные не объединены в структуру, простое копирование всей группы может потребовать большого числа операторов. Вот пример на Visual Basic:

**Пример громоздкого копирования группы (Visual Basic)**

```
newName = oldName
newAddress = oldAddress
newPhone = oldPhone
newSsn = oldSsn
newGender = oldGender
newSalary = oldSalary
```

Каждый раз, желая передать сведения о работнике, вам приходится иметь дело со всеми этими операторами. Если вам нужно добавить новый элемент данных, например *numWithholdings* , вам придется найти все места, где написаны эти присваивания, и добавить еще одно: *newNumWithholdings* = *oldNumWithholdings*.

Представьте, как ужасно будет менять местами данные о двух работниках. Вам не надо напрягать воображение — вот пример:

**Пример утомительного способа обмена двух групп данных (Visual Basic)**

```
‘ Поменять местами старые и новые данные о работнике.
previousOldName = oldName
previousOldAddress = oldAddress
previousOldPhone = oldPhone
previousOldSsn = oldSsn
previousOldGender = oldGender
previousOldSalary = oldSalary

oldName = newName
oldAddress = newAddress
oldPhone = newPhone
oldSsn = newSsn
oldGender = newGender
oldSalary = newSalary

newName = previousOldName
newAddress = previousOldAddress
newPhone = previousOldPhone
newSsn = previousOldSsn
newGender = previousOldGender
newSalary = previousOldSalary
```

Более легкое решение проблемы — объявить структурную переменную:

**Пример объявления структуры (Visual Basic)**

```
Structure Employee
   name As String
   address As String
   phone As String
   ssn As String
   gender As String
   salary As long
```

```
End Structure
Dim newEmployee As Employee
Dim oldEmployee As Employee
Dim previousOldEmployee As Employee
```

Теперь вы можете поменять элементы в старой и новой структуре с помощью трех операторов:

**Пример более легкого способа обмена двух групп данных (Visual Basic)**

```
previousOldEmployee = oldEmployee
oldEmployee = newEmployee
newEmployee = previousOldEmployee
```

Если вы хотите добавить новое поле, например *numWithholdings*, вы просто вносите его в объявление *Structure*. Ни одно из вышеприведенных выражений не потребует никаких изменений. C++ и другие языки также содержат подобные возможности.

***Используйте структуры для упрощения списка параметров*** Сократить список параметров метода позволяют структурированные переменные. Технология похожа на только что продемонстрированную. Вместо того чтобы передавать параметры по одному, можно объединить взаимосвязанные элементы в структуру и передать все скопом. Вот пример утомительного способа передачи группы общих параметров:

**Перекрестная ссылка** О том, как распределять данные между методами, см. подраздел «Поддерживайте сопряжение слабым» раздела 5.3.

**Пример громоздкого вызова метода, не использующего структуру (Visual Basic)**

```
HardWayRoutine( name, address, phone, ssn, gender, salary )
```

А вот пример простого способа вызвать метод с помощью структурированной переменной, состоящей из параметров первого метода:

**Пример элегантного вызова метода, использующего структуру (Visual Basic)**

```
EasyWayRoutine( employee )
```

Если вы хотите добавить *numWithholdings* к первому варианту программы, вам придется прочесать весь код и изменить каждый вызов *HardWayRoutine()*. Если же вы добавите элемент к структуре *Employee*, вам совсем не придется изменять параметры вызова *EasyWayRoutine()*.

Вы можете довести эту идею до крайности, поместив все переменные вашей программы в одну большую, жирную структуру и передавая ее всюду. Аккуратные программисты избегают объединения большего количества данных, чем это необходимо логически. Более того, аккуратисты стараются не передавать параметры в виде структур, если нужны лишь одно-два поля из этой структуры — в этом случае передаются указанные поля. Это аспект вопроса о сокрытии информации: часть данных скрыта *в* методе, а часть — *от* метода. Между методами должна передаваться только та информация, которую необходимо знать.

**Перекрестная ссылка** Об опасностях передачи слишком большого объема данных см. подраздел «Поддерживайте сопряжение слабым» раздела 5.3.

***Используйте структуры для упрощения сопровождения*** Так как, применяя структуры, вы группируете взаимосвязанные данные, изменение структуры требует минимальных исправлений в программе. В большей степени это относится к участкам кода, не связанным с вносимым изменением логически. Поскольку изменения часто приводят к ошибкам, то чем меньше изменений, тем меньше ошибок. Если в структуре *Employee* есть поле *title* и вы решаете его удалить, вам не нужно исправлять ни списки параметров, ни операторы присваивания, использующие эту структуру целиком. Конечно, вам придется поправить код, напрямую работающий со званиями работников, но эти действия связаны с процессом удаления поля *title* концептуально, и поэтому вы вряд ли о них забудете.

На участках кода, логически не связанных с полем *title*, преимущество структурирования данных еще более очевидно. Иногда программы содержат выражения, концептуально работающие скорее с набором данных, а не с отдельными компонентами. В этих случаях отдельные элементы, такие как поле *title*, упоминаются только потому, что они — часть набора. Эти участки кода не имеют логических причин работать конкретно с полем *title*, поэтому при изменении *title* такие участки очень легко пропустить. Если же вы используете структуру, все будет нормально, потому что код ссылается на набор взаимосвязанных данных, а не на каждый элемент индивидуально.

# 13.2. Указатели

Использование указателей — одна из наиболее подверженных ошибкам областей программирования. Это привело к тому, что современные языки, такие как Java, C# и Visual Basic, не предоставляют указатель в качестве типа данных. Применять указатели и так сложно, а правильное применение требует от вас отличного понимания того, как ваш компилятор управляет распределением памяти. Многие общие проблемы с безопасностью, особенно случаи переполнения буфера, могут быть сведены к ошибочному использованию указателей. (Howard and LeBlanc, 2003).

Даже если в вашем языке не нужны указатели, их хорошее понимание поможет вам разобраться, как работает ваш язык программирования. А щедрая доза защитного программирования будет еще полезнее.

## Парадигма для понимания указателей

Концептуально каждый указатель состоит из двух частей: области памяти и знания, как следует интерпретировать содержимое этой области.

### Область памяти

Область памяти — это адрес, часто представленный в шестнадцатеричном виде. В 32-разрядном процессоре адрес будет 32-битным числом, например *0x0001EA40*. Сам по себе указатель содержит только этот адрес. Чтобы обратиться к данным, на которые этот указатель указывает, надо пойти по этому адресу и как-то интерпретировать содержимое памяти в этой области. Сам по себе этот участок памяти — просто набор битов. Чтобы он обрел смысл, его надо как-то истолковать.

### Знание, как интерпретировать содержимое

Информация о том, как интерпретировать содержимое области памяти, предоставляется основным типом указателя. Если указатель указывает на целое число, то на самом деле это значит, что компилятор интерпретирует область памяти, задаваемую указателем, как целое число. Конечно, у вас может быть указатель на целое число, строку или число с плавающей точкой, которые ссылаются на одну и ту же область памяти. Но только один из них будет корректно интерпретировать содержимое этой области.

Говоря об указателях, полезно помнить, что память сама по себе не имеет однозначной интерпретации. И только с помощью конкретного типа указателя набор битов в некоторой области памяти истолковывается как осмысленное значение.

Рис. 13-1. содержит несколько представлений одной и той же области памяти, интерпретированной разными способами.

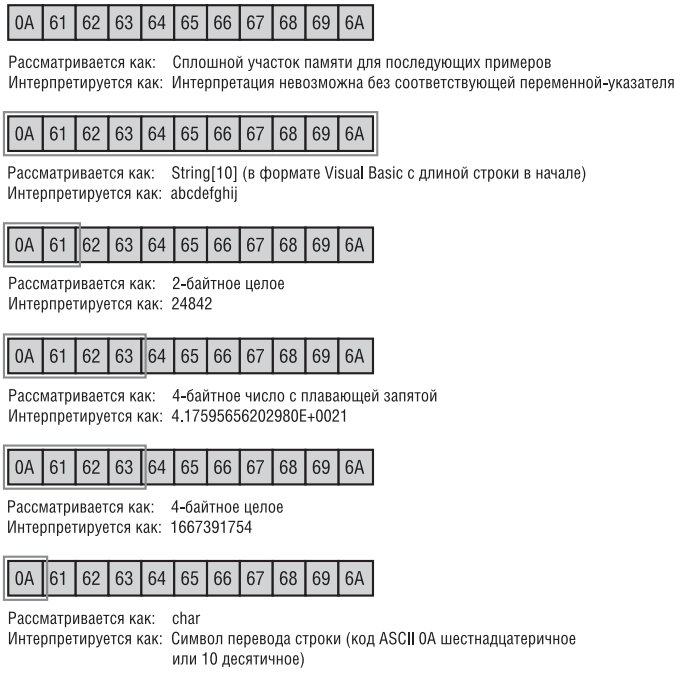



***Рис. 13-1.*** *Объем памяти, используемый каждым типом данных, показан двойной линией*

В каждом случае на рис. 13-1 указатель ссылается на область памяти, содержащую шестнадцатеричное значение *0x0A*. Количество байт, используемых после *0A*, зависит от того, как память интерпретируется. Содержимое памяти также зависит от ее интерпретации. (Еще содержимое зависит и от используемого процессора, так что не забудьте об этом, если будете пытаться повторить эти результаты на вашем Cray-десктопе.) Одна и та же необработанная область памяти может быть представлена как строка, целое число, число с плавающей точкой или иначе — все зависит от основного типа указателя, ссылающегося на эту область памяти.

## Основные советы по использованию указателей

Обычно ошибку легко найти, но трудно исправить. Но это не относится к проблемам с указателями. Ошибка в указателе — чаще всего результат того, что он указывает не туда, куда должен. Когда вы присваиваете значение некорректной переменной-указателю, вы записываете данные туда, куда не должны записывать. Это называется «повреждением памяти». Порой оно приводит к ужасным крушениям системы, порой изменяет результаты вычислений в другой части программы, порой вынуждает программу неожиданно завершить работу метода, а порой вообще ничего не делает. В последнем случае ошибка в указателе как бомба с часовым механизмом — она разрушит вашу программу за пять минут до ее показа самому главному заказчику. По симптомам таких ошибок тяжело понять, что их вызывало. Поэтому самым трудоемким в процессе исправления ошибок указателя является поиск их причины.

Для успешной работы с указателями требуется двухэтапная стратегия. Во-первых, старайтесь изначально не делать в них ошибок. Проблемы с указателями так сложно обнаружить, что дополнительные превентивные меры вполне оправданны. Во-вторых, выявляйте ошибки в указателях как можно быстрее после того, как они закодированы. Симптомы ошибок в указателях настолько изменчивы, что дополнительные меры с целью сделать эти симптомы более предсказуемыми, также вполне оправданны. Вот как можно добиться этих ключевых целей.

***Изолируйте операции с указателями в методах или классах*** Допустим, в нескольких частях программы используется связный список. Вместо того чтобы каждый раз обрабатывать его вручную, напишите методы доступа *NextLink()*, *PreviousLink()*, *InsertLink()* и *DeleteLink()*. Минимизировав количество мест, в которых выполняется обращение к указателю, вы уменьшите вероятность неосторожных ошибок, распространяющихся по всей программе, на поиск которых уходит вечность. Поскольку такой код становится относительно независимым от деталей представления данных, вы также увеличиваете шансы его повторного использования в других программах. Написание методов, распределяющих память для указателей, — еще один способ централизовать управление вашими данными.

***Выполняйте объявление и определение указателей одновременно*** Присвоение переменной начального значения рядом с местом ее объявления — как правило, хорошая практика программирования. Она обладает особой ценностью при работе с указателями. Вот как не надо делать:

**Пример неправильной инициализации указателя (C++)**

```
Employee  *employeePtr;
// много кода
...
employeePtr = new Employee;
```

Даже если этот код изначально работает правильно, при дальнейших модификациях он подвержен ошибкам, так как существует шанс, что кто-нибудь попробует

использовать *employeePtr* после его объявления, но до инициализации. Вот более безопасный подход:

**Пример правильной инициализации указателя (C++)**

```
// много кода
...
Employee *employeePtr = new Employee;
```

***Удаляйте указатели в той же области действия, где они были созданы*** Соблюдайте симметрию при выделении и освобождении памяти для указателей. Если вы используете указатель в единственном блоке кода, вызывайте *new* для выделения памяти и *delete* для ее освобождения в том же блоке. Если вы распределяете память внутри метода, освобождайте ее внутри аналогичного метода, а если в конструкторе объекта — освобождайте в деструкторе этого объекта. Метод, выделяющий память для указателя, а затем ожидающий, что клиентский код вручную его освободит, нарушает целостность, что прямиком ведет к ошибкам.

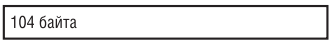
***Проверяйте указатели перед их применением*** Прежде чем использовать указатель в критической части вашей программы, удостоверьтесь, что он указывает на осмысленную область памяти. Так, если вы ожидаете, что память распределяется между адресами *StartData* и *EndData*, у вас должно вызывать подозрение, если значение указателя меньше, чем *StartData*, или больше, чем *EndData*. Вам надо определить значения *StartData* и *EndData* в вашей системе. Эту проверку можно выполнять автоматически, если обращаться к указателям не напрямую, а через методы доступа.

***Проверяйте переменную, на которую ссылается указатель, перед ее использованием*** Иногда вы можете выполнить корректную проверку значения, на которое ссылается указатель. Скажем, если вы предполагаете, что он указывает на целое число от 0 до 1000, значения больше 1000 должны вызывать у вас подозрение. Если указатель ссылается на строку в стиле C++, ее длина свыше 100 символов также может вызывать недоверие. Эти проверки тоже могут быть выполнены автоматически при работе с указателями с помощью методов доступа.

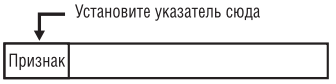
***Используйте закрепленные признаки для проверки повреждения памяти*** «Поле-тэг» (tag field) или «закрепленный признак» (dog tag) — это поле, которое вы добавляете к структуре исключительно с целью проверки ошибок. Когда вы выделяете память для переменной, поместите в это поле закрепленного признака значение, которое должно остаться неизменным. Используя структуру, особенно освобождая для нее память, проверяйте значение закрепленного признака. Если это поле не содержит ожидаемого значения, значит, данные были повреждены.

Удаляя указатель, измените значение этого поля. Так вы сможете выявить ошибку, если случайно попытаетесь освободить этот указатель еще раз. Например, пусть нам нужно выделить 100 байт памяти:

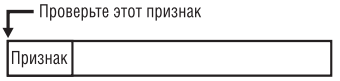
1. Выделите 104 байта — на 4 байта больше, чем требуется.



2. Укажите в первых четырех байтах значение обязательного признака, а затем верните указатель на область, следующую за этими четырьмя байтами.




3. Когда понадобится удалить указатель, проверьте значение признака.




4. Если значение признака корректно, присвойте ему *0* или другое значение, которое ваша программа будет считать недопустимым. Главное, чтобы его ошибочно не посчитали корректным после освобождения памяти. С той же целью заполните всю область памяти *0*, *0xCC* или любым другим неслучайным значением.
5. В заключение удалите указатель.


Размещение закрепленного признака в начале выделенного блока памяти поможет выявить попытки повторного освобождения блока. При этом вам не нужно поддерживать список используемых областей памяти. Размещение признака в конце блока позволит выявить попытки записи за допустимые границы области памяти. Для достижения обеих целей закрепленные признаки можно размещать и в начале, и в конце блока.

Вы можете использовать этот подход и для дополнительных проверок, предложенных ранее, — того, что значение указателя должно быть между адресами *StartData* и *EndData*. Однако, чтобы убедиться, что указатель содержит корректный адрес, вместо проверки возможного диапазона адресов следует проверять наличие этого указателя в списке используемых областей памяти.

Если вы проверите поле признака один раз — перед удалением переменной, то некорректный признак будет означать, что когда-то на протяжении жизни переменной ее содержимое было повреждено. Но чем чаще вы будете проверять этот признак, тем ближе к источнику проблемы будет обнаружено повреждение.

***Добавьте явную избыточность*** Альтернативой полю признака будет использование двух таких полей. Если данные в избыточных полях не совпадают, вы знаете, что память была повреждена. Этот способ может потребовать большого количества дополнительного кода, если напрямую манипулировать указателями. Но если работу с указателями изолировать в методах, то дублировать код придется лишь в нескольких местах.

***Используйте для ясности дополнительные переменные указателей*** Никогда не экономьте на переменных-указателях. Одну и ту же переменную нельзя вызвать для разных целей. Особенно это касается переменных-указателей. Довольно тяжело выяснить, какие действия выполняются со связным списком и без того, чтобы разбираться, почему одна переменная *genericLink* используется снова и снова и куда указывает *pointer->next->last->next*. Рассмотрим фрагмент:

**Пример кода традиционной вставки в список нового узла (C++)**

```
void InsertLink(
   Node *currentNode,
   Node *insertNode
   ) {
   // добавляем "insertNode" после "currentNode"
   insertNode->next = currentNode->next;
   insertNode->previous = currentNode;
   if ( currentNode->next != NULL ) {
```

Эта строка излишне сложна.

```
      currentNode->next->previous = insertNode;
   }
   currentNode->next = insertNode;
}
```

Этот традиционный код добавления нового узла в связный список излишне сложен для понимания. В добавлении элемента задействованы три объекта: текущий узел, узел, в данный момент следующий за текущим, и узел, который надо вставить между ними. Однако в коде явно упомянуты только два объекта: *insertNode* и *currentNode*. Из-за этого вам придется запомнить, что *currentNode->next* тоже участвует в алгоритме. Если вы попробуете изобразить диаграммой, что происходит, не используя элемент, изначально следующий за *currentNode*, у вас получится что-то вроде этого:


Гораздо лучшая диаграмма содержит все три объекта. Она может выглядеть так:


Вот пример кода, который явно упоминает все три объекта, участвующих в алгоритме:

**Пример более читабельного кода для вставки узла (C++)**

```
void InsertLink(
   Node *startNode,
   Node *newMiddleNode
   ) {
   // вставляем "newMiddleNode" между "startNode" и "followingNode"
   Node *followingNode = startNode->next;
   newMiddleNode->next = followingNode;
   newMiddleNode->previous = startNode;
   if ( followingNode != NULL ) {
      followingNode->previous = newMiddleNode;
   }
   startNode->next = newMiddleNode;
}
```

Этот код содержит одну дополнительную строку, но без участия выражения *currentNode->next->previous* из первого фрагмента этот пример легче для понимания.

***Упрощайте сложные выражения с указателями*** Сложные выражения с использованием указателей тяжело читать. Если в вашем коде есть выражения вроде *p->q->r->s.data*, подумайте о том человеке, которому придется это читать. Вот особенно вопиющий пример:

**Пример сложного для понимания выражения с указателем (C++)**

```
for ( rateIndex = 0; rateIndex < numRates; rateIndex++ ) {
   netRate[ rateIndex ] = baseRate[ rateIndex ] * rates->discounts->factors->net;
}
```

Подобные выражения заставляют разбираться в коде, а не читать его. Если в вашей программе есть сложное выражение, присвойте его понятно названной переменной, чтобы прояснить смысл операции. Вот улучшенная версия примера:

**Пример упрощения сложного выражения с указателем (C++)**

```
quantityDiscount = rates->discounts->factors->net;
for ( rateIndex = 0; rateIndex < numRates; rateIndex++ ) {
   netRate[ rateIndex ] = baseRate[ rateIndex ] * quantityDiscount;
}
```

Это упрощение не только позволяет увеличить удобочитаемость, но, возможно, и повысить производительность, упростив операцию с указателем внутри цикла. Но, как обычно, улучшение производительности надо измерить до того, как делать на это крупные ставки.

**Перекрестная ссылка** Такие диаграммы, как на рис. 13-2, могут стать частью внешней документации вашей программы. О хорошей практике документирования см. главу 32.

***Нарисуйте картинку*** Описание указателей в коде программы может сбивать с толку. Обычно помогает картинка. Например, изображение задачи по вставке элемента в связный список может выглядеть так (рис. 13-2):

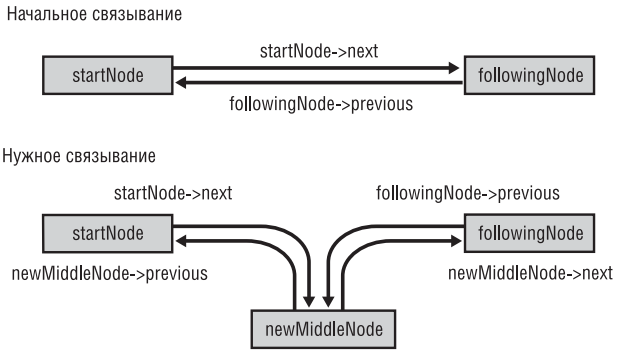



***Рис. 13-2.*** *Пример рисунка, помогающего осмыслить шаги, необходимые для изменения связей между элементами*

***Удаляйте указатели в связных списках в правильном порядке*** Обычной проблемой в работе с динамически созданными связными списками является освобождение сначала первого указателя, после чего становится невозможно

получить указатель на следующий узел списка. Чтобы избежать этой проблемы, перед удалением текущего элемента убедитесь, что у вас есть указатель на следующий элемент списка.

***Выделите «запасной парашют» памяти*** Если в программе используется динамическая память, необходимо избежать проблемы ее внезапной нехватки, приводящей к исчезновению пользовательских данных на бескрайних просторах оперативной памяти. Один из способов дать вашей программе запас прочности — заранее выделить «парашют» памяти. Определите, какой объем памяти нужен программе для сохранения работы, освобождения ресурсов и аккуратного завершения. Зарезервируйте эту память в начале работы программы как запасной парашют и оставьте ее в покое. Когда памяти станет не хватать, раскройте резервный парашют — освободите эту память и завершите работу программы.

***Уничтожайте мусор*** Ошибки указателей сложно отслеживать, потому что момент времени, когда память, адресуемая указателем, станет недействительной, не определен. Иногда содержимое памяти после освобождения указателя долго еще выглядит корректным. В другой раз ее содержимое изменится сразу.

**Дополнительные сведения** Отличное обсуждение безопасных подходов к обработке указателей в языке C см. в книге «Writing Solid Code» (Maguire, 1993).

Вы можете избежать ошибок с освобожденными указателями, записывая мусор в блоки памяти прямо перед их освобождением. Если вы используете методы доступа, то это, как и многие другие операции, можно делать автоматически. В C++ при каждом удалении указателя можно делать так:

**Пример принудительной записи мусорных данных в освобождаемую память (C++)**

```
memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) );
delete pointer;
```

Естественно, эта технология требует поддержки списка размеров памяти, выделенной для указателей, которые можно было бы получить функцией *MemoryBlockSize()*. Мы обсудим это позднее.

***Устанавливайте указатели* null *при их удалении или освобождении*** Известный тип ошибок указателей — это «висячий указатель» (dangling pointer), т. е. обращение к нему после вызова функций *delete* или *free*. Одна из причин, по которым ошибки в указателях так сложно обнаружить, в том, что иногда симптомы ошибки никак не проявляются. Записывая в указатели пустое значение после их освобождения, вы не измените факт чтения данных, адресуемых висячим указателем. Но вы добьетесь того, что запись данных по этому адресу приведет к ошибке. Возможно, это будет ужасная, катастрофическая ошибка, но по крайней мере ее обнаружите вы, а не кто-то другой.

Код, предшествующий операции *delete* в предыдущем примере, можно дополнить, чтобы обрабатывать и эту ситуацию:

**Пример установки указателя в *null* после его удаления (C++)**

```
memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) );
delete pointer;
pointer = NULL;
```

***Проверяйте корректность указателя перед его удалением*** Один из лучших способов обрушить программу — вызвать функции *delete()* или *free()* для указателя, который уже был освобожден. Увы, лишь немногие языки обнаруживают такой тип ошибок.

Если вы присваиваете освобождаемым указателям пустое значение, то перед использованием или повторным удалением указателя вы сможете проверить его на равенство *null*. Разумеется, если вы не устанавливаете в *null* освобождаемые указатели, у вас такой возможности не будет. В связи с этим можно предложить следующее дополнение к коду удаления указателя:

**Пример проверки утверждения о неравенстве указателя *null* перед его удалением (C++)**

```
ASSERT( pointer != NULL, "Attempting to delete null pointer." );
memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) );
delete pointer;
pointer = NULL;
```

***Отслеживайте распределение памяти для указателей*** Ведите список указателей, для которых была выделена память. Это позволит вам проверить, находится ли указатель в этом списке перед его освобождением. Вот как для этих целей может быть изменен код удаления указателя:

**Пример проверки, выделялась ли память для указателя (C++)**

```
ASSERT( pointer != NULL, "Attempting to delete null pointer." );
if ( IsPointerInList( pointer ) ) {
   memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) );
   RemovePointerFromList( pointer );
   delete pointer;
   pointer = NULL;
}
else {
   ASSERT( FALSE, "Attempting to delete unallocated pointer." );
}
```

***Напишите методы-оболочки, чтобы централизовать стратегию борьбы с ошибками в указателях*** Как видно из этого примера, каждый вызов операторов *new* и *delete* может сопровождаться достаточно большим количеством дополнительного кода. Некоторые технологии, описанные в этом разделе, являются взаимоисключающими или избыточными, и не хотелось бы использовать несколько конфликтующих стратегий в одной программе. Например, вам не надо создавать и проверять обязательные признаки, если вы поддерживаете собственный список действительных указателей.

Вы можете минимизировать избыточность в программе и уменьшить вероятность ошибок, написав методы-оболочки для общих операций с указателями. В C++ вы могли бы использовать следующие методы:

- ***SAFE_NEW*** Вызывает *new* для выделения памяти, добавляет указатель в список задействованных указателей и возвращает вновь созданный указатель вы-

зывающей стороне. Он может также проверить, что оператор new не вернул *null* (ошибка нехватки памяти). Поскольку это надо сделать только единожды в этом месте, упрощается процесс обработки ошибок в других частях вашей программы.

- ***SAFE_DELETE*** Проверяет, находится ли переданный ему указатель в списке действительных указателей. Если он там есть, метод записывает мусор в адресуемую им память, удаляет указатель из списка, вызывает C++-оператор *delete* для освобождения памяти и устанавливает указатель в *null*. Если указатель не найден в списке, *SAFE_DELETE* выводит диагностическое сообщение и прерывает программу.

Метод *SAFE_DELETE*, реализованный в виде макроса, может выглядеть так:

**Пример добавления оболочки для кода удаления указателя (C++)**

```
#define SAFE_DELETE( pointer ) { \
   ASSERT( pointer != NULL, "Attempting to delete null pointer."); \
   if ( IsPointerInList( pointer ) ) { \
      memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) ); \
      RemovePointerFromList( pointer ); \
      delete pointer; \
      pointer = NULL; \
   } \
   else { \
      ASSERT( FALSE, "Attempting to delete unallocated pointer." ); \
   } \
}
```

В C++ этот метод будет освобождать единичные указатели, поэтому вам придется создать похожий макрос *SAFE_DELETE_ARRAY* для удаления массивов.

**Перекрестная ссылка** О планах по удалению отладочного кода см. подраздел «Запланируйте удаление отладочных средств» раздела 8.6.

Централизовав управление памятью в этих двух методах, вы также сможете менять поведение *SAFE_NEW* и *SAFE_DELETE* в отладочной и промышленных версиях продукта. Например, обнаружив попытку освободить пустой указатель в период разработки, *SAFE_DELETE* может остановить программу. Но если это происходит во время эксплуатации, он может просто записать ошибку в журнал и продолжить выполнение.

Вы легко сможете адаптировать эту схему для функций *calloc* и *free* в языке C, а также для других языков, использующих указатели.

***Используйте технологию, не основанную на указателях*** Указатели в целом сложнее для понимания, они подвержены ошибкам и приводят к созданию машинно-зависимого, непереносимого кода. Если вы можете придумать разумную альтернативу указателям, избавьте себя от головной боли и возьмите ее за основу.

## Указатели в C++

Язык C++ добавил специфические тонкости при работе с указателями и ссылками. В следующих подразделах описаны основные принципы, применяемые в работе с указателями на C++.

**Дополнительные сведения** Множество других советов по применению указателей в C++ см. в «Effective C++», 2d ed. (Meyers, 1998) и «More Effective C++» (Meyers, 1996).

***Осознайте различие между указателями и ссылками*** В C++ и указатели (*), и ссылки (&) косвенно ссылаются на объект. Для непосвященных единственной, чисто косметической разницей между ними будет способ обращения к полю: *object->field* или *object.field*. Наиболее значительным различием является то, что ссылка обязана всегда ссылаться на объект, тогда как указатель может быть равен *null*. Кроме того, после инициализации ссылки нельзя изменить то, куда она ссылается.

***Используйте указатели для передачи параметров «по ссылке» и константные ссылки для передачи параметров «по значению»*** По умолчанию C++ передает в методы аргументы по значению, а не по ссылке. Когда объект передается по значению, C++ создает копию объекта, и при передаче объекта вызывающей программе вновь создается копия. Для больших объектов такое копирование может съедать много времени и ресурсов. Следовательно, при передаче объектов в метод вы обычно стараетесь избегать копирования объектов, а это означает, что вы хотите передавать их по ссылке, а не по значению.

Однако иногда хотелось бы использовать *семантику* передачи по значению (т. е. передаваемый объект должен остаться неизменным) и *реализацию* передачи параметра по ссылке (т. е. передавать сам объект, а не его копию).

В C++ решением этой проблемы является применение указателей для передачи по ссылке, и — как ни странно может звучать — «*константных* ссылок» для передачи по значению! Приведем пример:

**Пример передачи параметров по значению и по ссылке (C++)**

```
void SomeRoutine(
    const LARGE_OBJECT &nonmodifiableObject,
    LARGE_OBJECT *modifiableObject
);
```

Дополнительным преимуществом этого подхода является синтаксическое различие между изменяемыми и неизменяемыми объектами в вызванном методе. В изменяемых объектах ссылка на элементы будет осуществляться с помощью нотации *object->member*, тогда как в неизменяемых будет использоваться нотация *object.member*.

Недостаток этого подхода состоит в необходимости постоянного применения константных ссылок. Хорошим тоном считается использование модификатора *const* везде, где это возможно (Meyers, 1998). Поэтому в своем коде вы сможете объявлять передаваемые по значению параметры как константные ссылки. В библиотечном коде и других неподконтрольных вам местах вы столкнетесь с проблемой константных параметров. Компромиссной позицией будет все же задавать параметры, предназначенные только для чтения, с помощью ссылок, но не объявлять их константными. При этом подходе вы не в полной мере реализуете преимущество проверки компилятором попыток модификации неизменяемых аргументов метода, однако по крайней мере предоставляете возможность визуального различения *object->member* и *object.member*.

***Используйте автоматические указатели* auto_ptr** Если вы еще не выработали привычку использовать указатели *auto_ptr*, займитесь этим! Удаляя занятую память автоматически при выходе *auto_ptr* из области видимости, такие указатели решают множество проблем с утечками памяти, присущих обычным указателям. Книга «More Effective C++» Скотта Мейерса в правиле 9 содержит интересное обсуждение *auto_ptr* (Meyers, 1996).

***Изучите интеллектуальные указатели*** Интеллектуальные указатели — это замена обычных или «тупых» указателей (Meyers, 1996). Они действуют аналогично обычным, но предоставляют дополнительные возможности по управлению ресурсами, операциям копирования, присваивания, создания и удаления объектов. Перечисленные действия характерны для C++. Более полное обсуждение см. в правиле 28 книги «More Effective C++».

## Указатели в C

Вот несколько советов по применению указателей, которые в особенности имеют отношение к языку C.

***Используйте явный тип указателя вместо типа по умолчанию*** Язык C позволяет использовать указатели на *char* или *void* для любого типа переменной. Главное, что указатель куда-то указывает, и языку, в общем, не важно, на что именно. Но если вы используете явные типы для указателей, компилятор может выдавать предупреждение о несовпадающих типах указателей и некорректных преобразованиях. Если же явные типы не используются, он этого сделать не сможет. Старайтесь применять конкретные типы где только можно.

Из этого правила следует необходимость явного преобразования типа в тех случаях, когда нужно его изменить. Так, в этом фрагменте очевидно, что выделяется память для переменной типа *NODE_ PTR*:

#### Пример явного преобразования типа (C)

```
NodePtr = (NODE_PTR) calloc( 1, sizeof( NODE ) );
```

***Избегайте преобразования типов*** Этот совет о преобразовании типов не имеет ничего общего с учебой в актерской школе или отказом всегда играть негодяев. Он предлагает избегать втискивания переменной одного типа в переменную другого типа. Такое преобразование выключает способность вашего компилятора проверять несовпадения типов и тем самым пробивает брешь в броне защитного программирования. В программе, требующей многочисленных преобразований типов, вероятно, существуют какие-то архитектурные нестыковки, которые нужно пересмотреть. Попробуйте перепроектировать систему, в противном случае старайтесь избегать преобразований типов, насколько это возможно.

***Следуйте правилу звездочки при передаче параметров*** Вы можете получить значение аргумента из функции на языке C, только если в операции присваивания перед этим аргументом была указана звездочка (*). Многие программисты испытывают трудности при определении, когда C позволяет передавать значение обратно в вызывающий метод. Легко запомнить, что если вы указываете звездочку перед параметром, которому присваиваете значение, то это значение бу-

дет возвращено в вызывающий метод. Независимо от того, сколько звездочек вы указали в объявлении, для передачи значения в операторе присваивания должна быть хотя бы одна. Так, в следующем фрагменте значение, присвоенное переменной *parameter*, не будет передано в вызывающий метод, потому что операция присваивания не содержит звездочки:

**Пример передачи параметра, который не будет работать (C)**

```
void TryToPassBackAValue( int *parameter ) {
   parameter = SOME_VALUE;
}
```

А здесь значение, присвоенное параметру *parameter*, будет возвращено, потому что перед *parameter* указана звездочка:

**Пример передачи параметра, который сработает (C)**

```
void TryToPassBackAValue( int *parameter ) {
   *parameter = SOME_VALUE;
}
```

***Используйте* sizeof() *для определения объема памяти, необходимой для размещения переменной*** Легче использовать *sizeof()*, чем выяснять размер типа в справочнике. Кроме того, *sizeof()* работает с вашими собственными структурами, которые в справочнике не описаны. Так как значение вычисляется в момент компиляции, то *sizeof()* не влияет на производительность. Кроме того, он переносим: перекомпиляция в другой среде автоматически изменяет размеры, вычисленные *sizeof()*. И еще он прост в сопровождении, поскольку при изменении используемого типа изменится и рассчитываемый размер.

# 13.3. Глобальные данные

**Перекрестная ссылка** О различиях между глобальными данными и данными класса, см. подраздел «Ошибочное представление о данных класса как о глобальных данных» раздела 5.3.

Глобальные переменные доступны из любого места программы. Иногда этот термин небрежно используют для обозначения переменных с более широкой областью видимости, чем локальные переменные, таких как классовые переменные, доступные во всех методах класса. Но сама по себе доступность внутри единственного класса не означает, что переменная является глобальной.

Наиболее опытные программисты пришли к выводу, что применять глобальные переменные рискованней, чем локальные. Эти программисты также считают, что полезней осуществлять доступ к данным с помощью методов.

Даже если применение глобальных переменных не всегда ведет к ошибкам, оно все-таки вряд ли представляет собой хороший способ программирования.

## Распространенные проблемы с глобальными данными

Если вы без разбора используете глобальные переменные или считаете невозможность их применения ненужным ограничением, то, вероятно, вы еще не прониклись значимостью принципов модульности и сокрытия информации. Модульность, сокрытие информации и связанное с ними использование хорошо спроектированных классов, может, и не панацея, но они помогают сделать большие программы понятнее и легче в сопровождении. Когда вы это поймете, вам захочется писать методы и классы как можно меньше взаимодействующие с глобальными переменными и внешним миром.

Можно привести массу проблем, связанных с глобальными данными, но в основном они сводятся к следующим вариантам.

***Непреднамеренные изменения глобальных данных*** Вы можете изменить значение глобальной переменной в одном месте и ошибочно думать, что оно осталось прежним где-то в другом. Такая проблема известна как «побочный эффект». Например, в этом фрагменте *theAnswer* является глобальной переменной:

**Пример побочного эффекта (Visual Basic)**

```
theAnswer — глобальная переменная.
theAnswer = GetTheAnswer()
GetOtherAnswer() изменяет theAnswer.
otherAnswer = GetOtherAnswer()
Значение averageAnswer неправильно.
averageAnswer = (theAnswer + otherAnswer) / 2
```

Вы предполагаете, что вызов *GetOtherAnswer()* не изменяет значение *theAnswer*, потому что иначе среднее значение в третьей строке будет вычислено неверно. На самом деле *GetOtherAnswer()* все-таки изменяет *theAnswer*, и в программе возникает ошибка.

***Причудливые и захватывающие проблемы при использовании псевдонимов для глобальных данных*** Использование псевдонима означает обращение к переменной по двум и более именам. Это происходит, когда глобальная переменная передается в метод, а там используется и в качестве глобальной переменной, и в качестве параметра. Вот пример метода, работающего с глобальной переменной:

**Пример метода, подверженного проблеме с псевдонимами (Visual Basic)**

```
Sub WriteGlobal( ByRef inputVar As Integer )
   inputVar = 0
   globalVar = inputVar + 5
   MsgBox( "Input Variable:  " & Str( inputVar ) )
   MsgBox( "Global Variable: " & Str( globalVar ) )
End Sub
```

А вот код вызывающего метода с глобальной переменной в качестве аргумента:

**Пример вызова метода с аргументом, демонстрирующим проблему псевдонимов (Visual Basic)**

```
WriteGlobal( globalVar )
```

Поскольку *inputVar* инициализируется *0*, и *WriteGlobal()* добавляет *5* к *inputVar*, чтобы получить новое значение *globalVar*, вы ожидаете, что *globalVar* будет на 5 больше, чем *inputVar*. Но вот неожиданный результат:

**Результат проблемы с псевдонимами**

```
Input Variable:  5
Global Variable: 5
```

Хитрость в том, что *globalVar* и *inputVar* — на самом деле одна и та же переменная! Поскольку *globalVar* передается в *WriteGlobal()* вызывающим методом, к ней обращаются с помощью двух разных имен. Поэтому результат вызовов *MsgBox()* отличается от ожидаемого: они показывают одну и ту же переменную дважды, хотя и используют два разных имени.

***Проблемы реентерабельности глобальных данных*** Сейчас все чаще встречается код, который может выполняться одновременно несколькими потоками. Многопоточное программирование создает вероятность обращения к глобальным данным будут обращаться не только из разных методов, но и из разных экземпляров одной и той же программы. В такой среде вы должны быть уверены, что глобальные данные сохранят свои значения, даже если будет запущено несколько копий программы. Это важная проблема, и вы сможете ее избежать, используя технологии, предложенные ниже.

***Затруднение повторного использования кода, вызванное глобальными данными*** Для использования кода из одной программы в другой вам нужно вытащить его из первой программы и внедрить во вторую. В идеале вы могли бы извлечь отдельный метод или класс, встроить в другую программу и наслаждаться жизнью.

Глобальные данные усложняют картину. Если класс, который вы хотите использовать повторно, читает или записывает глобальные данные, вы не сможете просто перенести его в новую программу. Вам придется изменить либо новую программу, либо старый класс, чтобы они стали совместимы. Правильным решением будет модификация старого класса, чтобы он не использовал глобальные данные: сделав это, вы сможете в следующий раз повторно использовать этот класс без дополнительных усилий. Неправильным решением будет модификация новой программы с целью создания таких же глобальных данных, какие требуются старому классу. Это как вирус — глобальные данные не только влияют на исходный код, но и распространяются по новым программам, использующим какие-либо классы из старой.

***Проблемы с неопределенным порядком инициализации глобальных данных*** Порядок, в котором данные из разных «единиц трансляции» (файлов) будут инициализироваться, в некоторых языках программирования (в частности, C++) не определен. Если при инициализации глобальной переменной из одного файла

используется глобальная переменная из другого файла, значение второй переменной предсказать сложно, если только вы не предпримете специальные действия для их инициализации в правильном порядке.

Эта проблема решается с помощью обходного маневра, описанного в правиле 47 книги Скотта Мейерса «Effective C++» (Meyers, 1998). Но изощренность решения как раз и иллюстрирует ту излишнюю сложность, которую привносят глобальные данные.

***Нарушение модульности и интеллектуальной управляемости, привносимое глобальными данными*** Сущность создания программ, состоящих из более чем нескольких сотен строк кода, заключается в управлении сложностью. Единственный способ, позволяющий интеллектуально управлять большой программой, — это разбить ее на части так, чтобы в каждый момент времени думать только об одной из них. Модульность — наиболее мощный инструмент для разбиения программы на части.

Глобальные данные проделывают дыры в возможности модуляризации. Если вы используете глобальные данные, разве вы можете сосредоточиться только на одном методе? Нет. Вам приходится сосредоточиваться на этом методе и на всех других, в которых используются те же глобальные данные. Хотя эти данные и не разрушают модульность программы полностью, они ее ослабляют, и это достаточная причина, чтобы найти лучшее решение ваших проблем.

## Причины для использования глобальных данных

Ревнители чистоты данных иногда утверждают, что программисты никогда не должны использовать глобальные данные. Но большинство программ работают с «глобальными данными» в широком смысле этого слова. Записи в базе данных являются глобальными, так же как и данные конфигурационных файлов, например реестра Windows. Именованные константы — это тоже глобальные данные, хотя и не глобальные переменные.

При аккуратном применении глобальные переменные могут быть полезны в некоторых ситуациях.

***Хранение глобальных значений*** Иногда какие-то данные концептуально относятся к целой программе. Это может быть переменная, отражающая состояние программы, скажем, режим командной строки, или интерактивный, или нормальный режим, или режим восстановления после сбоев. Или это может быть информация, необходимая в течение всей программы, например, таблица с данными, используемая всеми методами программы.

***Эмуляция именованных констант*** Хотя C++, Java, Visual Basic и большинство современных языков поддерживают именованные константы, некоторые языки, такие как Python, Perl, Awk и язык сценариев UNIX, до сих пор — нет. Вы можете использовать глобальные переменные как подстановки для именованных констант, если ваш язык их не поддерживает. Так, вы можете заменить константные значения *1* и *0* глобальными переменными *TRUE* и *FALSE*, установленными в *1* и *0*. Или вы можете заменить число *66*, используемое как число строк на странице, переменной *LINES_PER_PAGE = 66*. Этот подход позволяет упростить дальнейшее изменение кода, кроме того, его легче читать. Такое упорядоченное применение

**Перекрестная ссылка** Об именованных константах см. раздел 12.7

глобальных данных — отличный пример программирования *с использованием* языка, а не *на* языке (см. раздел 34.4).

***Эмуляция перечислимых типов*** Вы также можете использовать глобальные переменные для эмуляции перечислимых типов в таких языках, как Python, которые напрямую такие типы не поддерживают.

***Оптимизация обращений к часто используемым данным*** Иногда переменная так часто вызывается, что упоминается в списке параметров каждого метода. Вместо того чтобы включать ее в каждый список параметров, вы можете сделать ее глобальной. Однако случаи, когда к переменной обращаются отовсюду, редки. Обычно она используется ограниченным набором методов. Их вы можете объединить в класс вместе с данными, с которыми они работают. Позднее мы вернемся к этому вопросу.

***Исключение бродячих данных*** Иногда вы передаете данные методу или классу только для того, чтобы передать в другой метод или класс. Например, у вас может быть объект-обработчик ошибок, применяемый в каждом методе. Если метод в середине цепочки вызовов не использует этот объект, он называется «бродячим» (tramp data). Применение глобальных переменных помогает исключить бродячие данные.

## Используйте глобальные данные только как последнее средство

Прежде чем вы решите использовать глобальные данные, рассмотрите следующие альтернативы.

***Начните с объявления всех переменных локальными и делайте их глобальными только по необходимости*** Изначально сделайте все переменные локальными по отношению к конкретным методам. Если выяснится, что они нужны еще где-то, сделайте их сначала закрытыми или защищенными переменными класса, прежде чем вы решите сделать их глобальными. Если в конце концов выяснится, что их придется сделать глобальными, сделайте, но только после того, как в этом убедитесь. Если вы с самого начала объявите переменную глобальной, вы никогда не сделаете ее локальной, но если она сначала будет локальной, вам, возможно, не понадобится делать ее глобальной.

***Различайте глобальные переменные и переменные-члены класса*** Некоторые переменные действительно глобальны в том плане, что к ним обращаются из любого места программы. Другие — на самом деле классовые переменные — интенсивно используются только некоторымо набором методов. Вполне нормально обращаться к классовой переменной из нескольких методов сколь угодно интенсивно. Если методу вне класса нужно использовать эту переменную, предоставьте ее значение посредством метода доступа. Не обращайтесь к членам класса напрямую (как если бы эти переменные были глобальными), даже если ваш язык программирования это позволяет. Этот совет равносилен высказыванию «Модуляризируйте! Модуляризируйте! Модуляризируйте!».

***Используйте методы доступа*** Создание методов доступа — основной подход для решения проблем с глобальными данными (см. следующий раздел).

## Используйте методы доступа вместо глобальных данных

Все, что вы можете сделать с глобальными данными, вы можете сделать лучше, используя методы доступа. Применение методов доступа — основная технология реализации абстрактных типов данных и достижения сокрытия информации. Даже если вы не хотите создавать полноценный абстрактный тип данных, вы все равно можете использовать методы доступа для централизации управления данными и для защиты от изменений.

### Преимущества методов доступа

Использование методов доступа имеет несколько преимуществ.

- Вы получаете централизованный контроль над данными. Если позднее вы обнаружите более подходящую реализацию структуры, вам не придется изменять код везде, где она упоминается. Изменения не всколыхнут всю вашу программу — они останутся внутри методов доступа.
- Вы можете быть уверены, что все ссылки на переменную изолированы. Добавляя элемент в стек с помощью таких выражений, как *stack.array[ stack.top ] = newElement*, вы легко можете забыть проверить переполнение стека и допустить серьезную ошибку. Используя же методы доступа (скажем, *PushStack( newElement )*), вы можете написать проверку переполнения стека в методе *PushStack()*. Проверка будет выполняться автоматически, и вы сможете про нее забыть.

**Перекрестная ссылка** Об изоляции данных см. раздел 8.5.

- Вы автоматически получаете главные преимущества сокрытия информации. Методы доступа представляют собой примеры сокрытия информации, даже если вы и не намеревались использовать их в этих целях. Вы можете изменять содержимое метода доступа, не затрагивая остальную часть программы. Эти методы позволяют вам отремонтировать интерьер вашего дома, оставив фасад прежним, так что ваши друзья смогут его узнать.

**Перекрестная ссылка** О сокрытии информации см. подраздел «Скрывайте секреты (к вопросу о сокрытии информации)» раздела 5.3.

- Методы доступа легко преобразуются в абстрактные типы данных. Одно из преимуществ этих методов в том, что вы можете создавать более высокий уровень абстракции, чем при использовании глобальных данных напрямую. Например, вместо кода *if lineCount > MAX_LINES* вы сможете, используя метод доступа, написать *if PageFull()*. Это небольшое изменение документирует цель проверки *if lineCount* прямо в коде программы. Оно дает небольшой выигрыш в читабельности, но постоянное внимание к таким деталям и создает различие между красиво написанным ПО и наскоро слепленным кодом.

### Как использовать методы доступа

Здесь представлена краткая версия теории и практики методов доступа: Скройте данные в классе. Объявите их с помощью ключевого слова *static* или аналогичного, чтобы гарантировать их существование в единственном экземпляре. Напишите методы, позволяющие получать и изменять данные. Потребуйте, чтобы код вне класса использовал эти методы, а не данные напрямую.

Например, если у вас есть глобальная статусная переменная *g_globalStatus*, описывающая общее состояние программы, вы можете создать два метода доступа:

*globalStatus.Get()* и *globalStatus.Set()*, каждый из которых делает то, что сказано в ее названии. Эти методы обращаются к переменной, спрятанной внутри класса, заменяющего *g_globalStatus*. Остальная часть программы может получать все преимущества бывшей глобальной переменной, вызывая *globalStatus.Get()* и *globalStatus.Set()*.

**Перекрестная ссылка** Ограничение доступа к глобальным переменным, даже если ваш язык не поддерживает это напрямую, — пример программирования с использованием языка, а не на языке (см. раздел 34.4).

Если язык не поддерживает классы, вы все равно можете создавать методы доступа для манипуляции глобальными данными. Однако вам придется устанавливать ограничения доступа к глобальным данным с помощью стандартов кодирования, а не встроенных средств языка.

Вот несколько основных принципов применения методов доступа для сокрытия глобальных переменных в языках, не имеющих встроенной поддержки этого.

***Требуйте, чтобы весь код обращался к данным через методы доступа*** Хорошим соглашением будет начинать имена всех глобальных переменных с префикса *g_*, и в дальнейшем требовать, чтобы никакой код не обращался к переменным с префиксом *g_* напрямую, кроме методов доступа к этим переменным. Весь остальной код работает с этими данными через методы доступа.

***Не валите все глобальные данные в одну кучу*** Если вы сложите все глобальные данные в одну большую кучу и напишете для них методы доступа, вы решите проблему глобальных данных, но утратите некоторые преимущества абстрактных типов данных и сокрытия информации. Раз уж вы пишете методы доступа, обдумайте, к каким классам принадлежит каждая глобальная переменная, и затем упакуйте данные и методы доступа к ним в этот класс.

***Управляйте доступом к глобальным переменным с помощьюблокировок*** По аналогии с управлением параллельным доступом к многопользовательской базой данных, блокировка требует, чтобы перед вызовом или обновлением значения глобальной переменной ее помечали для изменений (check out). После использования переменную можно освободить (check in). Если пока она занята (т. е. помечена для изменений), другая часть программы попытается к ней обратиться, процедура блокировки выводит сообщение об ошибке или генерирует исключение.

**Перекрестная ссылка** О планировании различий между рабочей и промышленной версиями программы см. подраздел «Запланируйте удаление отладочных средств» раздела 8.6, а также раздел 8.7.

Такое описание механизма блокировок опускает многие тонкости в написании кода, полностью поддерживающего параллельное выполнение. По этой причине упрощенные схемы блокировок вроде этой наиболее полезны на стадии разработки. Пока схема тщательно не продумана, она скорее всего не будет достаточно надежна для работы в промышленной версии. При вводе программы в эксплуатацию такой код должен быть заменен на более безопасный и выполняющий более элегантные действия, чем вывод сообщений об ошибках. Так, при обнаружении ситуации, когда несколько частей программы пытаются заблокировать одну и ту же глобальную переменную, он мог бы записать сообщение об ошибке в файл.

Такой способ защиты во время разработки довольно легко реализовать, если вы используете методы доступа. Но это было бы затруднительно сделать, если бы вы обращались к данным напрямую.

***Встройте уровень абстракции в методы доступа*** Разрабатывайте методы доступа в области определения задачи, а не на уровне деталей реализации. Этот подход позволяет улучшить читабельность, а также страхует от изменения деталей реализации.

Сравните пары выражений в табл. 13-1:

**Табл. 13-1. Обращение к глобальным данным напрямую и с помощью метода доступа**

| Непосредственное использование глобальных данных | Обращение к глобальным данным через методы доступа |
|---|---|
| *node = node.next* | *account = NextAccount( account )* |
| *node = node.next* | *employee = NextEmployee( employee )* |
| *node = node.next* | *rateLevel = NextRateLevel( rateLevel )* |
| *event = eventQueue[ queueFront ]* | *event = HighestPriorityEvent()* |
| *event = eventQueue[ queueBack ]* | *event = LowestPriorityEvent()* |

Смысл первых трех примеров в том, что абстрактный метод доступа гораздо информативнее общей структуры. Если вы используете структуры напрямую, вы одновременно делаете слишком многое: во-первых, показываете, что выполняет структура (переход к следующему элементу в связном списке), а во-вторых — что происходит по отношению к сущности, которую она представляет (выбор номера счета, следующего работника или процентной ставки). Это слишком тяжелая ноша для простой операции присваивания в структуре данных. Сокрытие информации за абстрактными методами доступа позволяет коду самому говорить за себя и заставляет читать программу на уровне области определения задачи, а не на уровне деталей реализации.

***Выполняйте доступ к данным на одном и том же уровне абстракции*** Если вы используете метод доступа для выполнения какого-то действия со структурой, все остальные действия должны производиться с помощью таких методов. Если вы считываете данные с помощью метода доступа, то и записывайте их с помощью метода. Если вы вызываете *InitStack()* для инициализации стека и *PushStack()* для добавления в него элементов, то вы создали целостное представление данных. Если же вы извлекаете элементы с помощью выражения *value = array[ stack.top ]*, то это представление данных противоречиво. Противоречивость усложняет код для понимания. Создайте метод *PopStack()* и используйте вместо *value = array[ stack.top ]*.

В примерах выражений в табл. 13-1. две операции с очередями событий происходят параллельно. Добавление в очередь — наиболее сложная из этих двух операций в таблице и потребует нескольких строк кода для поиска места вставки события, сдвига остальных элементов очереди для выделения места новому событию, и установки нового начала или конца очереди. Удаление события из очереди по сложности будет примерно таким же. Если во время кодирования сложные операции будут помещены в мето-

**Перекрестная ссылка** Применение методов доступа для очереди событий предполагает необходимость создания класса (см. главу 6).

ды, а в остальных будет применяться прямой доступ к данным, это создаст безобразное, нераспараллеливаемое использование структуры. Теперь сравните пары выражений в табл. 13-2:

**Табл. 13-2. Распараллеливаемое и нераспараллеливаемое применение сложных данных**

| Нераспараллеливаемое использование сложных данных | Распараллеливаемое использование сложных данных |
|---|---|
| *event = EventQueue[ queueFront ]* | *event = HighestPriorityEvent()* |
| *event = EventQueue[ queueBack ]* | *event = LowestPriorityEvent()* |
| *AddEvent( event )* | *AddEvent( event )* |
| *eventCount = eventCount - 1* | *RemoveEvent( event )* |

Может показаться, что эти принципы стоит применять только в больших программах, однако методы доступа показали себя как эффективный способ решения проблем с глобальными данными. В качестве бонуса они делают код более читабельным и добавляют гибкость.

## Как уменьшить риск использования глобальных данных

Как правило, глобальные данные должны быть переменными класса, который не был правильно спроектирован или разработан. В редких случаях данные действительно должны быть глобальными, но доступ к ним может осуществляться посредством оболочки методов доступа, что позволит минимизировать потенциальные проблемы. В крохотном числе оставшихся вариантов вам действительно необходимы глобальные данные. В этих случаях вы можете рассматривать принципы, перечисленные ниже, как прививки, дающие возможность пить воду в зарубежной поездке: они болезненны, но увеличивают шансы остаться здоровым.

**Перекрестная ссылка** О соглашениях по именованию глобальных переменных см. подраздел «Идентифицируйте глобальные переменные» раздела 11.4.

***Разработайте соглашения по именованию, которые сделают глобальные переменные очевидными*** Вы можете избежать некоторых ошибок, просто сделав очевидным факт, что вы работаете с глобальными данными. Если вы используете глобальные переменные для нескольких целей (например, как переменные и как замену именованных констант), убедитесь, что ваши соглашения по именованию делают различия между этими типами использования.

***Создайте хорошо аннотированный список всех глобальных переменных*** Если соглашение по именованию указывает, что данная переменная является глобальной, будет полезно показать, что эта переменная делает. Список глобальных переменных — один из наиболее полезных инструментов, который может иметь программист, работающий с вашей программой.

***Не храните промежуточных результатов в глобальных переменных*** Если вам нужно вычислить новое значение глобальной переменной, присвойте ей окончательный результат в конце вычислений, а не храните в ней результаты промежуточных расчетов.

***Не считайте, что вы не используете глобальные переменные, поместив все данные в чудовищный объект и передавая его всюду*** Размещение всех возможных данных в одном огромном объекте может формально удовлетворять принципу отказа от глобальных переменных, но это приводит исключительно к накладным расходам и не создает преимуществ реальной инкапсуляции. Если вы используете глобальные данные, делайте это открыто. Не пытайтесь замаскировать это с помощью объектов, страдающих ожирением.

# Дополнительные ресурсы

Далее указаны дополнительные ресурсы, в которых освещаются необычные типы данных:

http://cc2e.com/1385

Maguire Steve. *Writing Solid Code*. Redmond, WA: Microsoft Press, 1993. Глава 3 содержит отличное обсуждение опасностей использования указателей и множество специальных советов по решению проблем с указателями.

Meyers Scott. *Effective C++*, 2d ed. Reading, MA: Addison-Wesley, 1998; Meyers Scott. *More Effective C++*. Reading, MA: Addison-Wesley, 1996. Как говорится в названии, эти книги содержат большое количество советов по улучшению программ на C++, включая руководство по безопасному и эффективному использованию указателей. В частности, «More Effective C++» содержит отличное обсуждение вопросов управления памятью в языке C++.

### Контрольный список: применение необычных типов данных

http://cc2e.com/1392

**Структуры**

- ❑ Используете ли вы структуры вместо отдельных переменных для организации и манипуляции группами взаимосвязанных данных?
- ❑ Рассматривали ли вы создание класса как альтернативу использованию структуры?

**Глобальные данные**

- ❑ Действительно ли все переменные объявлены локально или в области видимости класса, если только они не обязательно должны быть глобальными?
- ❑ Различаются ли в соглашениях по именованию переменных локальные, классовые и глобальные данные?
- ❑ Документированы ли все глобальные переменные?
- ❑ Свободен ли код от псевдоглобальных данных — мамонтообразных объектов, содержащих мешанину из данных, передающихся в каждый метод?
- ❑ Используются ли методы доступа вместо глобальных данных?
- ❑ Организованы ли данные и методы доступа к ним в классы?
- ❑ Предоставляют ли методы доступа уровень абстракции, независимый от реализации используемого типа данных?
- ❑ Находятся ли все методы доступа на одном уровне абстракции?

**Указатели**

- ❑ Изолированы ли операции с указателями в методах?
- ❑ Корректны ли обращения к указателям или они могут быть «висячими»?

- ❑ Проверяет ли код корректность указателей перед их использованием?
- ❑ Проверяется ли корректность переменной, на которую ссылается указатель, перед ее использованием?
- ❑ Присваивается ли указателям пустое значение после их освобождения?
- ❑ Использует ли код все необходимые для читабельности переменные-указатели?
- ❑ Освобождаются ли указатели в связных списках в правильном порядке?
- ❑ Выделяет ли программа «резервный парашют» памяти, чтобы иметь возможность аккуратно завершить выполнение в случае нехватки памяти?
- ❑ Используются ли указатели только как последнее средство, когда другие методы неприменимы?

## Ключевые моменты

- Структуры могут помочь сделать программы менее сложными, упростить их понимание и сопровождение.
- Принимая решение использовать структуру, подумайте, не будет ли класс подходить лучше.
- Работа с указателями чревата ошибками. Обезопасьте себя, используя методы или классы для доступа к ним и практику защитного программирования.
- Избегайте глобальных переменных не только потому, что они опасны, но и потому что их можно заменить чем-то лучшим.
- Если вы не можете отказаться от глобальных переменных, работайте с ними через методы доступа. Эти методы предоставляют все то же и даже больше, что и глобальные переменные.

Часть IV

# ОПЕРАТОРЫ

ГЛАВА 14

# Организация последовательного кода

http://cc2e.com/1465

**Содержание**



**Связанные темы**

- Общие вопросы управления: глава 19
- Код с условными операторами: глава 15
- Код с операторами цикла: глава 16
- Область видимости переменных и объектов: раздел 10.4

В этой главе мы начнем рассматривать программирование не с точки зрения данных, а с точки зрения выражений. Глава представляет самую простую управляющую логику программы: размещение выражений и их блоков в последовательном порядке.

Хотя размещение последовательного кода относительно простая задача, некоторые организационные тонкости влияют на качество, корректность, читабельность и управляемость кода.

## 14.1. Операторы, следующие в определенном порядке

Проще всего организовать такие выражения, для которых важен порядок следования. Вот пример:

**Пример выражений, для которых важен порядок следования (Java)**

```
data = ReadData();
results = CalculateResultsFromData( data );
PrintResults( results );
```

Если только в этом фрагменте кода не произойдет нечто непонятное, выражения должны выполняться в указанном порядке. Данные должны быть прочитаны прежде, чем результаты могут быть вычислены, а результаты должны быть вычислены прежде, чем их можно будет напечатать.

Основная идея этого примера состоит в зависимостях. Третье выражение зависит от второго, второе — от первого. Факт зависимости одного выражения от другого в этом примере понятен из имен методов. А вот здесь зависимости менее очевидны:

**Пример выражений, для которых порядок следования важен, но не настолько очевиден (Java)**

```
revenue.ComputeMonthly();
revenue.ComputeQuarterly();
revenue.ComputeAnnual();
```

В этом случае квартальный доход вычисляется в предположении, что месячные доходы уже подсчитаны. Знание бухучета, даже в общих чертах, может вам подсказывать, что квартальные доходы должны вычисляться перед годовыми. Это зависимость, но при простом прочтении кода она не видна. А здесь зависимости не просто не очевидны, но буквально скрыты:

**Пример выражений, для которых порядковые зависимости скрыты (Visual Basic)**

```
ComputeMarketingExpense
ComputeSalesExpense
ComputeTravelExpense
ComputePersonnelExpense
DisplayExpenseSummary
```

Допустим, метод *ComputeMarketingExpense()* инициализирует переменные-члены класса, в которые все остальные методы помещают данные. В этом случае его нужно вызывать перед остальными методами. Как это узнать при прочтении кода? Исходя из того, что вызовы методов не содержат параметров, вы могли бы предположить, что каждый из этих методов использует данные класса. Но вы не можете знать это наверняка, прочитав этот код.

Если зависимости между выражениями требуют размещения их в определенном порядке, требуются дополнительные действия, чтобы сделать зависимости явными.

***Организуйте код так, чтобы зависимости были очевидными*** В предыдущем примере на Visual Basic *ComputeMarketingExpense()* не должен инициализировать классовые переменные. Имя метода предполагает, что *ComputeMarketingExpense()* работает аналогично *ComputeSalesExpense()*, *ComputeTravelExpense()* только с маркетинговыми данными, а не с данными о продажах или другими расходами. То, что *ComputeMarketingExpense()* инициализирует переменные-члены класса, — случайность, которой следует избегать. Почему инициализация должна выполняться в этом методе, а не в двух других? Пока вы не сможете придумать хорошую причину для этого, инициализацию классовых переменных следует осуще-

ствлять иным методом, например *InitializeExpenseData()*. Имя метода явно указывает на то, что он должен быть вызван перед другими расчетами расходов.

***Называйте методы так, чтобы зависимости были очевидными*** В примере на Visual Basic метод *ComputeMarketingExpense()* назван неправильно, поскольку он делает больше, чем просто вычисляет расходы на маркетинг: он еще инициализирует члены класса. Если вы против создания отдельного метода для инициализации данных, дайте по крайней мере методу *ComputeMarketingExpense()* имя, описывающее все выполняемые им функции. В данном случае *ComputeMarketingExpenseAndInitializeMemberData()* будет более адекватным именем. Вы можете сказать, что это имя ужасно, потому что слишком длинное. Но оно описывает то, что делает метод и вовсе не ужасно. Ужасен сам метод!

**Перекрестная ссылка** Об использовании методов и их параметров см. главу 5.

***Используйте параметры методов, чтобы сделать зависимости очевидными*** Возвращаясь к примеру на Visual Basic, можно сказать, что, поскольку никакие данные между методами не передаются, неизвестно, используют ли эти методы одни и те же данные. Переписав код так, чтобы происходила передача данных, вы сообщаете, что порядок выполнения имеет значение. Новый код может выглядеть, например, так:

**Пример данных, которые позволяют предположить порядковую зависимость (Visual Basic)**

```
InitializeExpenseData( expenseData )
ComputeMarketingExpense( expenseData )
ComputeSalesExpense( expenseData )
ComputeTravelExpense( expenseData )
ComputePersonnelExpense( expenseData )
DisplayExpenseSummary( expenseData )
```

Поскольку все методы используют *expenseData*, это наводит на мысль, что они могут работать с одними и теми же данными и что порядок выражений может быть важен.

В этом примере лучшим подходом может быть преобразование процедур в функции, которые принимают *expenseData* на входе и возвращают обновленное значение *expenseData*. Это сделает наличие зависимостей в коде еще более явным.

**Пример данных и вызовов методов, которые указывают на порядковую зависимость (Visual Basic)**

```
expenseData = InitializeExpenseData( expenseData )
expenseData = ComputeMarketingExpense( expenseData )
expenseData = ComputeSalesExpense( expenseData )
expenseData = ComputeTravelExpense( expenseData )
expenseData = ComputePersonnelExpense( expenseData )
DisplayExpenseSummary( expenseData )
```

Данные могут также указывать, что порядок выполнения не имеет значения, как в этом случае:

**Пример данных, которые не указывают на порядковую зависимость (Visual Basic)**

```
ComputeMarketingExpense( marketingData )
ComputeSalesExpense( salesData )
ComputeTravelExpense( travelData )
ComputePersonnelExpense( personnelData )
DisplayExpenseSummary( marketingData, salesData, travelData, personnelData )
```

Так как методы в первых четырех строках не имеют общих данных, код подразумевает, что порядок их вызова значения не имеет. Поскольку метод в пятой строке использует данные каждого из первых четырех методов, вы можете предположить, что его надо выполнять после всех этих методов.

***Документируйте неявные зависимости с помощью комментариев*** Попробуйте, во-первых, написать код без порядковых зависимостей, во-вторых — написать код, который делает зависимости очевидными. Если вам все еще кажется, что зависимости видны недостаточно ясно, задокументируйте их. Документирование неявных зависимостей — один из аспектов документирования допущений, сделанных при кодировании, что необходимо для написания систем, пригодных для сопровождения и модификации. В примере на Visual Basic будет полезно поместить такие комментарии:

**Пример выражений, в которых порядковые зависимости скрыты, но разъясняются с помощью комментариев (Visual Basic)**

```
‘ Рассчитываем расходы. В каждом методе используется переменная класса
’ expenseData. Метод DisplayExpenseSummary должен вызываться последним,
’ так как он зависит от данных, вычисленных другими методами.
InitializeExpenseData
ComputeMarketingExpense
ComputeSalesExpense
ComputeTravelExpense
ComputePersonnelExpense
DisplayExpenseSummary
```

В этом коде не используются методики, проясняющие порядковые зависимости. Было бы лучше положиться на такие методики, а не на простые комментарии, но если вы сопровождаете код, находящийся под строгим контролем, или почему-либо не можете его улучшать, используйте документирование для компенсации недостатков кодирования.

***Проверяйте зависимости с помощью утверждений или кода обработки ошибок*** Если последовательность кода достаточно критична, вы можете использовать утверждения или статусные переменные и код обработки ошибок, чтобы задокументировать необходимый порядок. Например, в конструкторе класса вы можете инициализировать член класса *isExpenseDataInitialized* значением *false*. Затем в *InitializeExpenseData()* вы устанавливаете *isExpenseDataInitialized* в *true*. Каждая функция, зависящая от инициализации *expenseData*, может проверить, установлено ли значение *isExpenseDataInitialized* в *true*, прежде чем выполнять операции с *expenseData*. Если зависимости между методами глубже, вам могут потребоваться такие переменные, как *isMarketingExpenseComputed*, *isSalesExpenseComputed* и т. д.

Этот способ требует создания новых переменных, нового кода инициализации и нового кода проверки ошибок, что увеличивает возможность добавления ошибок. Преимущества этого подхода должны сравниваться с привнесенной им сложностью и увеличением вероятности появления вторичных ошибок.

## 14.2. Операторы, следующие в произвольном порядке

Вам могут встречаться ситуации, когда кажется, что порядок выполнения нескольких выражений или нескольких блоков кода не имеет значения. Одно выражение не зависит от другого и логически из него не следует. Но поскольку упорядоченность влияет на читабельность, производительность и качество сопровождения, вы можете использовать второстепенные критерии для определения порядка следования выражений или блоков кода. Главный принцип — это Принцип Схожести: *Располагайте взаимосвязанные действия вместе.*

### Размещение кода для чтения сверху вниз

Основная идея в том, что необходимо позволить читать программу сверху вниз, а не перескакивая с места на место. Эксперты согласны, что порядок просмотра кода сверху вниз способствует улучшению читабельности. Однако простого размещения последовательности команд сверху вниз недостаточно. Если тому, кто читает ваш код, приходится просматривать всю программу в поиске необходимой информации, то такой код нужно реорганизовать. Рассмотрим пример:

**Пример плохого кода, в котором приходится перескакивать с места на место (C++)**

```
MarketingData  marketingData;
SalesData  salesData;
TravelData  travelData;

travelData.ComputeQuarterly();
salesData.ComputeQuarterly();
marketingData.ComputeQuarterly();

salesData.ComputeAnnual();
marketingData.ComputeAnnual();
travelData.ComputeAnnual();

salesData.Print();
travelData.Print();
marketingData.Print();
```

Допустим, вы хотите выяснить, как рассчитывается *marketingData*. Вам придется начать с последней строки и проследить все упоминания *marketingData* вплоть до первой строки. *marketingData* встречается только в нескольких других местах, но вы должны помнить, как *marketingData* используется в каждом случае между

первым и последним своим упоминанием. Иначе говоря, вам придется просмотреть и осмыслить каждую строку кода в этом фрагменте, чтобы выяснить, как вычисляется *marketingData*. И, разумеется, этот пример гораздо проще, чем код, встречающийся в реальных системах. Вот тот же код, но лучше организованный:

**Пример хорошего, последовательного кода, который читается сверху вниз (C++)**

```
MarketingData  marketingData;
marketingData.ComputeQuarterly();
marketingData.ComputeAnnual();
marketingData.Print();

SalesData  salesData;
salesData.ComputeQuarterly();
salesData.ComputeAnnual();
salesData.Print();

TravelData  travelData;
travelData.ComputeQuarterly();
travelData.ComputeAnnual();
travelData.Print();
```

Этот код лучше по нескольким причинам. Упоминания каждой переменной располагаются вместе — они «локализованы». Число строк кода, в которых объекты являются «живыми», невелико. И, возможно, самое важное: код теперь выглядит так, что его можно разбить на отдельные методы для данных по маркетингу, продажам и поездкам. Первый фрагмент не содержал подсказки, что эта декомпозиция возможна.

**Перекрестная ссылка** Более формальное определение «живых» переменных см. в подразделе «Измерение времени жизни переменной» раздела 10.4.

## Группировка взаимосвязанных выражений

Размещайте взаимосвязанные выражения вместе. Они могут быть связаны, так как работают с одними и теми же данными, выполняют схожие задачи или зависят от порядка выполнения друг друга.

**Перекрестная ссылка** Если вы придерживаетесь Процесса Программирования с Псевдокодом, ваш код будет автоматически группироваться в блоки взаимосвязанных выражений (см. главу 9).

Есть простой способ убедиться, что взаимосвязанные выражения хорошо сгруппированы. Распечатайте текст вашего метода и обведите рамкой взаимосвязанные выражения. Если они хорошо упорядочены, вы получите картинку, похожую на рис. 14-1, где рамки не перекрываются.

***Рис. 14-1.*** *Если код хорошо организован в группы, то рамки вокруг взаимосвязанных разделов не перекрываются; они могут быть вложенными*

**Перекрестная ссылка** Об объединении операций над переменными см. раздел 10.4.

Если выражения плохо организованы, вы получите картинку, похожую на рис. 14-2, где рамки перекрываются. Если выяснится, что перекрытие происходит, реорганизуйте ваш код, чтобы взаимосвязанные выражения были лучше сгруппированы.

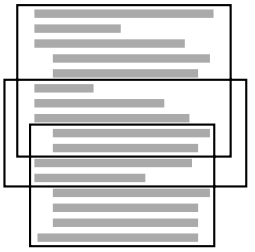


***Рис. 14-2.*** *Если код организован неудачно, то рамки вокруг связанных разделов пересекаются*

После того, как вы сгруппируете взаимосвязанные выражения, может выясниться, что они сильно связаны между собой, а к предшествующему и последующему коду не имеют значимого отношения. В этом случае, возможно, следует выделить эти выражения в отдельный метод.

http://cc2e.com/1472

### Контрольный список: организация последовательного кода

- ❑ Способствует ли код выявлению зависимостей между выражениями?
- ❑ Способствуют ли имена методов выявлению зависимостей?
- ❑ Способствуют ли параметры методов выявлению зависимостей?
- ❑ Описывают ли комментарии такие зависимости, которые иначе не будут явными?
- ❑ Используются ли вспомогательные переменные для проверки последовательных действий в критических частях кода?
- ❑ Возможно ли прочтение кода сверху вниз?
- ❑ Сгруппированы ли вместе взаимосвязанные выражения?
- ❑ Перенесены ли относительно независимые группы выражений в отдельные методы?

## Ключевые моменты

- Главный принцип организации последовательного кода — упорядочение зависимостей.
- Зависимости должны быть сделаны явными с помощью хороших имен методов, списков параметров, комментариев и — если последовательность кода достаточно критична — с помощью вспомогательных переменных.
- Если порядковые зависимости в коде отсутствуют, старайтесь размещать взаимосвязанные выражения как можно ближе друг к другу.

ГЛАВА 15

# Условные операторы

http://cc2e.com/1538

**Содержание**



**Связанные темы**

- Укрощение глубокой вложенности: раздел 19.4
- Общие вопросы управления: глава 19
- Код с операторами цикла: глава 16
- Последовательный код: глава 14
- Отношения между типами данных и управляющими структурами: раздел 10.7

Условный оператор управляет выполнением других операторов. Их выполнение «обусловливают» такие операторы, как *if, else, case* и *switch*. Хотя операторы цикла, например *while* и *for,* по смыслу тоже можно отнести к условным, обычно их рассматривают отдельно. Операторы *while* и *for* мы обсудим в главе 16.

## 15.1. Операторы *if*

В зависимости от выбранного языка программирования вы можете использовать несколько видов *if*-операторов. Простейшие из них — *if* или *if-then*. Оператор *if-then-else* немного сложнее, а наибольшую сложность представляют последовательности *if-then-else-if*.

### Простые операторы *if-then*

Следуйте этим правилам при написании *if*-выражений.

***Сначала напишите код номинального хода алгоритма, затем опишите исключительные случаи*** Пишите код так, чтобы нормальный путь выполнения был очевиден. Убедитесь, что нестандартные обстоятельства не затмевают смысл основного алгоритма. Это важно как с точки зрения читабельности, так и с точки зрения производительности.

***Убедитесь, что при сравнении на равенство ветвление корректно*** Использование > вместо >= или < вместо <= — это аналог ошибки потери единицы при

обращении к массиву или вычислении индекса цикла. Чтобы ее избежать, в операторе цикла следует рассматривать граничные точки, а в условных операторах — учитывать случаи равенства.

***Размещайте нормальный вариант после* if*, а не после* else** Пишите код так, чтобы нормальный вариант развития событий обрабатывался в первую очередь. Это совпадает с главным принципом размещения действий, являющихся результатом выбора, как можно ближе к точке этого выбора. Вот пример кода, который выполняет неоднократную проверку ошибок, беспорядочно разбросанную по тексту:

**Перекрестная ссылка** Другие способы обращения с кодом обработки ошибок описаны в разделе 19.4.

### Пример кода, который беспорядочно обрабатывает многочисленные ошибки (Visual Basic)

```
OpenFile( inputFile, status )
If ( status = Status_Error ) Then

Ошибочная ситуация.

   errorType = FileOpenError
Else

Нормальная ситуация.

   ReadFile( inputFile, fileData, status )
   If ( status = Status_Success ) Then

Нормальная ситуация.

      SummarizeFileData( fileData, summaryData, status )
      If ( status = Status_Error ) Then

Ошибочная ситуация.

         errorType = ErrorType_DataSummaryError
      Else

Нормальная ситуация.

         PrintSummary( summaryData )
         SaveSummaryData( summaryData, status )
         If ( status = Status_Error ) Then

Ошибочная ситуация.

            errorType = ErrorType_SummarySaveError
         Else

Нормальная ситуация.

            UpdateAllAccounts()
            EraseUndoFile()
            errorType = ErrorType_None
         End If
      End If
   Else
```

```
      errorType = ErrorType_FileReadError
   End If
End If
```

Этот код сложен для понимания, так как в нем перемешаны нормальные и ошибочные ситуации. Тяжело проследить путь, проходимый в коде при нормальных обстоятельствах. Кроме того, так как ошибки иногда обрабатываются в блоке *if*, а не *else*, тяжело найти, в каких же ветвях *if* обрабатываются нормальные ситуации. В переписанном примере нормальный путь последовательно кодируется первым, а ошибочные ситуации — последними. Это упрощает поиск и чтение номинального варианта алгоритма.

### Пример кода, который систематично обрабатывает большое количество ошибок (Visual Basic)

```
OpenFile( inputFile, status )
If ( status = Status_Success ) Then
```

Нормальная ситуация.

```
   ReadFile( inputFile, fileData, status )
   If ( status = Status_Success ) Then
```

Нормальная ситуация.

```
      SummarizeFileData( fileData, summaryData, status )
      If ( status = Status_Success ) Then
```

Нормальная ситуация.

```
         PrintSummary( summaryData )
         SaveSummaryData( summaryData, status )
         If ( status = Status_Success ) Then
```

Нормальная ситуация.

```
            UpdateAllAccounts()
            EraseUndoFile()
             errorType = ErrorType_None
         Else
```

Ошибочная ситуация.

```
            errorType = ErrorType_SummarySaveError
         End If
      Else
```

Ошибочная ситуация.

```
         errorType = ErrorType_DataSummaryError
      End If
   Else
```

Ошибочная ситуация.

```
      errorType = ErrorType_FileReadError
   End If
Else
```

```
Ошибочная ситуация.
    errorType = ErrorType_FileOpenError
End If
```

Здесь можно проследить главное направление *if*-проверок, чтобы выяснить нормальный вариант событий. Этот фрагмент позволяет фокусировать чтение в основном направлении, а не преодолевать исключительные ситуации, поэтому этот код в целом читабельнее. Стек ошибочных условий, расположенный внизу вложения, — признак хорошо написанного кода обработки ошибок.

Этот пример иллюстрирует один систематический подход к обработке нормальных и ошибочных ситуаций. Другие решения этой проблемы: использование предохранительных конструкций, диспетчеризация полиморфных объектов, вынесение внутренних проверок в отдельные методы — обсуждаются на протяжении всей книги. Полный список существующих подходов см. в разделе 19.4.

***Размещайте осмысленные выражения после оператора if*** Иногда можно встретить код, в котором блок *if* пуст:

**Пример пустого блока *if* (Java)**

```
if ( SomeTest )
   ;
else {
   // делаем что-то
   ...
}
```

Опытные программисты не станут писать такой код хотя бы затем, чтобы избежать лишней работы по вводу пустой строки и оператора *else*. Этот код выглядит глупо и может быть легко улучшен путем отрицания предиката в выражении *if*, перемещения кода из блока *else* в блок *if* и удаления блока *else*. Вот как будет выглядеть код после таких изменений:

**Перекрестная ссылка** Один из ключей к конструированию эффективного оператора *if*- — создание правильного управляющего логического выражения. Об эффективном применении логических выражений см. раздел 19.1.

**Пример преобразования пустого блока *if* (Java)**

```
if ( ! someTest ) {
   // делаем что-то
   ...
}
```

***Рассмотрите вопрос использования блока* else** Если вы считаете, что вам нужен простой оператор *if*, подумайте, может, вам на самом деле нужен вариант с *if-then-else*. Классический анализ General Motors показал, что в 50–80% случаев использования операторов *if* следовало применять и оператор *else* (Elshoff, 1976).

Одна из причин добавления блока *else* — даже пустого — в том, чтобы продемонстрировать, что вариант с *else* был учтен. Конечно, кодирование пустых выражений в *else* просто для того, чтобы показать, что этот вариант рассмотрен, может

быть преувеличением, но хотя бы принимайте вариант с *else* во внимание. Если вы задаете *if*-проверку, не имеющую блока *else*, то, кроме очевидных случаев, пишите в комментариях объяснение, почему *else* отсутствует, скажем, так:

**Пример полезного, прокомментированного блока *else* (Java)**

```
// Если цвет задан корректно.
if ( COLOR_MIN <= color && color <= COLOR_MAX ) {
   // Делаем что-то
   ...
}
else {
   // Иначе цвет задан некорректно.
   // Вывод на экран не выполняется -- просто игнорируем команду.
}
```

***Проверяйте корректность выражения* else** При тестировании кода вы можете решить, что достаточно проверить основной блок *if* и все. Однако, если можно проверить вариант в *else*, не забудьте это сделать.

***Проверяйте возможную перестановку блоков* if *и* else** Частая ошибка при программировании выражений *if-then* состоит в размещении кода из блока *if* в блоке *else*, т. е. инвертировании логики выражения *if*. Проверяйте ваш код на наличие этой ошибки.

## Последовательности операторов *if-then-else*

Если язык не поддерживает операторы *case* или поддерживает их только частично, вам часто придется писать последовательные проверки *if-then-else*. К примеру, код распределения символов по категориям, может выглядеть в виде такой цепочки:

**Перекрестная ссылка** Об упрощении сложных выражений см. раздел 19.1.

**Пример использования последовательности *if-then-else* для распределения символов по категориям (C++)**

```
if ( inputCharacter < SPACE ) {
    characterType = CharacterType_ControlCharacter;
}
else if (
   inputCharacter == ' ' ||
   inputCharacter == ',' ||
   inputCharacter == '.' ||
   inputCharacter == '!' ||
   inputCharacter == '(' ||
   inputCharacter == ')' ||
   inputCharacter == ':' ||
   inputCharacter == ';' ||
   inputCharacter == '?' ||
   inputCharacter == '-'
   ) {
    characterType = CharacterType_Punctuation;
}
```

```
else if ( '0' <= inputCharacter && inputCharacter <= '9' ) {
    characterType = CharacterType_Digit;
}
else if (
    ( 'a' <= inputCharacter && inputCharacter <= 'z' ) ||
    ( 'A' <= inputCharacter && inputCharacter <= 'Z' )
    ) {
    characterType = CharacterType_Letter;
}
```

Учитывайте советы, приведенные далее, при написании последовательных *if-then-else*.

***Упрощайте сложные проверки с помощью вызовов логических функций*** Одна из причин, по которой код из предыдущего примера сложно читать, в том, что проверки категорий символов довольно сложны. Для улучшения читабельности вы можете заменить их вызовами функций, возвращающих логические значения. Вот как этот пример может выглядеть после замены условий логическими функциями:

**Пример последовательности *if-then-else*, использующей вызовы логических функций (C++)**

```
if ( IsControl( inputCharacter ) ) {
    characterType = CharacterType_ControlCharacter;
}
else if ( IsPunctuation( inputCharacter ) ) {
    characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
    characterType = CharacterType_Digit;
}
else if ( IsLetter( inputCharacter ) ) {
    characterType = CharacterType_Letter;
}
```

***Размещайте наиболее вероятные варианты раньше остальных*** Поместив в начало наиболее часто встречающиеся ситуации, вы минимизируете то количество кода, обрабатывающего исключительные случаи, которое придется прочитать при поиске обычных вариантов. Вы увеличите эффективность, потому что уменьшите число проверок, выполняемых кодом в большинстве случаев. В приведенном примере буквы обычно встречаются чаще, чем знаки пунктуации, но проверка этих знаков написана первой. Вот как исправить код, чтобы буквы проверялись в первую очередь:

**Пример проверки прежде всего наиболее часто встречающихся вариантов (C++)**

```
Этот случай встречается чаще других, поэтому проверяем его первым.
if ( IsLetter( inputCharacter ) ) {
    characterType = CharacterType_Letter;
}
```

```
else if ( IsPunctuation( inputCharacter ) ) {
     characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
    characterType = CharacterType_Digit;
}

Этот случай наиболее редкий, поэтому проверяем его последним.

else if ( IsControl( inputCharacter ) ) {
    characterType = CharacterType_ControlCharacter;
}
```

**Перекрестная ссылка** Это также хороший пример того, как использовать последовательность *if-then-else* вместо кода глубокой вложенности. Об этой методике см. раздел 19.4.

***Убедитесь, что учтены все варианты*** Закодируйте в последнем блоке *else* сообщение об ошибке или утверждение, чтобы отловить ситуации, которые вы не планировали. Это сообщение об ошибке предназначено скорее вам, а не пользователю, так что сформулируйте его соответственно. Вот как модифицировать пример классификации символов для выполнения проверки «других вариантов»:

**Пример использования варианта по умолчанию для перехвата ошибок (C++)**

```
if ( IsLetter( inputCharacter ) ) {
    characterType = CharacterType_Letter;
}
else if ( IsPunctuation( inputCharacter ) ) {
    characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
    characterType = CharacterType_Digit;
}
else if ( IsControl( inputCharacter ) ) {
    characterType = CharacterType_ControlCharacter;
}
else {
    DisplayInternalError( "Unexpected type of character detected." );
}
```

***Замените последовательности* if-then-else *другими конструкциями, которые поддерживает ваш язык программирования*** Некоторые языки, скажем, Microsoft Visual Basic и Ada, предоставляют операторы *case*, поддерживающие строки, перечисления и логические функции. Используйте их — они проще в написании и чтении, чем цепочки *if-then-else*. Код классификации типов символов, написанный на Visual Basic с помощью оператора *case*, может выглядеть так:

**Пример использования оператора *case* вместо последовательности *if-then-else* (Visual Basic)**

```
Select Case inputCharacter
   Case "a" To "z"
      characterType = CharacterType_Letter
```

```
    Case " ", ",", ".", "!", "(", ")", ":", ";", "?", "-"
        characterType = CharacterType_Punctuation
    Case "0" To "9"
        characterType = CharacterType_Digit
     Case FIRST_CONTROL_CHARACTER To LAST_CONTROL_CHARACTER
        characterType = CharacterType_Control
    Case Else
        DisplayInternalError( "Unexpected type of character detected." )
End Select
```

# 15.2. Операторы *case*

Оператор *case* или *switch* — конструкция, сильно варьирующаяся от языка к языку. C++ и Java поддерживают *case* только для порядковых типов, рассматривая по одному значению за раз. Visual Basic поддерживает *case* для порядковых типов и предоставляет мощные средства для обозначения диапазонов и комбинаций значений. Многие языки сценариев вообще не поддерживают *case*.

Следующие разделы представляют основные принципы эффективного использования операторов *case*.

## Выбор наиболее эффективного порядка вариантов

Организовать порядок следования вариантов в операторе *case* можно по-разному. Если выражение *case* невелико и содержит только три варианта и три соответствующих строки кода, то выбранный порядок большого значения не имеет. Однако в случае длинного оператора *case* (скажем, обрабатывающего десятки различных событий в программе, управляемой событиями) порядок следования достаточно важен. Давайте рассмотрим способы упорядочения.

***Упорядочивайте варианты по алфавиту или численно*** Если все варианты равнозначны, их размещение в алфавитном порядке улучшает читабельность. В этом случае нужный вариант легко выбрать из группы.

***Поместите правильный вариант первым*** Если у вас есть один корректный вариант и несколько исключений, поместите правильное значение первым. Отметьте в комментариях, что этот вариант является нормальным, а все остальные — исключительными.

***Отсортируйте варианты по частоте*** Поместите наиболее часто встречающиеся случаи в начало, а более редкие — в конец списка. Этот подход имеет два преимущества. Первое: читатели легко обнаружат ситуации, встречающиеся чаще всего. Второе: при сканировании списка в поисках определенных значений скорее всего будут требоваться наиболее часто встречающиеся варианты, поэтому их размещение в начале кода сделает поиск быстрее.

## Советы по использованию операторов *case*

Далее перечислены советы по применению операторов *case*.

***Сделайте обработку каждого варианта простой*** Код, связанный с каждым вариантом, должен быть корот-

**Перекрестная ссылка** Другие советы по упрощению кода см. в главе 24.

ким. Краткость позволяет сделать структуру оператора *case* прозрачнее. Если действия, предпринимаемые для какого-то варианта слишком сложны, напишите метод и вызывайте его, а не размещайте весь этот код прямо в блоке *case*.

***Не конструируйте искусственные переменные с целью получить возможность использовать оператор case*** Оператор *case* следует применять для простых данных, которые легко разбить на категории. Если ваши данные нельзя назвать простыми, используйте цепочки *if-then-else*. Фальшивые переменные сбивают с толки, и их следует избегать. Например, не делайте так:

**Пример создания искусственной переменной для оператора *case* — плохая практика (Java)**

```
action = userCommand[ 0 ];
switch ( action ) {
   case 'c':
      Copy();
      break;
   case 'd':
       DeleteCharacter();
      break;
   case 'f':
      Format();
      break;
   case 'h':
      Help();
      break;
   ...
   default:
       HandleUserInputError( ErrorType.InvalidUserCommand );
}
```

Переменная *action* управляет оператором *case*. В данном случае *action* конструируется с помощью первого символа строки *userCommand*, которая вводится пользователем.

**Перекрестная ссылка** В противоположность этому совету иногда можно улучшить читабельность, присваивая значение сложного выражения хорошо названной логической переменной или функции (см. подраздел «Упрощение сложных выражений» раздела 19.1).

Этот потенциально опасный код может привести к проблемам. Вообще, если вы фабрикуете специальную переменную для *case*, реальные данные могут не вписываться в данное выражение так, как вы это себе представляете. В приведенном примере, если пользователь набирает copy, в операторе *case* отрезается первый символ «*c*», и корректно вызывается метод *Copy()*. С другой стороны, если пользователь набирает cement overshoes, clambake или cellulite, *case* точно также отрежет символ «*c*» и вызовет *Copy()*. Проверка ошибочных команд в блоке *else* оператора *case* не будет нормально функционировать, так как будет обрабатывать только неправильные первые буквы, а не команды целиком.

Вместо создания искусственной переменной в этом коде лучше использовать последовательность *if-then-else-if* и проверять целую строку. Вот как эффективно переписать код:

**Пример использования операторов *if-then-else* вместо искусственной переменной для оператора *case* — хорошая практика (Java)**

```
if ( UserCommand.equals( COMMAND_STRING_COPY ) ) {
   Copy();
}
else if ( UserCommand.equals( COMMAND_STRING_DELETE ) ) {
   DeleteCharacter();
}
else if ( UserCommand.equals( COMMAND_STRING_FORMAT ) ) {
   Format();
}
else if ( UserCommand.equals( COMMAND_STRING_HELP ) ) {
   Help();
}
...
else {
   HandleUserInputError( ErrorType_InvalidCommandInput );
}
```

***Используйте вариант по умолчанию только для обработки настоящих значений по умолчанию*** Порой, когда остается единственный вариант, вы решаете закодировать его как вариант по умолчанию. Заманчиво, но... неразумно: вы лишаетесь автоматического документирования, предоставляемого метками оператора *case*, и теряете возможность определения ошибок с помощью варианта по умолчанию.

Такие операторы *case* подвержены ошибкам при модификации. Если вы используете настоящее значение по умолчанию, то добавление нового варианта тривиально: вы просто дописываете новую метку и соответствующий код. Если же значение по умолчанию искусственно, провести изменения сложнее. Вам придется добавить новый вариант, возможно, сделав его новым умолчанием, а затем поменять прежний вариант по умолчанию так, чтобы он стал обычным вариантом. Поэтому изначально используйте только настоящие значения по умолчанию.

***Используйте вариант по умолчанию для выявления ошибок*** Если вариант по умолчанию в операторе *case* не используется и не планируется для иных действий, разместите в нем диагностическое сообщение:

**Пример использования варианта по умолчанию для выявления ошибок — хорошая практика (Java)**

```
switch ( commandShortcutLetter ) {
   case 'a':
      PrintAnnualReport();
      break;
   case 'p':
      // Никаких действий не требуется, но этот вариант предусмотрен.
      break;
   case 'q':
      PrintQuarterlyReport();
      break;
```

```
   case 's':
      PrintSummaryReport();
      break;
   default:
      DisplayInternalError( "Internal Error 905: Call customer support." );
}
```

Такие сообщения полезны и в отладочном, и в промышленном коде. Большинство пользователей предпочитают сообщения вроде «Внутренняя ошибка: Пожалуйста, позвоните в службу поддержки» отказу системы или, что еще хуже, немного неверным результатам, которые выглядят вполне нормально до того, как их проверит начальник.

Если вариант по умолчанию служит не для выявления ошибок, а для иных целей, то подразумевается, что любой выбор будет корректным. Дважды проверьте ваш код и убедитесь, что любое возможное значение на входе оператора *case* будет допустимо. Если обнаружится какой-то некорректный вариант, перепишите выражения, чтобы условие по умолчанию проверяло ошибки.

***В C++ и Java старайтесь не писать код, проваливающийся сквозь блоки оператора* case** C-подобные языки (C, C++ и Java) не выполняют выход из блока *case* автоматически. Вместо этого вы должны явно закодировать конец блока. Если этого не сделать, программа провалится сквозь этот блок и начнет выполнять следующий. Это иногда приводит к некоторым вопиющим способам программирования, включая следующий пример:

**Пример неправильного использования оператора *case* (C++)**

```
switch ( InputVar ) {
   case 'A': if ( test ) {
                  // оператор 1
                  // оператор 2
   case 'B':      // оператор 3
                  // оператор 4
                  ...
                  }
               ...
             break;
   ...
}
```

**Перекрестная ссылка** Такое форматирование кода делает пример лучше, чем он есть на самом деле. О том, как сделать, чтобы хороший код выглядел хорошо, а плохой — плохо, см. полдраздел «Разметка концов строк» раздела 31.3 и оставшуюся часть главы 31.

Этот способ плох, так как приводит к перемешиванию управляющих конструкций. Разобраться во вложенных управляющих конструкциях довольно тяжело, а в перекрывающихся — практически невозможно. Модификация вариантов *'A'* или *'B'* будет посложнее хирургической операции на головном мозге, и вполне вероятно, что их придется полностью переписать, чтобы получить работоспособную версию. Что вы и можете сделать с самого начала. В общем, лучше избегать сквозного перехода в блоках оператора *case*.

***В C++ ясно и безошибочно обозначайте код, проваливающийся сквозь блоки оператора* case** Если вы намеренно написали код, который должен выполняться в нескольких блоках *case* подряд, внятно прокомментируйте место, где это происходит, и объясните, почему это должно быть закодировано таким способом:

**Пример документирования сквозного выполнения блоков *case* (C++)**

```
switch ( errorDocumentationLevel ) {
   case DocumentationLevel_Full:
      DisplayErrorDetails( errorNumber );
      // СКВОЗНОЙ ПЕРЕХОД – Полная документация также печатает
      // суммарные комментарии.

   case DocumentationLevel_Summary:
      DisplayErrorSummary( errorNumber );
      // СКВОЗНОЙ ПЕРЕХОД – Суммарная документация
      // также печатает номер ошибки.

   case DocumentationLevel_NumberOnly:
      DisplayErrorNumber( errorNumber );
      break;

   default:
      DisplayInternalError( "Internal Error 905: Call customer support." );
}
```

Эта методика встречается так же часто, как и люди, которые предпочитают подержанный «Понтиак Ацтек» новенькому «Корвету». Обычно код, переходящий сквозь один блок *case* к другому, просто напрашивается на ошибки при модификации и его следует избегать.

### Контрольный список: использование условных операторов

http://cc2e.com/1545

**Операторы *if-then***

- ❑ Очевиден ли номинальный путь выполнения кода?
- ❑ Правильно ли выполняется ветвление при проверке *if-then* на равенство?
- ❑ Присутствует и задокументирован ли блок *else*?
- ❑ Корректен ли блок else?
- ❑ Правильно ли расположены блоки *if* и *else* — нет ли инверсии?
- ❑ Следует ли нормальный вариант после *if*, а не после else?

**Последовательности *if-then-else-if***

- ❑ Преобразуются ли сложные проверки в вызовы логических функций?
- ❑ Проверяются ли наиболее вероятные случаи первыми?
- ❑ Все ли варианты учитываются?
- ❑ Является ли последовательность *if-then-else-if* лучшей реализацией или лучше использовать оператор case?

**Операторы *case***

- ❑ Разумно ли отсортированы варианты в операторе *case*?
- ❑ Сделаны ли действия, выполняемые для каждого варианта, простыми, например, с помощью преобразования в методы в случае необходимости?
- ❑ Проверяет ли оператор *case* реальную переменную, а не искусственно созданную, приводящую к неправильному использованию оператора *case*?
- ❑ Корректны ли значения, обрабатываемые в блоке по умолчанию?
- ❑ Используется ли блок по умолчанию для выявления ошибок и сообщения о непредвиденных ситуациях?
- ❑ В языках C, C++ или Java содержит ли каждый блок *case* команды для выхода?

# Ключевые моменты

- В простых выражениях *if-else* обращайте внимание на порядок блоков *if* и *else*, особенно если они обрабатывают множество ошибок. Убедитесь, что номинальный вариант прослеживается ясно.
- Для последовательностей *if-then-else* и операторов *case* выбирайте порядок, позволяющий улучшить читабельность.
- Для перехвата ошибок используйте блок по умолчанию в операторе *case* или последний блок *else* в цепочке операторов *if-then-else*.
- Управляющие конструкции не равнозначны. Выбирайте конструкцию, наиболее подходящую для данного участка кода.

ГЛАВА 16

# Циклы

**Содержание**

http://cc2e.com/1609

- 16.1. Выбор типа цикла
- 16.2. Управление циклом
- 16.3. Простое создание цикла — изнутри наружу
- 16.4. Соответствие между циклами и массивами

**Связанные темы**

- Укрощение глубокой вложенности: раздел 19.4
- Общие вопросы управления: глава 19
- Код с условными операторами: глава 15
- Последовательный код: глава 14
- Отношения между управляющими структурами и типами данных: раздел 10.7

Цикл — это неформальное обозначение любой структуры итеративного типа, т. е. такой, которая заставляет программу повторно выполнять некий блок кода. Наиболее распространенными видами циклов являются *for, while* и *do-while* в C++ и Java, *For-Next, While-Wend* и *Do-Loop-While* — в Microsoft Visual Basic. Использование циклов — один из наиболее сложных аспектов программирования. Знание, как и когда применять каждый тип цикла, — это решающий фактор в конструировании высококачественного ПО.

## 16.1. Выбор типа цикла

В большинстве языков можно использовать несколько видов циклов, перечисленных ниже.

- Цикл с подсчетом выполняется определенное количество раз, к примеру, один раз для каждого работника.
- Постоянно вычисляемый цикл не знает заранее, сколько раз он будет выполняться и проверяет необходимость завершения при каждой итерации. Например, он выполняется, пока остаются деньги, пока пользователь не выберет команду завершения или не встретится ошибка.

- Бесконечный цикл выполняется все время с момента старта. Такие циклы можно встретить во встроенных системах, таких как кардиостимуляторы, микроволновые печи и автопилоты.
- Цикл с итератором выполняет некоторые действия однократно для каждого элемента контейнерного класса.

Первое, чем отличаются эти циклы друг от друга, — это гибкость в определении числа итераций: выполняется ли цикл указанное количество раз или проверяет необходимость завершения на каждом шаге.

Кроме того, эти варианты отличаются расположением проверки завершения. Вы можете поместить проверку в начало, середину или конец цикла. Эта характеристика определяет, будет ли цикл выполняться хоть раз. Если условие цикла проверяется в начале, то его тело не обязательно будет выполняться. Если цикл проверяется в конце, то его тело выполняется минимум один раз. Если же проверка находится в середине, то часть цикла, предшествующая ей, выполняется не менее раза, а код, следующий за проверкой, не обязательно будет выполняться.

Гибкость и расположение проверки условия определяют тип цикла, выбираемый в качестве управляющей структуры. В табл. 16-1 показаны разновидности циклов в нескольких языках программирования и характеризуются их гибкость и расположение проверки.

**Табл. 16-1. Типы циклов**

| Язык | Тип цикла | Гибкость | Место проверки |
|---|---|---|---|
| Visual Basic | *For-Next* | Нет | Начало |
| | *While-Wend* | Да | Начало |
| | *Do-Loop-While* | Да | Начало или конец |
| | *For-Each* | Нет | Начало |
| C, C++, C#, Java | *for* | Да | Начало |
| | *while* | Да | Начало |
| | *do-while* | Да | Конец |
| | *foreach*[1] | Нет | Начало |

## Когда использовать цикл *while*

Новички иногда считают, что условие цикла *while* проверяется постоянно и цикл завершается в тот момент, когда это условие становится ложным, независимо от того, какой оператор в это время выполняется (Curtis et al, 1986). Хотя цикл *while* и не столь гибок, он все же является гибким вариантом цикла. Если вы заранее не знаете, сколько итераций должен выполнить цикл, используйте *while*. Вопреки тому что думают новички, проверка выхода из цикла выполняется только раз за итерацию, и главным вопросом в отношении циклов *while* является выбор места этой проверки — в начале или конце цикла.

[1] Реализован только в C#. На момент написания книги планируется в других языках, включая Java.

### Цикл с проверкой в начале

В качестве цикла с проверкой в начале вы можете использовать цикл *while* в языках C++, C#, Java, Visual Basic и многих других. Вы также можете эмулировать цикл *while* в других языках.

### Цикл с проверкой в конце

Время от времени вам требуется гибкий цикл, который должен выполняться хотя бы раз. В этом случае можно применить *while* с проверкой условия в конце цикла. Вы можете использовать варианты *do-while* в C++, C# и Java, *Do-Loop-While* вVisual Basic или эмулировать циклы с проверкой в конце в других языках

### Когда использовать цикл с выходом

В цикле с выходом условие выхода содержится внутри тела цикла, а не в его начале или конце. Циклы с выходом реализованы исключительно в Visual Basic, но вы можете эмулировать их, используя структурированные конструкции *while* и *break* в C++, C и Java или операторы *goto* в других языках.

### Нормальные циклы с выходом

Цикл с выходом обычно состоит из начала цикла, тела цикла (включая условие выхода) и конца цикла:

#### Пример обычного цикла с выходом (Visual Basic)

```
Do

Операторы.

   ...
   If ( some exit condition ) Then Exit Do

Еще операторы.

   ...
Loop
```

Обычно цикл с выходом нужен, когда проверка условия в начале или конце цикла требует кодирования полутора циклов. Вот пример на C++, в котором нужен цикл с выходом, но он не используется:

#### Пример дублированного кода, который подвержен ошибкам при сопровождении (C++)

```
// Вычисляем счет и рейтинги.
score  = 0;

Эти строки появляются здесь...

GetNextRating( &ratingIncrement );
rating = rating + ratingIncrement;
while ( ( score < targetScore ) && ( ratingIncrement != 0 ) ) {
   GetNextScore( &scoreIncrement );
   score = score + scoreIncrement;
```

```
...и повторяются здесь.
    GetNextRating( &ratingIncrement );
    rating = rating + ratingIncrement;
}
```

Первые две строки в начале примера повторяются в последних двух строках цикла *while*. При модификации вы легко можете забыть о поддержании двух параллельных наборов строк. Другой программист, изменяющий код, возможно, и не догадается, что эти строки должны сохраняться одинаковыми. В любом случае результатом будут ошибки, возникшие из-за неполной модификации. Вот как можно переписать код более ясно:

**Пример цикла с выходом, более легкого в сопровождении (C++)**

```
// Вычисляем счет и рейтинги. Этот код использует бесконечный цикл
// и оператор break для имитации цикла с выходом.
score = 0;
while ( true ) {
    GetNextRating( &ratingIncrement );
    rating = rating + ratingIncrement;

Это условие выхода из цикла (и теперь оно может быть упрощено с помощью теорем ДеМоргана,
описанных в разделе 19.1).
    if ( !( ( score < targetScore ) && ( ratingIncrement != 0 ) ) ) {
       break;
    }

    GetNextScore( &scoreIncrement );
    score = score + scoreIncrement;
}
```

Вот как тот же код можно написать на Visual Basic:

**Пример цикла с выходом (Visual Basic)**

```
' Вычисляем счет и рейтинги.
score = 0
Do
    GetNextRating( ratingIncrement )
    rating = rating + ratingIncrement

    If ( not ( score < targetScore and ratingIncrement <> 0 ) ) Then Exit Do

    GetNextScore( ScoreIncrement )
    score = score + scoreIncrement
Loop
```

При использовании этого типа циклов учитывайте следующие тонкие моменты.

Разместите все условия выхода в одном месте. Распространение их по коду практически гарантирует, что то или иное условие завершения будет пропущено при отладке, модификации или тестировании.

Пишите комментарии для ясности. Если вы применяете цикл с выходом в языке, который не поддерживает его напрямую, используйте комментарии, чтобы сделать свои действия очевидными.

**Перекрестная ссылка** Другие сведения об условиях завершения представлены ниже в этой главе. Об использовании комментариев в циклах см. подраздел «Комментирование управляющих структур» раздела 32.5.

Цикл с выходом — структурированная управляющая конструкция, имеющая один вход и один выход. Такая структура является предпочтительным вариантом цикла (Software Productivity Consortium, 1989). Доказано, что этот тип цикла легче для понимания, чем другие. Группа студентов-программистов сравнила такой цикл с другими вариантами, имеющими выход в начале или конце (Soloway, Bonar и Ehrlich, 1983). Тесты на понимание для цикла с выходом выполнялись студентами на 25% успешнее. Авторы курса пришли к выводу, что структура цикла с выходом лучше, чем другие циклы, моделирует способ человеческого представления итеративного процесса.

В повседневной практике цикл с выходом пока еще не широко распространен. Присяжные все еще заперты в накуренной комнате, споря о том, годиться ли эта методика для промышленного кода. Пока они там томятся, цикл с выходом будет хорошим инструментом в вашем программистском наборе — при условии его аккуратного использования.

### Аномальные циклы с выходом

Другой вид цикла с выходом служит для замены следующего варианта «полуторного» цикла:

**Пример входа в середину цикла с помощью *goto* — плохая практика (C++)**

```
goto Start;
while ( expression ) {
   // Делаем что-то.
   ...

   Start:

   // Делаем что-то еще.
   ...
}
```

На первый взгляд, этот цикл похож на предыдущие примеры цикла с выходом. Он используется, если выражение, обозначенное как *// делаем что-то*, не должно выполняться при первом проходе цикла, а выражение *// делаем что-то еще* — должно. Это тоже конструкция с одним входом и выходом: единственный вход в цикл — через оператор *goto* в начале, а выход — с помощью условия *while*. Этот подход содержит две проблемы: он использует *goto* и довольно необычен, чем сбивает с толку.

В C++ вы можете добиться того же эффекта без использования *goto*, как показано в следующем примере. Если язык не поддерживает команду *break*, вы можете эмулировать ее, применив *goto*.

**Пример кода, переписанного без использования *goto* — лучший вариант (C++)**

```
while ( true ) {
```

Блоки перед и после *break* поменяны местами.

```
   // Делаем что-то еще.
   ...

   if ( !( expression ) ) {
      break;
   }

   // Делаем что-то.
   ...
}
```

## Когда использовать цикл *for*

**Дополнительные сведения** О других хороших приемах использования циклов *for* см. в «Writing Solid Code» (Maguire, 1993).

Цикл *for* — хороший вариант, если вам нужен цикл, выполняющийся определенное количество раз. Вы можете использовать *for* в C++, C, Java, Visual Basic и большинстве других языков.

Применяйте циклы *for* в простых случаях, не требующих управления изнутри тела цикла. Используйте их, когда управление циклом заключается в простом инкременте или декременте, скажем, при проходе по элементам контейнера. Особенность цикла *for* в том, что его надо настроить в начале выполнения и забыть о нем. Вам ничего не надо делать внутри него для управления его работой. Если существует условие, по которому выполнение цикла прерывается изнутри, вместо *for* используйте конструкцию *while*.

Не изменяйте значение индекса цикла *for* явно, чтобы принудительно его завершить. Вместо этого используйте *while*. Цикл *for* предназначен для простых случаев. Более сложные задачи организации циклов лучше решать с помощью цикла *while*.

## Когда использовать цикл *foreach*

Цикл *foreach* (или его эквиваленты *For-Each* в Visual Basic, *for-in* в Python) полезен для выполнения действий над каждым элементом массива или другого контейнера. Его преимущество в том, что он позволяет обойтись без вспомогательной арифметики для обслуживания цикла и, таким образом, избежать ошибок. Вот пример такого цикла:

**Пример цикла *foreach* (C#)**

```
int [] fibonacciSequence = new int [] { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
int oddFibonacciNumbers = 0;
int evenFibonacciNumbers = 0;

// Подсчитываем количество четных и нечетных чисел в последовательности Фибоначчи.
foreach ( int fibonacciNumber in fibonacciSequence ) {
   if ( fibonacciNumber % 2 ) == 0 ) {
      evenFibonacciNumbers++;
   }
   else {
      oddFibonacciNumbers++;
   }
}

Console.WriteLine( "Found {0} odd numbers and {1} even numbers.",
    oddFibonacciNumbers, evenFibonacciNumbers );
```

# 16.2. Управление циклом

Что плохого может случиться с циклом? Любой ответ должен включать некорректную или пропущенную инициализацию цикла, невыполненную инициализацию накопительных переменных (или других переменных, относящихся к циклу), неправильную вложенность, неправильное завершение цикла, отсутствие инкрементации переменной цикла или ее неправильную инкрементацию, а также неправильное индексирование элемента массива с помощью индекса цикла.

Вы можете предотвратить эти проблемы, соблюдая два правила. Во-первых, минимизируйте число факторов, влияющих на цикл. Упрощайте, упрощайте и еще раз упрощайте! Во-вторых, рассматривайте содержимое цикла так, будто это метод: вынесите за пределы цикла все управление, какое только возможно. Явно объявите все условия выполнения тела цикла. Не заставляйте читателя смотреть внутрь цикла, чтобы понять его управление. Думайте о цикле, как о черном ящике: окружающий код знает об управляющих условиях, но не о содержимом цикла.

**Пример представления цикла в виде черного ящика (C++)**

```
while ( !inputFile.EndOfFile() && moreDataAvailable ) {
```

При каких условиях этот цикл завершится? Очевидно, все, что вам известно, что или *inputFile.EndOfFile()* станет истиной, или *MoreDataAvailable* станет ложью.

**Перекрестная ссылка** Если вы используете технологию *while ( true )-break*, описанную выше, то условие выхода находится внутри черного ящика. Даже если вы используете только одно условие выхода, вы теряете преимущество рассмотрения цикла в виде черного ящика.

## Вход в цикл

Следуйте принципам, приведенным далее, при разработке входа в цикл.

***Размещайте вход в цикл только в одном месте*** Разнообразие структур, управляющих циклом, позволяет проводить проверку его завершения в начале, середине или конце цикла. Эти структуры имеют достаточно широкие возможности, чтобы вы могли закодировать вход в цикл только сверху. Нет нужды делать вход в цикл в нескольких местах.

***Размещайте инициализационный код непосредственно перед циклом*** Принцип схожести пропагандирует размещение взаимосвязанных выражений вместе. Если взаимосвязанные выражения разбросаны по всем методу, то при внесении исправлений их легко пропустить, сделав изменения не полностью. Если же взаимосвязанные выражения располагаются рядом, избежать ошибок при модификации становится легче.

**Перекрестная ссылка** Об ограничении области видимости переменных цикла см. подраздел «Ограничьте видимость переменных-индексов цикла самим циклом» далее в этой главе.

Поместите операторы инициализации цикла рядом с этим циклом. Если вы этого не сделаете, то, вполне вероятно, это приведет к ошибкам, когда вы соберетесь преобразовать данный цикл в цикл большего размера и забудете исправить инициализационный код. Такая же ошибка может возникнуть, когда вы переместите или скопируете код цикла в другой метод, забыв переместить инициализационный код. Размещение кода инициализации вдали от цикла — в разделе объявления данных или во вспомогательном разделе в начале метода — грозит неприятностями с инициализацией.

***Используйте* while (true) *для бесконечных циклов*** Вам может понадобиться цикл, выполняющийся без завершения, — например, цикл в таких изделиях, как кардиостимулятор или микроволновая печь. Или цикл должен завершаться только в ответ на событие — так называемый «событийный цикл». Вы можете закодировать такой бесконечный цикл несколькими способами. Имитация цикла с помощью выражений вида *for i = 1 to 99999* — плохая идея, поскольку конкретное значение границ цикла скрывает его смысл: *99999* может быть вполне допустимым значением. Кроме того, такой фальшивый бесконечный цикл плохо поддается сопровождению.

Идиома *while( true )* считается стандартным способом написания бесконечных циклов в C++, Java, Visual Basic и других языках, поддерживающих операции сравнения. Некоторые программисты предпочитают использовать *for( ;; )* — это приемлемая альтернатива.

***Предпочитайте циклы* for*, если они применимы*** В цикле *for* управляющий код находится в одном месте, что способствует созданию легко читаемых циклов. При модификации ПО программисты часто делают ошибку, изменяя код инициализации в начале цикла *while* и забывая исправить соответствующий код в конце цикла. В цикле *for* необходимый код расположен в начале цикла, что упрощает модификацию кода. Если вы можете использовать цикл *for* вместо других циклов, сделайте это.

***Не используйте цикл* for*, если цикл while подходит больше*** Обычным злоупотреблением гибкой структурой цикла *for* в языках C++, C# и Java является размещение частей цикла *while* в заголовке цикла *for*. Взгляните на цикл *while*, втиснутый в заголовок цикла *for*:

**Пример цикла *while*, злостно втиснутого в заголовок цикла *for* (C++)**

```
// Чтение всех записей из файла.
for ( inputFile.MoveToStart(), recordCount = 0; !inputFile.EndOfFile();
   recordCount++ ) {
   inputFile.GetRecord();
}
```

Преимущество цикла *for* в языке C++ по сравнению с другими языками состоит в его большей гибкости по отношению к информации, которую он может использовать для инициализации и завершения. Недостатком такой гибкости является возможность помещения в заголовок цикла выражений, не имеющих ничего общего с управлением циклом.

Зарезервируйте заголовок цикла *for* для выражений, управляющих циклом: выполняющих инициализацию, завершение и движение к завершению. В приведенном примере выражение *inputFile.GetRecord()* в теле цикла продвигает цикл в сторону завершения, а выражения *recordCount* — нет; это вспомогательные выражения, не управляющие циклом. Размещение *recordCount* в заголовке цикла, а *inputFile.GetRecord()* — вне его создает путаницу и фальшивое впечатление, что *recordCount* управляет циклом.

Если в данном случае вы хотите использовать цикл *for*, а не *while*, поместите управляющие выражения в заголовок, а все остальные из него уберите. Вот правильный способ использования заголовка цикла:

**Пример логичного, хоть и нетрадиционного использования заголовка цикла *for* (C++)**

```
recordCount = 0;
for ( inputFile.MoveToStart(); !inputFile.EndOfFile(); inputFile.GetRecord() ) {
   recordCount++;
}
```

Все содержимое заголовка цикла в этом примере относится к управлению циклом. Выражение *inputFile.MoveToStart()* инициализирует цикл, выражение *!inputFile.EndOfFile()* проверяет его завершение, а *inputFile.GetRecord()* продвигает цикл в сторону завершения. Выражения, относящиеся к *recordCount*, не продвигают цикл в сторону завершения напрямую и поэтому вполне уместно не включены в заголовок цикла. Возможно, цикл *while* все же больше подходит для этой работы, но этот код по крайней мере логично использует заголовок цикла. Для галочки покажем, как будет выглядеть этот код при использовании цикла *while*:

**Пример соответствующего использования цикла *while* (C++)**

```
// Чтение всех записей из файла.
inputFile.MoveToStart();
recordCount = 0;
while ( !inputFile.EndOfFile() ) {
   inputFile.GetRecord();
   recordCount++;
}
```

## Обработка середины цикла

Следующие подразделы описывают обработку середины цикла:

***Используйте { и } для обрамления выражений в цикле*** Всегда используйте скобки. Они ничего не стоят в плане скорости или размера во время выполнения, но повышают читабельность и предотвращают ошибки при изменении кода. Это хорошая практика защитного программирования.

***Избегайте пустых циклов*** В C++ и Java возможно создание пустого цикла, все действия которого закодированы в той же строке, что и проверка выхода из цикла. Вот пример:

**Пример пустого цикла (C++)**

```
while ( ( inputChar = dataFile.GetChar() ) != CharType_Eof ) {
   ;
}
```

Здесь тело цикла пустое, потому что само выражение *while* делает две вещи: выполняет циклические действия (*inputChar = dataFile.GetChar()*) и проверяет, завершить ли работу цикла (*inputChar != CharType_Eof*). Цикл будет яснее, если его перекодировать так, чтобы его работа была более очевидной читателю:

**Пример пустого цикла, преобразованного в полноценный цикл (C++)**

```
do {
   inputChar = dataFile.GetChar();
} while ( inputChar != CharType_Eof  );
```

Новый код занимает три полных строки по сравнению с одной строкой и точкой с запятой. Но это допустимо, так как он и выполняет работу для трех строк, а не для одной.

***Располагайте служебные операции либо в начале, либо в конце цикла*** Служебные операции цикла — это выражения вроде *i = i + 1* или *j++*, чье основное назначение не выполнять работу в цикле, а управлять циклом. В этом примере показаны служебные действия, выполняемые в конце цикла:

**Пример служебных выражений, расположенных в конце цикла (C++)**

```
nameCount = 0;
totalLength = 0;
while ( !inputFile.EndOfFile() ) {
   // Выполняем работу цикла
   inputFile >> inputString;
   names[ nameCount ] = inputString;
   ...

   // Готовимся к следующей итерации цикла – служебные действия.
```

Вот служебные операторы.

```
   nameCount++;
   totalLength = totalLength + inputString.length();
}
```

Как правило, переменные, которые вы инициализируете перед циклом, и есть те переменные, которыми вы манипулируете в служебной части цикла.

***Заставьте каждый цикл выполнять только одну функцию*** Простой факт, что цикл может использоваться для выполнения двух дел одновременно, — недостаточное оправдание для их совмещения. Циклы должны быть подобны методам в том плане, что каждый должен делать только одно дело и делать его хорошо. Если использование двух циклов, когда хватит и одного, кажется неэффективным, напишите код в виде двух циклов, прокомментируйте, что их можно объединить для эффективности, и дождитесь, пока тесты оценки производительности покажут проблему в этом месте. Только после этого объединяйте два цикла в один.

**Перекрестная ссылка** Об оптимизации см. главы 25 и 26.

## Завершение цикла

Следующие подразделы описывают обработку конца цикла.

***Убедитесь, что выполнение цикла закончилось*** Это основной принцип. Мысленно моделируйте выполнение цикла до тех пор, пока не будете уверены, что при любых обстоятельствах он завершен. Продумайте номинальные варианты, граничные точки и каждый из исключительных случаев.

***Сделайте условие завершения цикла очевидным*** Если вы используете цикл *for*, не забавляетесь с индексом цикла и не применяете операторы *goto* или *break* для выхода из него, то условие завершения будет очевидным. Аналогично, если вы используете циклы *while* или *repeat-until* и поместили все управление в выражение *while* или *repeat-until*, условие завершения также будет очевидным. Смысл в том, чтобы размещать управление в одном месте.

***Не играйте с индексом цикла* for *для завершения цикла*** Некоторые программисты взламывают значение индекса цикла *for* для более раннего завершения цикла. Вот пример:

**Пример неправильного обращения с индексом цикла (Java)**

```
for ( int i = 0; i < 100; i++ ) {
   // Некоторый код
   ...
   if ( ... ) {
Здесь индекс портится.
      i = 100;
   }

   // Еще код
   ...
}
```

Смысл этого примера в завершении цикла при каком-то условии с помощью установки значения *i* в *100*, что больше, чем границы диапазона цикла *for* от *0* до *99*. Фактически все хорошие программисты избегают такого способа — это при-

знак любительского подхода. Когда вы задаете цикл *for*, манипуляции с его счетчиком должны быть под запретом. Для получения большей управляемости условиями выхода используйте цикл *while*.

***Избегайте писать код, зависящий от последнего значения индекса цикла***
Использование значения индекса цикла после его завершения — дурной тон. Конечное значение индекса меняется от языка к языку и от реализации к реализации. Значения различаются, когда цикл завершается нормально или аномально. Даже если вы не задумываясь можете назвать это конечное значение, следующему читателю кода, возможно, придется о нем задуматься. Более правильным вариантом, к тому же более самодокументируемым, будет присвоение последнего значения какой-либо переменной в подходящем месте внутри цикла.

Этот код некорректно использует конечное значение индекса:

**Пример кода, который неправильно применяет последнее значение индекса цикла (C++)**

```
for ( recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
   if ( entry[ recordCount ] == testValue ) {
      break;
   }
}
// Много кода
...

Здесь неправильное применение завершающего значения индекса цикла.
if ( recordCount < MAX_RECORDS ) {
   return( true );
}
else {
   return( false );
}
```

В этом фрагменте вторая проверка *recordCount* < *MaxRecords* производит впечатление, что цикл будет проходить по всем элементам *entry[]* и вернет *true*, если найдет значение, равное *testValue*, и *false* в противном случае. Тяжело помнить, будет ли индекс инкрементироваться после конца цикла, поэтому легко сделать ошибку потери единицы. Лучше переписать код так, чтобы он не зависел от последнего значения индекса. Вот пример обновленного кода:

**Пример кода, который не делает ошибки при использовании последнего значения индекса цикла (C++)**

```
found = false;
for ( recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
   if ( entry[ recordCount ] == testValue ) {
      found = true;
      break;
   }
}
```

```
// Много кода
...
return( found );
```

Этот второй фрагмент использует дополнительную переменную и располагает обращения к *recordCount* в более ограниченном пространстве. Как часто бывает при применении вспомогательной логической переменной, результирующий код становится яснее.

***Рассмотрите использование счетчиков безопасности*** Счетчик безопасности — это переменная, увеличивающаяся при каждом проходе цикла, чтобы определить, не слишком ли много раз выполняется цикл. Если вы пишете программу, в которой любая ошибка будет катастрофической, вы можете использовать счетчики безопасности, чтобы убедиться, что все циклы заканчиваются. Такой цикл на C++ вполне может использовать счетчик безопасности:

**Пример цикла, который мог бы использовать счетчик безопасности (C++)**

```
do {
   node = node->Next;
   ...
} while ( node->Next != NULL );
```

Вот тот же код с добавленным счетчиком безопасности:

**Пример использования счетчика безопасности (C++)**

```
safetyCounter = 0;
do {
   node = node->Next;
   ...

Здесь код счетчика безопасности.

   safetyCounter++;
   if ( safetyCounter >= SAFETY_LIMIT ) {
      Assert( false, "Internal Error: Safety-Counter Violation." );
   }
   ...
} while ( node->Next != NULL );
```

Счетчики безопасности не панацея. Добавляемые в код по одному, они увеличивают сложность и могут привести к дополнительным ошибкам. Так как они не применяются в каждом цикле, вы можете забыть поддержать код счетчика при модификации циклов в той части программы, где они все же используются. Но если счетчики безопасности вводятся на уровне проектного стандарта для критических циклов, вы будете ожидать их, и код этих счетчиков будет не более подвержен ошибкам, чем любой другой.

### Досрочное завершение цикла

Многие языки предоставляют средства для завершения цикла без выполнения условий *for* или *while*. В данном обсуждении слово *break* обозначает общий тер-

мин для оператора *break* в C++, C и Java; выражений *Exit-Do* и *Exit-For* в Visual Basic и подобных конструкций, включая имитации с помощью *goto*, в языках, не поддерживающих *break* напрямую. Оператор *break* (или его эквивалент) приводит к завершению цикла через нормальный канал выхода. Программа продолжает выполнение с первого оператора, расположенного после цикла.

Оператор *continue* похож на *break* в том смысле, что это вспомогательное средство для управления циклом. Однако вместо выхода из цикла, *continue* заставляет программу пропустить тело цикла и продолжить выполнение со следующей итерации. Оператор *continue* — это сокращенный вариант блока *if-then*, предотвращающего выполнение остальной части цикла.

***Рассмотрите использование операторов* break *вместо логических флагов в цикле* while** Порой добавление логических флагов в цикл *while* с целью имитации выхода из тела цикла усложняет чтение кода. Иногда вы можете убрать несколько уровней отступа в цикле и упростить его управление, просто используя *break* вместо группы проверок *if*. Размещение нескольких отдельных условий *break* рядом с кодом, приводящим к их выполнению, может уменьшить вложенность и сделать цикл читабельнее.

***Остерегайтесь цикла с множеством операторов* break*, разбросанных по всему коду*** Цикл, содержащий большое количество операторов *break*, может сигнализировать о нечетком представлении структуры цикла или его роли в окружающем коде. Рост числа *break* увеличивает вероятность, что цикл может быть более ясно представлен в виде набора нескольких циклов вместо одного цикла с множеством выходов.

Согласно статье в «Software Engineering Notes» программная ошибка, которая 15 января 1990 года на 9 часов вывела из строя телефонную сеть Нью-Йорка, возникла благодаря лишнему оператору *break*.(SEN, 1990):

**Пример ошибочного использования оператора *break* в блоке *do-switch-if* (C++)**

```
do {
   ...
   switch
      ...
      if () {
         ...

Этот break предназначался для if, но вместо этого привел к выходу из switch.

         break;
         ...
      }
      ...
} while ( ... );
```

Большое количество *break* не обязательно означает ошибку, но их присутствие в цикле — тревожный сигнал: как канарейка в шахте, задыхающаяся из-за недостатка воздуха, вместо того чтобы петь.

***Используйте* continue *для проверок в начале цикла*** Хорошим применением оператора *continue* будет перемещение операций в конец тела цикла после проверки некоторого условия в его начале. Например, если цикл читает записи, отбрасывает часть из них, а остальные обрабатывает, вы можете поместить подобную проверку в начало цикла:

**Пример относительно безопасного использования *continue* (псевдокод)**

```
while ( not eof( file ) ) do
   read( record, file )
   if ( record.Type <> targetType ) then
      continue

   - Обрабатываем запись targetType.
   ...
end while
```

Такое использование *continue* позволяет избегать проверок *if*, что эффективно уменьшит отступы внутри всего тела цикла. С другой стороны, если *continue* возникает в середине или конце цикла, используйте вместо него *if*.

***Используйте структуру* break *с метками, если ваш язык ее поддерживает*** Java поддерживает помеченные операторы *break*, что позволяет предотвратить проблемы, приведшие к выходу из строя телефонов в Нью-Йорке. *break* с меткой можно использовать для выхода из цикла *for*, условия *if* или любого блока кода, заключенного в скобки (Arnold, Gosling and Holmes, 2000).

Вот возможное решение «нью-йоркской проблемы», переписанное на Java вместо C++, что позволяет использовать *break* с меткой:

**Пример лучшего использования помеченного оператора *break* в блоке *do-switch-if* (Java)**

```
do {
   ...
   switch
      ...
      CALL_CENTER_DOWN:
      if () {
         ...

Назначение помеченного break однозначно.
         break CALL_CENTER_DOWN;
         ...
      }
      ...
} while ( ... );
```

***Используйте операторы* break *и* continue *очень осторожно*** Применение *break* исключает возможность представления цикла в виде черного ящика. Если вы ограничиваетесь только одним выражением для управления условием выхода из цикла, то получаете мощное средство для упрощения циклов. Применение *break*

заставляет читателя смотреть внутрь цикла, чтобы разобраться в его управлении. Это усложняет понимание цикла.

Используйте *break* только после того, как рассмотрели все альтернативы. Вы не можете сказать с уверенностью, хороши или плохи конструкции *continue* и *break*. Некоторые ученые утверждают, что это допустимые технологии в структурном программировании, а некоторые — что нет. Поскольку вы не знаете, правильно ли применять *continue* и *break* вообще, используйте их, но не забывайте, что вы можете быть неправы. На самом деле это сводится к простому утверждению: если вы не можете аргументировать применение *break* или *continue*, не применяя их.

## Проверка граничных точек

При разработке цикла обычно представляют интерес три точки: первая итерация, случайно выбранная итерация в середине и последняя итерация. Когда вы создаете цикл, мысленно пройдитесь по этим трем точкам и убедитесь, что в цикле нет ошибки потери единицы. Если цикл содержит какие-то специальные случаи, выполнение которых отличается от первой или последней итерации, проверьте их тоже. Если цикл производит сложные вычисления, достаньте свой калькулятор и проверьте их вручную.

Готовность выполнять такой вид проверки — ключевое различие между квалифицированными и неквалифицированными программистами. Первые проделывают мысленное моделирование и вычисления вручную, потому что знают, что эти меры помогут им найти ошибки.

Вторые имеют склонность к случайному экспериментированию, пока не найдут правдоподобную комбинацию. Если цикл не работает так, как предполагалось, неумелый программист меняет знак < на <=. Если и это не помогает, он исправляет индекс цикла, добавляя или вычитая 1. В конечном счете таким способом программист может нащупать правильную комбинацию или просто заменить изначальную ошибку более незаметной. Даже если этот случайный процесс приведет к правильной программе, программист не будет знать, почему она работает корректно.

Мысленное моделирование и ручные вычисления могут дать несколько преимуществ. Умственная тренировка приводит к меньшему количеству ошибок при первоначальном кодировании, более быстрому обнаружению проблем при отладке и в целом более полному пониманию программы. Умственные упражнения означают, что вы знаете, как работает код, а не просто предполагаете это.

## Использование переменных цикла

Далее описаны некоторые принципы применения переменных цикла.

**Перекрестная ссылка** Об именовании переменных цикла см. подраздел «Именование индексов циклов» раздела 11.2.

***Используйте порядковые или перечислимые типы для границ массивов и циклов*** Обычно счетчики циклов должны быть целыми значениями. Числа с плавающей запятой плохо инкрементируются. Например, вы можете прибавить 1,0 к 26 742 897,0 и получить 26 742 897,0 вместо 26 742 898,0. Если это число используется как индекс цикла, вы получите бесконечный цикл.

***Используйте смысловые имена переменных, чтобы сделать вложенные циклы читабельными*** Массивы часто индексируются с помощью тех же переменных, что используются как индексы цикла. Если у вас одномерный массив, то вы еще сможете выйти сухим их воды, применяя *i*, *j* или *k* для его индексации. Но если у массива два и более измерений, вам следует задавать значимые имена для индексов, чтобы прояснить свои действия. Смысловые имена индексов массивов одновременно уточняют и назначение цикла, и элемент массива, к которому вы планируете обратиться.

Вот пример кода, который не применяет этот принцип: в нем использованы бессмысленные имена *i*, *j* и *k*:

**Пример неправильных имен переменных цикла (Java)**

```
for ( int i = 0; i < numPayCodes; i++ ) {
   for ( int j = 0; j < 12; j++ ) {
      for ( int k = 0; k < numDivisions; k++ ) {
         sum = sum + transaction[ j ][ i ][ k ];
      }
   }
}
```

Как вы думаете, что означают индексы в элементе *transaction*? Сообщают ли переменные *i*, *j* и *k* что-либо о содержимом *transaction*? Если вы знаете объявление *transaction*, можете ли вы легко определить, указаны ли индексы в правильном порядке? Вот тот же цикл с более читабельными именами переменных:

**Пример хороших имен переменных цикла на Java**

```
for ( int payCodeIdx = 0; payCodeIdx < numPayCodes; payCodeIdx++ ) {
   for (int month = 0; month < 12; month++ ) {
      for ( int divisionIdx = 0; divisionIdx < numDivisions; divisionIdx++ ) {
         sum = sum + transaction[ month ][ payCodeIdx ][ divisionIdx ];
      }
   }
}
```

Как вы думаете, что означают индексы в элементе *transaction* на этот раз? В этом случае ответ получить проще, потому что имена переменных *payCodeIdx, month* и *divisionIdx* гораздо красноречивее, чем *i*, *j* и *k*. Компьютер с одинаковой легкостью прочитает обе версии цикла. Однако людям легче будет читать вторую версию, чем первую, поэтому второй вариант лучше, поскольку ваша основная аудитория состоит из людей, а не из компьютеров.

***Используйте смысловые имена во избежание пересечения индексов*** Привычное использование переменных *i*, *j* и *k* приводит к увеличению риска пересечения индексов — использованию одного и того же имени индекса для разных целей. Взгляните:

**Пример пересечения индексов (C++)**

```
i сначала используется здесь...
for ( i = 0;  i < numPayCodes; i++ ) {
   // много кода
   ...
   for ( j = 0; j < 12; j++ ) {
      // много кода
      ...
...а теперь здесь
      for ( i = 0; i < numDivisions; i++ ) {
         sum = sum + transaction[ j ][ i ][ k ];
      }
   }
}
```

Применение *i* настолько привычно, что эта переменная используется в одной вложенной структуре дважды. Второй цикл *for*, управляемый *i*, конфликтует с первым — это и есть пересечение индексов. Применение более значимых имен, чем *i*, *j* и *k,* предотвратило бы проблему. Вообще, если тело цикла содержит больше пары строк кода, или может вырасти, или входит в группу вложенных циклов, избегайте переменных *i*, *j* и *k*.

***Ограничивайте видимость переменных-индексов цикла самим циклом*** Пересечение индексов цикла и другое применение индексов вне самих циклов — настолько важная проблема, что разработчики языка Ada решили сделать индексы цикла *for* недоступными вне цикла. Попытка использования переменной-индекса вне цикла *for* приводит к ошибке времени компиляции.

C++ и Java в какой-то мере реализуют ту же идею — они позволяют объявлять индексы цикла в нем самом, но не требуют этого. Выше, в примере раздела «Избегайте писать код, зависящий от последнего значения индекса цикла», переменная *recordCount* может быть объявлена внутри выражения *for*, что ограничит ее область видимости этим циклом:

**Пример объявления переменной-индекса цикла внутри цикла *for* (C++)**

```
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
   // Циклический код, использующий recordCount.
}
```

В принципе эта методика должна позволять создавать код, повторно объявляющий переменную *recordCount* в нескольких циклах без риска неправильного использования двух разных *recordCount*. Такое применение позволило бы писать, например, такой код:

**Пример объявления переменных-индексов внутри циклов *for* и их (возможно!) безопасное повторное использование (C++)**

```
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
   // Циклический код, использующий recordCount.
}
```

```
// Промежуточный код.
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
    // Дополнительный циклический код, использующий другую переменную recordCount.
}
```

Такая методика полезна для документирования назначения переменной *recordCount*. Однако не полагайтесь на ваш компилятор в вопросе области видимости *recordCount*. В разделе 6.3.3.1 книги «The C++ Programming Language» (Stroustrup, 1997) говорится, что переменная *recordCount* должна иметь область видимости, ограниченную ее циклом. Но, проверив эту функциональность в трех разных компиляторах C++, я получил три разных результата:

- первый компилятор сигнализировал о повторном объявлении переменной *recordCount* во втором цикле *for* и сгенерировал ошибку;
- второй компилятор допустил объявление переменной *recordCount* во втором цикле *for*, но разрешил ее использование вне первого цикла *for*;
- третий компилятор разрешил оба объявления переменных *recordCount* и не допустил использования ни одной из них за пределами циклов, где они объявлялись.

Как это часто бывает с наиболее эзотерическими свойствами языка, реализации компиляторов могут различаться.

## Насколько длинным может быть цикл?

Длина цикла может измеряться в строках кода или глубине вложенности.

***Делайте циклы достаточно короткими, чтобы их можно было увидеть сразу целиком*** Если вы обычно смотрите на циклы на вашем мониторе, а ваш монитор показывает 50 строк, то установите 50-строчное ограничение длины. Эксперты предложили ограничивать длину цикла одной страницей. Однако когда вы оцените преимущество создания простого кода, вы редко будете писать циклы длиннее 15 или 20 строк.

***Ограничивайте вложенность тремя уровнями*** Исследования показали, что способность программистов разобраться в цикле существенно снижается, если уровень вложенности превышает три уровня (Yourdon, 1986a). Если вам нужно большее число уровней, сделайте цикл короче (концептуально), вынеся его часть в отдельный метод или упростив управляющую структуру.

**Перекрестная ссылка** Об упрощении вложенности см. раздел 19.4.

***Выделяйте внутреннюю часть длинных циклов в отдельные методы*** Если цикл хорошо спроектирован, то код внутри него часто можно выделить в один или несколько методов, которые будут вызываться из цикла.

***Делайте длинные циклы особенно ясными*** Длина увеличивает сложность. Если вы пишете короткий цикл, вы можете использовать более рискованные управляющие структуры, такие как *break* и *continue*, множественные выходы, сложные условия завершения и т.д. Если вы пишете более длинный цикл и проявляете хоть какую-то заботу о читателях, вы предусмотрите в цикле только один выход и сделаете условие выхода исключительно понятным.

# 16.3. Простое создание цикла — изнутри наружу

Если у вас иногда возникают затруднения при кодировании сложного цикла (что бывает у большинства программистов), есть простой способ реализовать его с первого раза. Вот как это сделать. Начните с одного действия. Закодируйте его с помощью констант. Затем сделайте отступ, окружите его циклом и замените константы индексами цикла или вычисляемыми выражениями. Добавьте еще один цикл, если он нужен, и замените другие константы. Повторите процесс нужное число раз. После этого добавьте код инициализации. Так как вы начали с одного действия и двигались в сторону его обобщения, то можете рассматривать этот процесс как кодирование изнутри наружу.

**Перекрестная ссылка** Кодирование цикла изнутри наружу похоже на ППП (см. главу 9).

Допустим, вы разрабатываете программу для страховой компании. Ставки для страхования жизни варьируются в зависимости от возраста и пола страхователя. Ваша задача — написать метод, вычисляющий общую страховую премию для группы лиц. Вам нужен цикл, который будет брать ставку для каждого человека из списка и добавлять ее к общей сумме. Вот что нужно сделать.

Во-первых, в комментариях напишите шаги, которые должно выполнять тело цикла. Легче записать, что необходимо сделать, когда вы не думаете о деталях синтаксиса, индексах цикла, массива и т. п.

### Шаг 1: Создание цикла изнутри наружу (псевдокод)

```
- Получить ставку из таблицы.
- Добавить ставку к общей сумме.
```

Во-вторых, замените комментарии в теле цикла на код, насколько это возможно без фактического написания всего цикла. В данном случае возьмите ставку для одного лица и добавьте ее к сумме. Используйте реальные данные, а не абстракции.

### Шаг 2: Создание цикла изнутри наружу (псевдокод)

```
table еще не использует индексов.
rate = table[ ]
totalRate = totalRate + rate
```

Пример предполагает, что *table* — это массив, содержащий данные о ставках. Сначала вам не надо беспокоиться об индексах массива. *rate* — это переменная, в которой хранится ставка, выбранная из таблицы ставок. Соответственно *totalRate* — переменная, содержащая сумму всех ставок.

Далее добавьте индексы к массиву *table*:

### Шаг 3: Создание цикла изнутри наружу (псевдокод)

```
rate = table[ census.Age ][ census.Gender ]
totalRate = totalRate + rate
```

Доступ к элементам массива осуществляется в зависимости от возраста и пола, поэтому *census.Age* и *census.Gender* служат для индексации массива. Пример пред-

полагает, что *census* — это структура, содержащая сведения о людях из рассчитываемой группы.

Следующий шаг — построение цикла вокруг существующих выражений. Поскольку цикл должен вычислять ставки для каждого человека из группы, индекс должен перечислять всех членов группы.

#### Шаг 4: Создание цикла изнутри наружу (псевдокод)

```
For person = firstPerson to lastPerson
   rate = table[ census.Age, census.Gender ]
   totalRate = totalRate + rate
End For
```

Все, что вы должны сделать, — это поместить цикл *for* вокруг существующего кода и добавить к нему пару *begin-end*. Напоследок убедитесь, что переменные, использующие индекс цикла *person*, написаны правильно. В данном случае переменная *census* изменяется вместе с *person*, поэтому ее следует корректно проиндексировать.

#### Шаг 5: Создание цикла изнутри наружу (псевдокод)

```
For person = firstPerson to lastPerson
   rate = table[ census[ person ].Age, census[ person ].Gender ]
   totalRate = totalRate + rate
End For
```

И, наконец, напишите необходимую инициализацию. В этом примере нужно инициализировать переменную *totalRate*.

#### Последний шаг: Создание цикла изнутри наружу (псевдокод)

```
totalRate = 0
For person = firstPerson to lastPerson
   rate = table[ census[ person ].Age, census[ person ].Gender ]
   totalRate = totalRate + rate
End For
```

Если вы хотите добавить еще один цикл вокруг цикла *person*, продолжайте таким же образом. Вы не должны жестко придерживаться этого порядка. Идея в том, чтобы начать с чего-то определенного, думать только об одной задаче в каждый момент времени и строить цикл из простых компонентов. Предпринимайте маленькие, понятные шаги, постепенно обобщая и усложняя цикл. Таким образом, вы минимизируете количество кода, на котором необходимо одновременно сосредоточиваться и, следовательно, уменьшите вероятность ошибки.

## 16.4. Соответствие между циклами и массивами

Циклы и массивы часто связаны друг с другом. Зачастую цикл создается для манипуляций с массивами, и счетчики цикла один к одному соответствуют индексам массива. Так, следующие индексы циклов *for* соответствуют индексам массива:

**Перекрестная ссылка** О соответствии между циклами и массивами см. также раздел 10.7.

### Пример умножения массивов (Java)

```
for ( int row = 0; row < maxRows; row++ ) {
   for ( int column = 0; column < maxCols; column++ ) {
      product[ row ][ column ] = a[ row ][ column ] * b[ row ][ column ];
   }
}
```

В языке Java цикл для таких операций с массивами необходим. Но стоит заметить, что циклические структуры и массивы не обязательно должны использоваться вместе. Некоторые языки, особенно APL и Fortran 90 и более поздние, предоставляют операции с массивами, исключающие необходимость применять такие циклы, как только что продемонстрированные. Вот так выглядит фрагмент кода на APL, выполняющий ту же операцию:

### Пример умножения массивов (APL)

```
product <- a x b
```

Вариант на APL проще и менее подвержен ошибкам. Он использует только три операнда, тогда как фрагмент на Java — 17. Он не содержит переменных цикла, индексов массива или управляющих структур, которые можно некорректно закодировать.

Из этих примеров следует, что частично программирование направлено на решение задачи, а частично — на решение этой задачи на определенном языке. Выбранный вами язык существенно влияет на получаемый результат.

http://cc2e.com/1616

### Контрольный список: циклы

**Выбор и создание цикла**

- ❑ Используется ли цикл *while* вместо цикла *for*, если он больше подходит?
- ❑ Создавался ли цикл изнутри наружу?

**Вход в цикл**

- ❑ Выполняется ли вход в цикл сверху?
- ❑ Расположен ли код инициализации непосредственно перед циклом?
- ❑ Если необходим бесконечный или событийный цикл, конструируется ли он явно, или сделан такой ляп, как *for i = 1 to 9999*?
- ❑ В цикле *for* в C++, C или Java резервируется ли заголовок цикла только для управляющего кода?

**Тело цикла**

- ❑ Использует ли цикл скобки *{* и *}* или их эквиваленты для обрамления тела цикла и предотвращения проблем, связанных с неправильной модификацией?
- ❑ Содержит ли тело цикла хоть что-то? Не пустое ли оно?
- ❑ Сгруппированы ли служебные операции в начале или конце цикла?
- ❑ Выполняет ли цикл одну и только одну функцию, как это делает хорошо спроектированный метод?
- ❑ Достаточно ли цикл короткий, чтобы его можно было сразу увидеть целиком?
- ❑ Не превышает ли вложенность цикла трех уровней?

- ❑ Вынесено ли содержимое длинного цикла в отдельный метод?
- ❑ Если цикл достаточно длинный, написан ли он особенно ясно?

**Индексы цикла**

- ❑ Если это цикл *for*, не выполняются ли манипуляции с индексом цикла внутри самого цикла?
- ❑ Используется ли для сохранения важных значений индекса цикла вне этого цикла специальная переменная, а не сам индекс цикла?
- ❑ Является ли индекс цикла порядковым или перечислимым типом, но не типом с плавающей запятой?
- ❑ Имеет ли индекс цикла смысловое имя?
- ❑ Не содержит ли цикл пересечения индексов?

**Завершение цикла**

- ❑ Завершается ли цикл при всех возможных условиях?
- ❑ Использует ли цикл счетчики безопасности, если они приняты на уровне стандарта?
- ❑ Очевидно ли условие завершения цикла?
- ❑ Если используются операторы *break* или *continue*, корректны ли они?

# Ключевые моменты

- Циклы сложны для понимания. Сохраняя их простыми, вы помогаете читателям вашего кода.
- К способам упрощения циклов относятся: избегание экзотических видов циклов, минимизация вложенности, создание очевидных входов и выходов цикла и хранение служебного кода в одном месте.
- Индексы цикла часто употребляются неправильно. Называйте их понятно и используйте только с одной целью.
- Аккуратно продумайте весь цикл, чтобы убедиться, что он работает правильно во всех случаях и завершается при любых возможных обстоятельствах.

ГЛАВА 17

# Нестандартные управляющие структуры

http://cc2e.com/1778

**Содержание**



**Связанные темы**

- Общие вопросы управления: глава 19
- Последовательный код: глава 14
- Код с условными операторами: глава 15
- Код с циклами: глава 16
- Обработка исключений: раздел 8.4

Несколько управляющих структур существует в сумрачной зоне между передовым краем технологии и полной дискредитацией и несостоятельностью, и часто в одно и то же время! Эти конструкции доступны не во всех языках, но там, где они есть, они могут быть полезны при аккуратном применении.

## 17.1. Множественные возвраты из метода

Большинство языков поддерживает некоторые способы возврата управления после частичного выполнения метода. Операторы *return* и *exit* — управляющие структуры, которые позволяют программе при желании завершить работу метода. В результате функция завершается через нормальный канал выхода, возвращая управление вызывающему методу. Слово *return* здесь используется как общий термин, обозначающий *return* в C++ и Java, *Exit Sub* и *Exit Function* в Visual Basic и аналогичные конструкции. Далее перечислены некоторые принципы использования оператора *return*.

***Используйте return, если это повышает читабельность*** В некоторых методах при получении ответа хочется сразу вернуть управление вызывающей стороне. Если метод определен так, что обнаружение ошибки не требует никакой дополнительной очистки ресурсов, то отсутствие немедленного возврата означает необходимость писать лишний код.

Вот хороший пример ситуации, когда возврат из нескольких частей метода имеет смысл:

**Пример правильного множественного возврата из метода (C++)**

Этот метод возвращает перечислимый тип *Comparison*.

```
Comparison Compare( int value1, int value2 ) {
   if ( value1 < value2 ) {
      return Comparison_LessThan;
   }
   else if ( value1 > value2 ) {
      return Comparison_GreaterThan;
   }
   return Comparison_Equal;
}
```

Другие примеры не настолько однозначны, что будет проиллюстрировано ниже.

***Упрощайте сложную обработку ошибок с помощью сторожевых операторов (досрочных* return *или* exit*)*** Если программа вынуждена проверять большое количество ошибочных ситуаций перед выполнением номинальных действий, это может привести к коду очень большой вложенности и замаскировать номинальный вариант. Вот пример такого кода:

**Код, скрывающий номинальный вариант (Visual Basic)**

```
If file.validName() Then
   If file.Open() Then
      If encryptionKey.valid() Then
         If file.Decrypt( encryptionKey ) Then
```

Здесь код номинального варианта.

```
            ' Много кода.
            ...
         End If
      End If
   End If
End If
```

Отступ основного кода метода внутри четырех условий *if* выглядит неэстетично, особенно если этот код в самом внутреннем блоке *if* состоит из множества строк. В таких случаях часто можно упростить логику, если все ошибочные ситуации проверять сначала, расчистив дорогу для номинального хода алгоритма. Вот как это может выглядеть:

**Простой код, использующий сторожевые операторы для прояснения номинального варианта (Visual Basic)**

```
' Выполняем инициализацию. При обнаружении ошибок завершаем работу.
If Not file.validName() Then Exit Sub
If Not file.Open() Then Exit Sub
If Not encryptionKey.valid() Then Exit Sub
If Not file.Decrypt( encryptionKey ) Then Exit Sub

' Много кода.
...
```

В таком простом примере описанный способ выглядит аккуратным решением, но промышленный код при обнаружении ошибки часто требует большего количества служебных операций или действий по очистке ресурсов. Вот более реалистичный пример:

**Более реалистичный код, использующий сторожевые операторы для прояснения номинального варианта (Visual Basic)**

```
' Выполняем инициализацию. При обнаружении ошибок завершаем работу.
If Not file.validName() Then
    errorStatus = FileError_InvalidFileName
   Exit Sub
End If

If Not file.Open() Then
    errorStatus = FileError_CantOpenFile
   Exit Sub
End If

If Not encryptionKey.valid() Then
    errorStatus = FileError_InvalidEncryptionKey
   Exit Sub
End If

If Not file.Decrypt( encryptionKey ) Then
    errorStatus = FileError_CantDecryptFile
   Exit Sub
End If

Здесь код номинального варианта.

' Много кода.
...
```

В коде промышленного масштаба использование *Exit Sub* приводит к написанию довольно большого количества кода до обработки номинального варианта. Однако *Exit Sub* позволяет избежать глубокой вложенности, присущей первому примеру, и если код первого примера расширить с целью установки значений переменной *errorStatus*, то вариант с *Exit Sub* покажется лучшим с точки зрения группировки взаимосвязанных выражений. Когда вся пыль осядет, подход с *Exit Sub* покажется более удобным для чтения и сопровождения, и за небольшую цену.

***Минимизируйте число возвратов из каждого метода*** Тяжело понять логику метода, если при чтении его последних строк вы не подозреваете о возможности выхода из него где-то вверху. По этой причине используйте операторы возврата благоразумно — только если они улучшают читабельность.

# 17.2. Рекурсия

При рекурсии метод решает небольшую часть задачи, разбивает задачу на меньшие порции и вызывает сам себя для решения каждой из этих порций. Обычно рекурсию применяют, когда небольшую часть задачи легко решить, а саму задачу просто разложить на составные части.

Рекурсия не часто бывает необходима, но при аккуратном использовании она позволяет создавать элегантные решения, как в этом примере, где алгоритм сортировки иллюстрирует отличное применение рекурсии:

**Пример алгоритма сортировки, использующего рекурсию (Java)**

```
void QuickSort( int firstIndex, int lastIndex, String [] names ) {
   if ( lastIndex > firstIndex ) {
      int midPoint = Partition( firstIndex, lastIndex, names );

Здесь выполняются рекурсивные вызовы.

      QuickSort( firstIndex, midPoint-1, names );
      QuickSort( midPoint+1, lastIndex, names )
   }
}
```

В этом фрагменте алгоритм сортировки разрезает массив на две части и затем вызывает сам себя для сортировки каждой половины массива. Когда ему будет передан участок массива, слишком короткий для сортировки, т. е. когда *( lastIndex <= firstIndex )*, он перестанет вызывать сам себя.

Для малой группы задач рекурсия позволяет создать простые, элегантные решения. Для несколько большей группы задач она позволяет создать простые, элегантные, трудные для понимания решения. Для большинства задач она создает исключительно запутанные решения — в таких случаях использование простых итераций обычно более понятно. Поэтому применяйте рекурсию выборочно.

## Примеры рекурсии

Допустим, у вас есть тип данных, представляющий лабиринт. Лабиринт — это обычно некая сетка, в узлах которой вы можете повернуть направо, налево, переместиться вверх или вниз. Часто существует возможность двигаться в нескольких направлениях.

Как вы будете разрабатывать программу для поиска пути через лабиринт (рис. 17-1)? Если вы используете рекурсию, ответ довольно прост. Вы начинаете от входа и пробуете все возможные повороты, пока не найдете выхода. Попадая в точку в первый раз, вы пробуете повернуть налево, если это невозможно, то пробуете пойти вверх или вниз. В конце концов вы пытаетесь пойти направо. Вам не надо боять-

ся заблудиться, потому что на каждом перекрестке вы оставляете несколько хлебных крошек и не поворачиваете в одну и ту же сторону дважды.

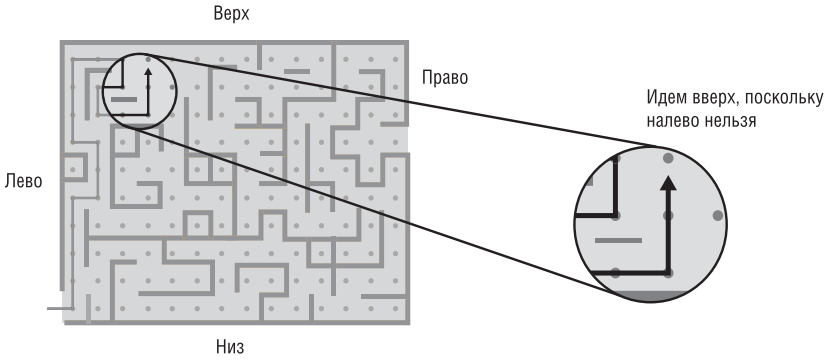



***Рис. 17-1.*** *Рекурсия может быть мощным оружием в борьбе со сложностью, если используется по назначению*

Рекурсивный код может выглядеть, например, так:

### Пример рекурсивного перемещения по лабиринту (C++)

```
bool FindPathThroughMaze( Maze maze, Point position ) {
   // Если это место уже исследовано, не надо снова его проверять.
   if ( AlreadyTried( maze, position ) ) {
      return false;
   }

   // Если это место и есть выход, объявляем успешное завершение.
   if ( ThisIsTheExit( maze, position ) ) {
      return true;
   }

   // Запоминаем, что это место исследовано.
   RememberPosition( maze, position );

   // Проверяем пути налево, вверх, направо, вниз;
   // если один из путей приводит к успеху, прекращаем поиск.
   if ( MoveLeft( maze, position, &newPosition ) ) {
      if ( FindPathThroughMaze( maze, newPosition ) ) {
         return true;
      }
   }

   if ( MoveUp( maze, position, &newPosition ) ) {
      if ( FindPathThroughMaze( maze, newPosition ) ) {
         return true;
      }
   }
```

```
   if ( MoveDown( maze, position, &newPosition ) ) {
      if ( FindPathThroughMaze( maze, newPosition ) ) {
         return true;
      }
   }

   if ( MoveRight( maze, position, &newPosition ) ) {
      if ( FindPathThroughMaze( maze, newPosition ) ) {
         return true;
      }
   }
   return false;
}
```

Первая строка кода проверяет, исследована ли уже данная точка. Одна из ключевых задач рекурсивного метода — предотвращение бесконечной рекурсии. В данном случае, если вы не будете проверять, что эта развилка уже исследовалась, вы можете бесконечно обследовать ее.

Второе выражение проверяет, не является ли эта позиция выходом из лабиринта. Если *ThisIsTheExit()* возвращает *true*, метод тоже возвращает *true*.

Третье выражение запоминает, что вы посетили данную точку. Это предотвращает бесконечную рекурсию, которая может возникнуть в результате замкнутого пути.

Остальные строки пытаются найти выход при движении налево, вверх, вниз и направо. Рекурсия прекратится, если метод когда-нибудь вернет *true*, т. е. будет найден выход из лабиринта.

Логика, используемая в этом примере, довольно прямолинейна. Большинство людей испытывают некоторый дискомфорт при виде рекурсивных методов, ссылающихся сами на себя. Однако в данном случае альтернативное решение было бы гораздо более трудоемким, и поэтому рекурсия отлично подходит.

## Советы по использованию рекурсии

При применении рекурсии имейте в виду эти советы.

***Убедитесь, что рекурсия остановится*** Проверьте метод, чтобы убедиться, что он включает нерекурсивный путь. Обычно это значит, что в методе присутствует проверка, останавливающая дальнейшую рекурсию, когда в ней отпадает необходимость. В примере с лабиринтом условия для *AlreadyTried()* и *ThisIsTheExit()* гарантируют, что рекурсия остановится.

***Предотвращайте бесконечную рекурсию с помощью счетчиков безопасности*** Если вы применяете рекурсию в ситуации, не позволяющей сделать простую проверку вроде описанной выше, добавьте счетчики безопасности, дабы избежать бесконечной рекурсии. Счетчиком должна быть такая переменная, которая не будет создаваться при каждом вызове метода. Используйте переменную-член класса или передавайте счетчик безопасности в виде параметра. Приведем пример:

Рекурсивная процедура должна иметь возможность изменять значение *safetyCounter*, поэтому в Visual Basic она объявляется как *ByRef*-параметр.

### Пример использования счетчика безопасности для предотвращения бесконечной рекурсии (Visual Basic)

```
Public Sub RecursiveProc( ByRef safetyCounter As Integer )
   If ( safetyCounter > SAFETY_LIMIT ) Then
      Exit Sub
   End If
   safetyCounter = safetyCounter + 1
   ...
   RecursiveProc( safetyCounter )
End Sub
```

В данном случае, если вложенность превысит предел счетчика безопасности, рекурсия прекратится.

Если вы не хотите передавать счетчик безопасности в виде отдельного параметра, можно использовать классовую переменную в C++, Java, Visual Basic или соответствующий эквивалент в других языках.

***Ограничьте рекурсию одним методом*** Циклическая рекурсия (A вызывает B вызывает C вызывает A) представляет опасность, потому что ее сложно обнаружить. Осмысление рекурсии в одном методе — достаточно трудоемкое занятие, а понимание рекурсии, охватывающей несколько методов, — это уже слишком. Если возникла циклическая рекурсия, обычно можно так перепроектировать методы, что рекурсия будет ограничена одним из них. Если это не получается, а вы все равно считаете, что рекурсия — это лучший подход, используйте счетчики безопасности в качестве страховки.

***Следите за стеком*** При использовании рекурсии вы точно не знаете, сколько места в стеке будет занимать ваша программа. Кроме того, тяжело заранее предсказать, как будет вести себя программа во время выполнения. Однако вы можете предпринять некоторые усилия для контроля ее поведения.

Во-первых, если вы добавили счетчик безопасности, одним из принципов установки его предела является возможный размер используемого стека. Сделайте этот предел достаточно низким, чтобы предотвратить переполнение стека.

Во-вторых, следите за выделением памяти для локальных переменных в рекурсивных функциях, особенно если эти переменные достаточно объемны. Иначе говоря, лучше использовать *new* для создания объектов в куче, чем позволять компилятору генерировать автоматические объекты в стеке.

***Не используйте рекурсию для факториалов и чисел Фибоначчи*** Одна из проблем с учебниками по вычислительной технике в том, что они предлагают глупые примеры рекурсии. Типичными примерами являются вычисление факториала или последовательности Фибоначчи. Рекурсия — мощный инструмент, и очень глупо использовать ее в этих двух случаях. Если бы программист, работающий у меня, применял рекурсию для вычисления факториала, я бы нанял кого-то другого. Вот рекурсивная версия вычисления факториала:

**Пример неправильного решения: вычисления факториала с помощью рекурсии (Java)**

```
int Factorial( int number ) {
   if ( number == 1 ) {
      return 1;
   }
   else {
      return number * Factorial( number - 1 );
   }
}
```

Не считая медленного выполнения и непредсказуемого использования памяти, рекурсивный вариант функции трудней для понимания, чем итеративный вариант:

**Пример правильного решения: использование итераций для вычисления факториала (Java:)**

```
int Factorial( int number ) {
   int intermediateResult = 1;
   for ( int factor = 2; factor <= number; factor++ ) {
      intermediateResult = intermediateResult * factor;
   }
   return intermediateResult;
}
```

Из этого примера можно усвоить три урока. Первый: учебники по ВычТеху не оказывают миру услугу своими примерами рекурсии. Второй, и более важный: рекурсия — гораздо более мощный инструмент, чем можно предположить из сбивающих с толку примеров расчета факториала и чисел Фибоначчи. Третий — самый важный: вы должны рассмотреть альтернативные варианты перед использованием рекурсии. Иногда один способ работает лучше, иногда — другой. Но прежде чем выбрать какой-то один, надо рассмотреть оба.

# 17.3. Оператор *goto*

Вы могли думать, что дебаты вокруг *goto* утихли, но короткая экскурсия по современным репозиториям исходного кода, таким как *SourceForge.net*, показывает, что *goto* все еще живздоров и глубоко укоренился на сервере вашей компании. Более того, современные эквиваленты обсуждения *goto* до сих пор возникают под разными личинами, включая дебаты о множественных возвратах из методов, множественных выходах из цикла, именованных выходах из цикла, обработке ошибок и исключений.

http://cc2e.com/1785

## Аргументы против *goto*

Основной аргумент против *goto* состоит в том, что код без *goto* — более качественный. Знаменитое письмо Дейкстры «Go To Statement Considered Harmful» («Обоснование пагубности оператора go to») в мартовском номере «Communications of the ACM» 1968 г. положило начало дискуссии. Дейкстра отметил, что качество кода

обратно пропорционально количеству *goto*, использованных программистом. В последующих работах Дейкстра утверждал, что корректность кода, не содержащего *goto*, доказать легче.

Код с операторами *goto* трудно форматировать. Для демонстрации логической структуры используются отступы, а *goto* влияет на логическую структуру. Однако использовать отступы, чтобы показать логику *goto* и места его перехода, сложно или даже невозможно.

Применение *goto* препятствует оптимизации, выполняемой компилятором. Некоторые виды оптимизации зависят от порядка выполнения нескольких выражений подряд. Безусловный переход *goto* усложняет анализ кода и уменьшает возможность оптимизации кода компилятором. Таким образом, даже если применение *goto* увеличивает эффективность на уровне исходного кода, суммарный эффект из-за невозможности оптимизации может уменьшиться.

Сторонники операторов *goto* иногда приводят довод, что они делают программу быстрее и проще. Но код, содержащий *goto*, обычно не самый быстрый и короткий из всех возможных. Изумительная классическая статья Дональда Кнута «Structured Programming with go to Statements» («Структурное программирование и операторы go to») содержит несколько примеров, в которых применение *goto* приводит к более медленному и объемному коду (Knuth, 1974).

На практике применение операторов *goto* приводит к нарушению принципа, что нормальный ход алгоритма должен быть строго сверху вниз. Даже если *goto* при аккуратном использовании не сбивают с толку, как только они появляются, они начинают распространяться по коду, как термиты по разрушающемуся дому. Если разрешен хотя бы один *goto*, вместе с пользой в код проникает и вред, так что лучше вообще запретить использование этого оператора.

В целом опыт двух десятилетий, прошедших с публикации письма Дейкстры показал всю недальновидность создания кода, перегруженного операторами *goto*. В своем обзоре литературы Бен Шнейдерман (Ben Shneiderman) сделал вывод, что факты свидетельствуют в пользу Дейкстры и нам лучше обходиться без *goto* (1980), а многие современные языки, включая Java, даже не содержат такой оператор.

## Аргументы в защиту *goto*

Сторонники *goto* ратуют за осторожное применение оператора при определенных обстоятельствах, а не за неразборчивое употребление. Большинство аргументов против *goto* говорит именно о неразборчивом его использовании. Дискуссия о *goto* вспыхнула, когда Fortran был наиболее популярным языком. Fortran не имел приличных циклов, и в отсутствие хорошего совета по поводу создания цикла с помощью *goto* программисты написали кучу спагетти-кода. Такой код, несомненно, коррелировал с выпуском низкокачественных программ, но это имело отдаленное отношение к аккуратному использованию *goto*, позволяющему заполнить пробел в возможностях, предоставляемых современными языками программирования.

Правильно расположенный *goto* способен помочь избавиться от дублирования кода. Такой код создает проблемы, если две его части модифицируются по-разному. Дублированный код увеличивает размер исходного и выполняемого файлов. Отри-

цательный эффект применения *goto* перевешивается недостатками дублированного кода.

Оператор *goto* может пригодиться в методе, который сначала распределяет ресурсы, выполняет с ними какие-то операции, а потом освобождает эти ресурсы. Используя *goto*, вы можете выполнять очистку в одном месте. Оператор *goto* уменьшает вероятность того, что вы забудете освободить ресурсы при обнаружении ошибки.

**Перекрестная ссылка** О применении операторов *goto* в коде, использующем ресурсы, см. ниже подраздел «Обработка ошибок и операторы *goto*». Об исключениях см. также раздел 8.4.

Порой *goto* позволяет создать более быстрый и короткий код. Вышеупомянутая статья Кнута 1974 года рассматривает несколько вариантов, в которых *goto* дает ощутимое преимущество.

Хорошее программирование не означает исключение всех *goto*. Систематическая декомпозиция, усовершенствование и разумный выбор управляющих структур обычно автоматически приводит к программам, не содержащим *goto*. Стремление к коду без *goto* — это не цель, а результат, и бесполезно заострять внимание исключительно на устранении *goto*.

Десятилетия исследований операторов *goto* не смогли продемонстрировать их вредоносность. В обзоре литературы Б.А. Шейл (B.A. Sheil) сделал вывод, что нереалистичные тестовые условия, плохой анализ данных и неубедительные результаты не подкрепляют заявления Шнейдермана и др., что число ошибок в коде пропорционально количеству *goto* (1981). Шейл не зашел так далеко, чтобы утверждать, что использование *goto* — хорошая идея, он лишь показал, что экспериментальные данные против этих операторов неубедительны.

> Факты свидетельствуют лишь о том, что намеренно хаотичная управляющая структура ухудшает производительность [программиста]. Эти эксперименты не предоставили практически никакого доказательства полезного эффекта какого-то конкретного способа структурирования управляющей логики.
>
> *Б.А. Шейл*

И, наконец, операторы *goto* входят во множество современных языков, включая Visual Basic, C++ и Ada — наиболее тщательно продуманный язык программирования в истории. Ada создавался уже после того, как были приведены все аргументы с обеих сторон дискуссии по *goto*, и после всестороннего рассмотрения вопроса разработчики Ada решили включить в него *goto*.

## Воображаемая дискуссия по поводу *goto*

Отличительная особенность большинства обсуждений *goto* — поверхностность. Спорщик, утверждающий, что «*goto* — это зло», приводит тривиальный фрагмент кода, содержащий операторы *goto*, а затем показывает, как легко его можно переписать без *goto*. Это доказывает главным образом то, что тривиальный код можно легко написать и без *goto*.

Спорщик, утверждающий: «Я не могу жить без *goto*», — обычно приводит случай, в котором исключение *goto* выливается в дополнительное сравнение или дублирование кода. Это доказывает в основном то, что есть случаи, в которых *goto* позволяет выполнить на одно сравнение меньше — незначительная выгода для современных компьютеров.

Большинство учебников также не помогает. Они приводят простой пример переписывания некоторого кода без *goto*, как будто это все объясняет. Вот обманчивый пример тривиального фрагмента кода из такого учебника:

**Пример кода, который должен легко переписываться без *goto* (C++)**

```
do {
   GetData( inputFile, data );
   if ( eof( inputFile ) ) {
      goto LOOP_EXIT;
   }
   DoSomething( data );
} while ( data != -1 );
LOOP_EXIT:
```

Книга быстро заменяет этот фрагмент кодом без *goto*:

**Пример предположительно эквивалентного кода, переписанного без *goto* (C++)**

```
GetData( inputFile, data );
while ( ( !eof( inputFile ) ) && ( ( data != -1 ) ) ) {
   DoSomething( data );
   GetData( inputFile, data )
}
```

Этот так называемый «простой» пример содержит ошибку. В случае, когда переменная *data* равна *-1*, преобразованный код отслеживает *-1* и выходит из цикла до выполнения *DoSomething()*. Исходный код выполняет *DoSomething()* до того, как *-1* обнаружена. Автор книги по программированию, пытаясь показать, как легко можно кодировать без *goto*, преобразовал собственный же пример некорректно. Но ему не стоит расстраиваться — другие книги содержат похожие ошибки. Даже профессионалы сталкиваются с трудностями при преобразовании кода, использующего *goto*.

Вот более точная реорганизация кода без *goto*:

**Пример действительно эквивалентного кода, переписанного без *goto* (C++)**

```
do {
   GetData( inputFile, data );
   if ( !eof( inputFile )) {
      DoSomething( data );
   }
} while ( ( data != -1 ) && ( !eof( inputFile ) ) );
```

Даже при правильном преобразовании кода этот пример все же искусственный, потому что он показывает тривиальный вариант использования *goto*. Это не тот случай, когда толковые программисты выбирают *goto* в качестве предпочтительной формы управления.

В наши дни уже тяжело добавить что-нибудь стоящее к теоретическим дебатам вокруг *goto*. Однако на что обычно не обращают внимания, так это на ситуации, в кото-

рых программист, полностью представляя себе альтернативы без *goto*, все же решает использовать его для улучшения читабельности и качества сопровождения.

Следующие разделы представляют случаи, в которых некоторые опытные программисты приводят доводы в пользу *goto*. В обсуждении рассматриваются примеры кода с операторами *goto* и кода, переписанного без их использования, и оцениваются достоинства и недостатки этих версий.

## Обработка ошибок и операторы *goto*

Создание высокоинтерактивного кода заставляет обращать особое внимание на обработку ошибок и освобождение ресурсов в случае возникновения ошибки. Следующий пример стирает группу файлов. Метод сначала получает группу файлов для удаления, затем находит каждый файл, открывает его, перезаписывает, а затем удаляет. Метод проверяет возникновение ошибок на каждом шаге.

**Пример кода с *goto*, который обрабатывает ошибки и освобождает ресурсы (Visual Basic)**

```
' Этот метод стирает группу файлов.
Sub PurgeFiles( ByRef errorState As Error_Code )
   Dim fileIndex As Integer
   Dim fileToPurge As Data_File
   Dim fileList As File_List
   Dim numFilesToPurge As Integer

   MakePurgeFileList( fileList, numFilesToPurge )

   errorState = FileStatus_Success
   fileIndex = 0
   While ( fileIndex < numFilesToPurge )
      fileIndex = fileIndex + 1
      If Not ( FindFile( fileList( fileIndex ), fileToPurge ) ) Then
         errorState = FileStatus_FileFindError

Здесь используется GoTo.
         GoTo END_PROC
      End If

       If Not OpenFile( fileToPurge ) Then
         errorState = FileStatus_FileOpenError

Здесь используется GoTo.
         GoTo END_PROC
      End If

      If Not OverwriteFile( fileToPurge ) Then
         errorState = FileStatus_FileOverwriteError
```

```
Здесь используется GoTo.
          GoTo END_PROC
       End If

       if Not Erase( fileToPurge ) Then
          errorState = FileStatus_FileEraseError
Здесь используется GoTo.
          GoTo END_PROC
       End If
    Wend

Здесь находится метка GoTo.
END_PROC:
    DeletePurgeFileList( fileList, numFilesToPurge )
End Sub
```

Этот метод — типичный пример обстоятельств, при которых опытные программисты решают использовать *goto*. Похожее случается, когда методу надо выделить и освободить такие ресурсы, как соединения с базами данных, память или временные файлы. Альтернативой *goto* в таких ситуациях обычно является дублирование кода для очистки ресурсов. В подобных случаях программист может сравнить нежелательность применения *goto* с головной болью от сопровождения дублированного кода и решить, что *goto* — меньшее зло.

Вы можете переписать предыдущий пример без *goto* несколькими способами, и все они будут иметь как плюсы, так и минусы. Далее приведены возможные стратегии преобразования:

**Перекрестная ссылка** Этот метод также можно переписать, используя операторы *break* и без *goto*. Об этом подходе см. подраздел «Досрочное завершение цикла» раздела 16.2.

***Переписать с помощью вложенных операторов* if** При перезаписи с помощью вложенных *if* располагайте блоки *if* так, чтобы следующая проверка условия выполнялась, только если предыдущая завершилась успешно. Это стандартный, приводимый в учебниках подход к удалению операторов *goto*. Рассмотрим метод, переписанный с помощью стандартного подхода:

**Код, избавившийся от *goto* с помощью вложенных *if* (Visual Basic)**

```
' Этот метод стирает группу файлов.
Sub PurgeFiles( ByRef errorState As Error_Code )
   Dim fileIndex As Integer
   Dim fileToPurge As Data_File
   Dim fileList As File_List
   Dim numFilesToPurge As Integer

   MakePurgeFileList( fileList, numFilesToPurge )

   errorState = FileStatus_Success
   fileIndex = 0
```

Условие *While* изменено — добавлена проверка *errorState*.

```
    While ( fileIndex < numFilesToPurge And errorState = FileStatus_Success )

        fileIndex = fileIndex + 1

         If FindFile( fileList( fileIndex ), fileToPurge ) Then
           If OpenFile( fileToPurge ) Then
              If OverwriteFile( fileToPurge ) Then
                 If Not Erase( fileToPurge ) Then
                    errorState = FileStatus_FileEraseError
                 End If
              Else ' невозможно перезаписать файл
                 errorState = FileStatus_FileOverwriteError
              End If
           Else ' невозможно открыть файл
              errorState = FileStatus_FileOpenError
           End If
        Else ' файл не найден
```

Эта строка расположена через 13 строк после условия *If*, к которому она относится.

```
           errorState = FileStatus_FileFindError
        End If
    Wend
    DeletePurgeFileList( fileList, numFilesToPurge )
End Sub
```

Тому, кто привык программировать без *goto*, возможно, будет легче читать этот код, чем первоначальную версию. И если вы используете данный вариант, вам не придется предстать перед судом противников *goto*.

Основной недостаток этого подхода с вложенными *if* в том, что уровень вложенности глубок, даже слишком. Для понимания кода вам нужно держать в голове весь набор вложенных *if* одновременно. Более того, расстояние между кодом обработки ошибок и кодом, ее инициирующим, слишком велико: например, выражение, присваивающее переменной *errorState* значение *FileStatus_FileFindError*, на 13 строк отстоит от соответствующей проверки *if*.

**Перекрестная ссылка** Об отступах и других вопросах разметки кода см. главу 31. Об уровнях вложенности см. раздел 19.4.

В варианте с *goto* ни одно выражение не отстоит более чем на четыре строки от условия, которое его вызывает. И вам нет нужды держать в голове всю структуру одновременно. По сути вы можете игнорировать все предыдущие условия, выполненные успешно, и сосредоточиться на следующей операции. В этом случае версия с *goto* гораздо удобнее для чтения и сопровождения, чем с вложенными *if*.

***Переписать код с использованием статусной переменной*** Чтобы переписать код с использованием статусной переменной (также называемой переменной состояния), создайте переменную, которая будет показывать, не находится ли метод в состоянии ошибки. В нашем случае метод уже содержит статусную переменную *errorState*, так что вы можете использовать ее.

**Код, избавившийся от *goto* с помощью статусной переменной (Visual Basic)**

```
' Этот метод стирает группу файлов.
Sub PurgeFiles( ByRef errorState As Error_Code )
    Dim fileIndex As Integer
    Dim fileToPurge As Data_File
    Dim fileList As File_List
    Dim numFilesToPurge As Integer

    MakePurgeFileList( fileList, numFilesToPurge )

    errorState = FileStatus_Success
    fileIndex = 0

Условие While изменено — добавлена проверка errorState.
    While ( fileIndex < numFilesToPurge ) And ( errorState = FileStatus_Success )

        fileIndex = fileIndex + 1

        If Not FindFile( fileList( fileIndex ), fileToPurge ) Then
            errorState = FileStatus_FileFindError
        End If

Проверяется статусная переменная.
        If ( errorState = FileStatus_Success ) Then
            If Not OpenFile( fileToPurge ) Then
                errorState = FileStatus_FileOpenError
            End If
        End If

Проверяется статусная переменная.
        If ( errorState = FileStatus_Success ) Then
            If Not OverwriteFile( fileToPurge ) Then
                errorState = FileStatus_FileOverwriteError
            End If
        End If

Проверяется статусная переменная.
        If ( errorState = FileStatus_Success ) Then
            If Not Erase( fileToPurge ) Then
                errorState = FileStatus_FileEraseError
            End If
        End If
    Wend
    DeletePurgeFileList( fileList, numFilesToPurge )
End Sub
```

Преимущество подхода со статусной переменной в том, что он позволяет избежать глубоко вложенных структур *if-then-else*, используемых в предыдущем примере, и тем самым легче для понимания. Кроме того, он помещает действия, сле-

дующие за проверкой *if-then-else*, ближе к месту самой проверки, чем в случае с вложенными *if*, и совсем не использует блоки *else*.

Понимание версии с вложенными *if* требует некоторой умственной гимнастики. Вариант со статусной переменной легче для понимания, потому что лучше моделирует способ человеческого мышления. Вы ищете файл. Если все в порядке, вы открываете файл. Если все до сих пор в порядке, вы перезаписываете файл. Если все до сих пор в порядке...

Недостаток этого подхода в том, что использование статусных переменных — не настолько распространенная практика, как хотелось бы. Подробно документируйте их применение, иначе некоторые программисты могут не понять, что вы имели в виду. В данном примере применение хорошо названных перечислимых типов оказывает существенную помощь.

***Переписать с помощью* try-finally** Некоторые языки, включая Visual Basic и Java, предоставляют конструкцию *try-finally,* которая может быть использована для очистки ресурсов в случае ошибки.

Чтобы переписать пример, используя подход с *try-finally*, поместите код, который должен проверять возможные ошибки, в блок *try*, а код очистки — в блок *finally*. Блок *try* задает область обработки исключений, а *finally* выполняет любое освобождение ресурсов. Блок *finally* будет вызываться всегда независимо от того, будет ли сгенерировано исключение и будет ли это исключение *перехвачено* в методе *PurgeFiles()*.

#### Код, избавившийся от *goto* с помощью *try-finally* (Visual Basic)

```
' Этот метод стирает группу файлов. Исключения передаются вызывающей стороне.
Sub PurgeFiles()
   Dim fileIndex As Integer
   Dim fileToPurge As Data_File
   Dim fileList As File_List
   Dim numFilesToPurge As Integer
   MakePurgeFileList( fileList, numFilesToPurge )
   Try
      fileIndex = 0
      While ( fileIndex < numFilesToPurge )
         fileIndex = fileIndex + 1
         FindFile( fileList( fileIndex ), fileToPurge )
         OpenFile( fileToPurge )
         OverwriteFile( fileToPurge )
         Erase( fileToPurge )
      Wend
   Finally
      DeletePurgeFileList( fileList, numFilesToPurge )
   End Try
End Sub
```

Этот подход предполагает, что все вызовы функций в случае ошибки генерируют исключения, а не возвращают коды ошибок.

Преимущество подхода с применением *try-finally* в том, что он проще, чем с *goto* и не использует *goto*. Кроме того, он позволяет избежать глубоко вложенных структур *if-then-else*.

Ограничением данного варианта с *try-finally* является то, что он должен быть последовательно реализован во всем коде. Если бы предыдущий пример был частью программы, использующей коды ошибок наряду с исключениями, то коду исключения пришлось бы устанавливать код ошибки для всех возможных ошибок, и это требование сделало бы фрагмент примерно таким же сложным, как и другие варианты.

### Сравнение рассмотренных подходов

**Перекрестная ссылка** Полный список методик, которые можно применять в аналогичных ситуациях, перечислен в подразделе «Сводка методик уменьшения глубины вложенности» раздела 19.4.

В защиту каждой из четырех приведенных методик есть что сказать. Подход с *goto* позволяет избежать глубокой вложенности и ненужных проверок, но, увы, он содержит *goto*. Подход с вложенными *if* позволяет обойтись без *goto*, но его глубокая вложенность преувеличивает картину логической сложности метода. Подход со статусной переменной избегает *goto* и глубокой вложенности, но добавляет дополнительные проверки. И, наконец, подход с *try-finally* тоже позволяет избежать как *goto*, так и глубокой вложенности, но доступен не во всех языках.

Вариант с *try-finally* наиболее предпочтителен в языках, предоставляющих такую конструкцию и в системах, еще не стандартизовавших какой-то иной подход. Если этот вариант невозможен, то подход со статусной переменной немного предпочтительнее, чем *goto* и вложенные *if*, так как он читабельнее и лучше моделирует задачу, однако это не делает его лучшим во всех ситуациях.

Все эти методики работают хорошо, если последовательно применяются ко всему коду проекта. Рассмотрите все плюсы и минусы, а затем примите решение на уровне проекта о том, какой подход предпочесть.

## Операторы *goto* и совместное использование кода в блоке *else*

Одна из возможных ситуаций, в которой некоторые программисты захотят использовать *goto*, — это случай, когда у вас есть две проверки условия и блок *else* и вы хотите выполнить код одного из условий и блока *else*. Вот пример варианта, который может кого-нибудь подвигнуть к использованию *goto*:

**Пример совместного использования кода в блоке *else* с помощью *goto* (C++)**

```
if ( statusOk ) {
   if ( dataAvailable ) {
      importantVariable = x;
      goto MID_LOOP;
   }
}
```

```
else {
   importantVariable = GetValue();

   MID_LOOP:

   // Много кода.
   ...
}
```

Это хороший, логически извилистый пример: его практически невозможно читать в том виде, в каком он есть, и тяжело правильно переписать без *goto*. Если вам кажется, что вы легко преобразуете его в вариант без *goto*, попросите кого-нибудь проверить ваш код! Несколько экспертов-программистов переписали его некорректно.

Вы можете изменить этот код разными способами. Можно продублировать код, вынести общий код в отдельный метод и вызывать его из двух мест или повторно проверять одно и то же условие. В большинстве языков новая версия будет больше и медленнее оригинала, но совсем не намного. Поэтому, если этот код не исключительно критичен к скорости и объему, перепишите его, не задумываясь об эффективности.

Лучший способ переписать этот код — вынести участок *// Много кода* в отдельный метод. После этого вы сможете вызывать его из тех мест, где раньше располагались *goto* и метка его перехода. Это позволит сохранить оригинальную структуру условного оператора. Вот как это выглядит:

**Пример совместного использования кода в блоке *else* с помощью вынесения общего кода в отдельный метод (C++)**

```
if ( statusOk ) {
   if ( dataAvailable ) {
      importantVariable = x;
      DoLotsOfCode( importantVariable );
   }
}
else {
   importantVariable = GetValue();
   DoLotsOfCode( importantVariable );
}
```

В общем случае написание нового метода — лучший подход. Однако иногда выносить дублированный код в отдельный метод непрактично. В этом случае как обходной маневр можно предложить реструктурирование условных выражений так, чтобы оставить код в том же методе, а не выносить в отдельный:

**Пример совместного использования кода в блоке *else* без применения *goto* (C++)**

```
if ( ( statusOk && dataAvailable ) || !statusOk ) {
   if ( statusOk && dataAvailable ) {
      importantVariable = x;
   }
```

```
    else {
        importantVariable = GetValue();
    }

    // Много кода.
    ...
}
```

**Перекрестная ссылка** Иной подход к данной проблеме — использование таблицы решений (см. главу 18).

Это точное механическое преобразование логики варианта с *goto*. Здесь переменная *statusOK* дополнительно проверяется два раза, а *dataAvailable* — один, но сам код эквивалентен. Если повторная проверка условия вас беспокоит, обратите внимание, что значение *statusOK* не обязательно проверять дважды в первом условии *if*. Кроме того, вы также можете опустить проверку *dataAvailable* во втором условии *if*.

## Краткий итог основных принципов использования *goto*

Использование *goto* — это вопрос религии. Моя догма: в современных языках вы легко можете заменить девять из десяти операторов *goto* эквивалентными последовательными конструкциями. В этих простых случаях вы должны заменять операторы *goto* просто по привычке. В сложных случаях вы также можете изгнать *goto* в девяти случаях из десяти: можно разбить код на меньшие по размеру методы, использовать *try-finally* или вложенные *if*, проверять и перепроверять статусную переменную или реструктурировать условные выражения. Исключить *goto* в таких случаях сложнее, но это хорошее умственное упражнение, а методы, обсуждаемые в этом разделе, предлагают вам инструменты для этих целей.

В одном случае, оставшемся из 100, в котором применение *goto* — вполне легальное решение задачи, подробно задокументируйте, а затем используйте его. Если у вас на ногах резиновые сапоги, не стоит обходить весь квартал, чтобы не запачкаться в грязной луже. Но не отвергайте варианты избавления от *goto*, предлагаемые другими программистами. Они могут заметить то, на что вы не обратили внимания.

Вот сводка принципов использования *goto*.

- Применяйте *goto* для эмуляции структурированных управляющих конструкций в языках, не поддерживающих их напрямую. Причем эмулируйте их точно — не злоупотребляйте дополнительной гибкостью, предоставляемой оператором *goto*.
- Не используйте *goto*, если доступна эквивалентная встроенная конструкция.

**Перекрестная ссылка** О повышении эффективности см. главы 25 и 26.

- Измеряйте производительность всех *goto*, используемых для повышения эффективности. В большинстве случаев вы можете переписать код без *goto* с целью повышения читабельности и при этом не потерять в эффективности. Если ваш случай — исключение, задокументируйте улучшение эффективности так, чтобы поборники кода без *goto* не удалили эти операторы, когда их увидят.
- Ограничьтесь использованием одной метки *goto* на метод, если только вы не эмулируете управляющие конструкции.

- Используйте операторы *goto* так, чтобы их переходы были только вперед, а не назад, если только вы не эмулируете управляющие конструкции.
- Убедитесь, что используются все метки *goto*. Неиспользуемые метки могут служить признаком недописанного кода, а именно того, в котором осуществляется переход по этим меткам. Если метки не используются, удалите их.
- Убедитесь, что *goto* не приводит к созданию недостижимого кода.
- Если вы менеджер, думайте о перспективе. Битва по поводу одного единственного *goto* не стоит поражения в целой войне. Если программист представляет себе альтернативы и готов к диалогу, то, возможно, использование *goto* вполне допустимо.

## 17.4. Перспективы нестандартных управляющих структур

Время от времени кто-нибудь решает, что эти управляющие структуры — хорошая идея:

- неограниченное использование операторов *goto*;
- возможность вычислять метку перехода *goto* динамически и переходить по этому адресу;
- возможность применения *goto* для перехода из середины одного метода в середину другого;
- возможность вызывать метод с помощью номера строки или метки, которые позволят начать выполнение с середины метода;
- возможность генерации кода программой на лету и немедленного его выполнения.

В свое время каждая из этих идей считалась приемлемой или даже желательной, хотя сейчас они все выглядят безнадежно устаревшими или опасными. Область разработки ПО развивается во многом благодаря *ограничению* того, что программисты могут делать со своим кодом. В связи с этим я рассматриваю нетрадиционные управляющие структуры с большим скептицизмом. Я подозреваю, что большинство конструкций, упомянутых в этой главе, со временем окажется на свалке программистских отходов наряду с вычисляемыми метками *goto*, плавающими точками входа в методы, самомодифицирующимся кодом и другими структурами, отдающими предпочтение гибкости и удобству в ущерб структурированности и возможности управления сложностью.

## Дополнительные ресурсы

Следующие материалы позволят расширить ваши представления о нестандартных управляющих структурах.

http://cc2e.com/1792

### Возвраты

Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999. В описании метода рефакторинга под названием «Замените вложенные

условные выражения сторожевыми операторами» Фаулер предлагает использовать множественные возвраты из методов для уменьшения вложенности набора *if*-выражений. Фаулер приводит доводы в защиту того, что множественные операторы *return* — подходящий способ улучшения ясности кода и нет никакого вреда в том, чтобы иметь несколько выходов из метода.

### Операторы *goto*

Следующие статьи содержат полное обсуждение *goto*. Этот спор до сих пор возникает время от времени на рабочих местах, в учебниках и журналах, но вы не услышите ничего такого, что не было бы полностью исследовано 20 лет назад.

http://cc2e.com/1799

Dijkstra, Edsger. «Go To Statement Considered Harmful». *Communications of the ACM* 11, no. 3 (March, 1968): 147–148, также доступно по адресу *www.cs.utexas.edu/users/EWD/*. Это то самое знаменитое письмо, которым Дейкстра поднес спичку к бумаге и воспламенил одну из самых долгих дискуссий в истории разработки ПО.

Wulf, W. A. «A Case Against the GOTO». *Proceedings of the 25th National ACM Conference*, August 1972: 791–797. Эта статья — еще один аргумент против беспорядочного использования *goto*. Вульф утверждает, что, если языки программирования будут содержать необходимые управляющие структуры, необходимость в *goto* исчезнет. С 1972 г., когда была написана эта статья, такие языки, как C++, Java и Visual Basic, доказали свою корректность по Вульфу.

Knuth, Donald. «Structured Programming with go to Statements», 1974. *Classics in Software Engineering*, edited by Edward Yourdon. Englewood Cliffs, NJ: Yourdon Press, 1979. Эта длинная статья не полностью посвящена *goto*, но содержит кучу примеров кода, который становится эффективнее после исключения *goto*, и еще одну кучу примеров кода, который становится эффективней после добавления *goto*.

Rubin, Frank. «'GOTO Considered Harmful' Considered Harmful». *Communications of the ACM* 30, no. 3 (March, 1987): 195–196. В этой несколько резкой статье, обращенной к редактору, Рубин утверждает, что программирование без *goto* стоило бизнесу «сотни миллионов долларов». Затем он предлагает краткий фрагмент кода, использующего *goto*, и утверждает, что он превосходит свои аналоги без *goto*.

Ответы, полученные на письмо Рубина, представляют больший интерес, чем само письмо. Пять месяцев журнал «Communications of the ACM» (CACM) публиковал письма, предлагающие разные версии программы Рубина из семи строк. Ответы равномерно распределились между защитниками и хулителями *goto*. Читатели предложили приблизительно 17 вариантов преобразования, которые полностью покрывают все подходы к исключению *goto*. Редактор CACM заметил, что это письмо вызвало больше откликов, чем любой другой вопрос, когда-либо обсуждавшийся на страницах CACM.

Последовавшие письма можно найти в номерах:

- *Communications of the ACM* 30, no. 5 (May, 1987): 351–355;
- *Communications of the ACM* 30, no. 6 (June, 1987): 475–478;
- *Communications of the ACM* 30, no. 7 (July, 1987): 632–634;
- *Communications of the ACM* 30, no. 8 (August, 1987): 659–662;
- *Communications of the ACM* 30, no. 12 (December, 1987): 997, 1085.

Clark, R. Lawrence, «A Linguistic Contribution of GOTO-less Programming». *Datamation*, December 1973. Эта классическая статья с юмором предлагает заменить термин «go to» (перейти к) на «come from» (перешел от). Она также была перепечатана в номере CACM в апреле 1974 года.

http://cc2e.com/1706

### Контрольный список: нестандартные управляющие структуры

http://cc2e.com/1713

**Возвраты**

- ❑ Используют ли методы операции возврата только при необходимости?
- ❑ Улучшают ли операторы возврата читабельность?

**Рекурсия**

- ❑ Содержит ли рекурсивный метод код для прекращения рекурсии?
- ❑ Использует ли метод счетчик безопасности для гарантии того, что выполнение будет завершено?
- ❑ Ограничена ли рекурсия одним методом?
- ❑ Соответствует ли глубина рекурсии ограничениям, налагаемым размерами стека программы?
- ❑ Является ли рекурсия лучшим способом реализации метода? Не лучше ли использовать простые итерации?

***goto***

- ❑ Используются ли операторы *goto* только как последнее средство и лишь для того, чтобы сделать код удобнее для чтения и сопровождения?
- ❑ Если goto используется ради эффективности, был ли прирост эффективности измерен и задокументирован?
- ❑ Ограничено ли использование *goto* одной меткой на метод?
- ❑ Выполняются ли переходы *goto* только вперед, а не назад?
- ❑ Все ли метки *goto* используются?

## Ключевые моменты

- Множественные возвраты могут улучшить читабельность и сопровождаемость метода и помогают избежать глубокой вложенности. Тем не менее использовать их нужно осторожно.
- Рекурсия предлагает изящное решение для небольшого набора задач. Ее тоже нужно использовать аккуратно.
- Иногда операторы *goto* — лучший способ облегчить чтение и сопровождение кода. Таких случаев очень немного. Используйте *goto* только как последнее средство.

ГЛАВА 18

# Табличные методы

http://cc2e.com/1865

**Содержание**



**Связанные темы**

- Сокрытие информации: подраздел «Скрывайте секреты (к вопросу о сокрытии информации)» раздела 5.3
- Проектирование классов: глава 6
- Использование таблиц решений для замены сложной логики: раздел 19.1
- Замена сложных выражений табличным поиском: раздел 26.1

Табличный метод — это схема, позволяющая искать информацию в таблице, а не использовать для этого логические выражения, такие как *if* и *case*. Практически все, что вы можете выбирать посредством логических операторов, можно выбирать, применяя таблицы. В простых случаях логические выражения проще и понятней. Но при усложнении логических построений таблицы становятся все привлекательнее.

Если вы уже знакомы с табличными методами, считайте эту главу обзором. В этом случае вы можете изучить «Пример гибкого формата сообщения» (раздел 18.2) в качестве иллюстрации того факта, что объектно-ориентированный дизайн не обязательно лучше других вариантов только потому, что он объектно-ориентированный. После этого можете переходить к обсуждению общих вопросов управления в главе 19.

# 18.1. Основные вопросы применения табличных методов

При определенных обстоятельствах табличный код проще, чем сложные логические выражения, легче поддается изменению и эффективнее. Допустим, вы хотите классифицировать символы, выделив буквы, знаки препинания и цифры. Вы можете использовать сложную логическую последовательность вроде этой:

**Пример использования сложной логики для классификации символов (Java)**

```
if ( ( ( 'a' <= inputChar ) && ( inputChar <= 'z' ) ) ||
   ( ( 'A' <= inputChar ) && ( inputChar <= 'Z' ) ) ) {
   charType = CharacterType.Letter;
}
else if ( ( inputChar == ' ' ) || ( inputChar == ',' ) ||
   ( inputChar == '.' ) || ( inputChar == '!' ) || ( inputChar == '(' ) ||
   ( inputChar == ')' ) || ( inputChar == ':' ) || ( inputChar == ';' ) ||
   ( inputChar == '?' ) || ( inputChar == '-' ) ) {
   charType = CharacterType.Punctuation;
}
else if ( ( '0' <= inputChar ) && ( inputChar <= '9' ) ) {
   charType = CharacterType.Digit;
}
```

Если бы вместо этого фрагмента вы использовали таблицу подстановки, то поместили бы тип каждого элемента в массив и обращались бы к нему по коду символа. Сложный фрагмент кода, представленный выше, заменялся бы на такое выражение:

**Пример использования таблицы подстановки для классификации символов (Java)**

```
charType = charTypeTable[ inputChar ];
```

Этот фрагмент предполагает, что массив *charTypeTable* был заранее заполнен. Вы поместили знания, доступные программе, в данные, а не в логику: в таблицу, а не в условия *if*.

## Два вопроса применения табличных методов

При применении табличных методов перед вами стоят два основных вопроса. Во-первых, вам надо решить, как будет выполняться поиск записей в таблице. Вы можете использовать какие-либо данные для прямого доступа к таблице. Так, если вам нужно систематизировать данные по месяцам, то выбор ключа для таблицы месяцев очевиден. Вы можете использовать массив с индексом от 1 до 12.

Другие данные затруднительно использовать для прямого поиска табличной записи. Так, для классификации информации по номеру социального страхования (SSN) вы не можете использовать этот номер в качестве ключа непосредственно, если, конечно, вы не собираетесь хранить в таблице 999-99-9999 записей. Вам понадобится более сложный подход. Вот какие способы, применяются для поиска записи в таблице:

- прямой доступ;
- индексированный доступ;
- ступенчатый доступ.

Каждый из этих вариантов доступа подробно описан ниже.

Второй вопрос, который нужно решить при использовании табличных методов: что хранить в таблице. Иногда результатом поиска в таблице являются данные — тогда можно хранить в таблице сами данные. Если же результатом поиска является действие, код, который описывает это действие, можно хранить, а в некоторых языках можно хранить ссылку на метод, выполняющий это действие. В каждом из этих случаев таблицы усложняются.

# 18.2. Таблицы с прямым доступом

Как и все таблицы подстановки, таблицы с прямым доступом предназначены для замены более сложных логических структур. Они имеют «прямой доступ», потому что вам не нужно ходить по кругу, чтобы найти в таблице необходимую информацию. Вы можете непосредственно выбрать нужную запись (рис. 18-1).

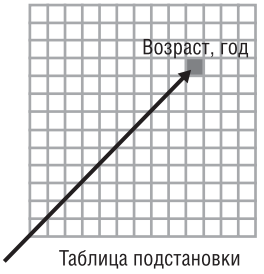



***Рис. 18-1.*** *Как следует из ее имени, таблица с прямым доступом позволяет обращаться к требуемому элементу напрямую*

## Пример определения количества дней в месяце

Допустим, вы хотите получить число дней в месяце (для простоты забудем о високосных годах). Разумеется, создание большого условия *if* — неуклюжий способ решения этой проблемы:

**Пример неуклюжего способа определения количества дней в месяце (Visual Basic)**

```
If ( month = 1 ) Then
   days = 31
ElseIf ( month = 2 ) Then
   days = 28
ElseIf ( month = 3 ) Then
   days = 31
ElseIf ( month = 4 ) Then
   days = 30
ElseIf ( month = 5 ) Then
   days = 31
ElseIf ( month = 6 ) Then
```

```
   days = 30
ElseIf ( month = 7 ) Then
   days = 31
ElseIf ( month = 8 ) Then
   days = 31
ElseIf ( month = 9 ) Then
   days = 30
ElseIf ( month = 10 ) Then
   days = 31
ElseIf ( month = 11 ) Then
   days = 30
ElseIf ( month = 12 ) Then
   days = 31
End If
```

Более простой и удобный для модификации способ выполнения тех же самых действий — разместить данные в таблице. В Visual Basic первым делом нужно заполнить таблицу:

**Пример элегантного способа определения количества дней в месяце (Visual Basic)**

```
' Инициализируем таблицу данными о количестве дней в месяцах.
Dim daysPerMonth() As Integer = _
   { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

А теперь вместо создания длинного выражения *if* для выяснения числа дней в месяце можно просто обратиться к элементу массива:

**Пример элегантного способа определения количества дней в месяце на Visual Basic (продолжение)**

```
days = daysPerMonth( month-1 )
```

Если вы хотите учитывать високосные годы в этой версии табличного поиска, код все еще будет простым. Допустим, *LeapYearIndex()* возвращает *0* или *1*:

**Пример элегантного способа определения количества дней в месяце на Visual Basic (продолжение)**

```
days = daysPerMonth( month-1, LeapYearIndex() )
```

Если бы в версии с выражением *if* тоже учитывались високосные годы, то длинная строка условий *if* еще более усложнилась бы.

Определение количества дней в месяце — удобный пример, так как переменную *month* можно использовать для поиска записи в таблице. Для прямого доступа к таблице зачастую можно использовать данные, которые могли бы управлять последовательностью *if*-выражений.

## Пример со ставками страхования

Допустим, вы пишете программу для вычисления ставок медицинского страхования, которые варьируются в зависимости от возраста, пола, семейного положе-

ния и от того, курит ли страхователь. Если бы вы писали для этих ставок логическую управляющую структуру, то получилось бы нечто вроде этого:

**Пример неуклюжего способа расчета ставки страхования (Java)**

```
if ( gender == Gender.Female ) {
   if ( maritalStatus == MaritalStatus.Single ) {
      if ( smokingStatus == SmokingStatus.NonSmoking ) {
         if ( age < 18 ) {
            rate = 200.00;
         }
         else if ( age == 18 ) {
            rate = 250.00;
         }
         else if ( age == 19 ) {
            rate = 300.00;
         }
         ...
         else if ( 65 < age ) {
            rate = 450.00;
      }
      else {
         if ( age < 18 ) {
            rate = 250.00;
         }
         else if ( age == 18 ) {
            rate = 300.00;
         }
         else if ( age == 19 ) {
            rate = 350.00;
         }
         ...
         else if ( 65 < age ) {
            rate = 575.00;
         }
      }
   else if ( maritalStatus == MaritalStatus.Married )
   ...
}
```

Эта сокращенная версия логической структуры — хорошая иллюстрация того, насколько сложной может получиться программа. Она не учитывает замужних женщин, всех мужчин и большинства возрастов между 18 и 65 годами. Вы можете вообразить, насколько сложной станет эта структура, если запрограммировать всю таблицу ставок.

Вы можете сказать: «Да, но почему вы проверяете каждый возраст? Почему бы не поместить ставку для каждого возраста в массив?» Хороший вопрос, и одним из очевидных усовершенствований будет размещение ставок для каждого возраста в отдельных массивах.

Однако лучшее решение — создать массив ставок не только для каждого возраста, но вообще для всех факторов. Вот как объявить такой массив на Visual Basic:

**Пример объявления данных для заполнения таблицы ставок страхования (Visual Basic)**

```
Public Enum SmokingStatus
   SmokingStatus_First = 0
   SmokingStatus_Smoking = 0
   SmokingStatus_NonSmoking = 1
   SmokingStatus_Last = 1
End Enum

Public Enum Gender
   Gender_First = 0
   Gender_Male = 0
   Gender_Female = 1
   Gender_Last = 1
End Enum

Public Enum MaritalStatus
   MaritalStatus_First = 0
   MaritalStatus_Single = 0
   MaritalStatus_Married = 1
   MaritalStatus_Last = 1
End Enum

Const MAX_AGE As Integer = 125

Dim rateTable ( SmokingStatus_Last, Gender_Last, MaritalStatus_Last, _
   MAX_AGE ) As Double
```

Определив массив, необходимо придумать способ его заполнения. Вы можете использовать операторы присваивания, читать данные из дискового файла, вычислять данные или делать что-то еще. После подготовки данные могут применяться при расчете ставок. Сложная логическая структура, показанная ранее, заменяется простым выражением, например:

**Перекрестная ссылка** Одно из преимуществ табличного подхода в том, что можно поместить данные из таблицы в файл и читать его во время выполнения. Это позволит вам изменять такие параметры, как ставки страхования, не изменяя саму программу (см. раздел 10.6).

**Пример элегантного способа определения ставки страхования (Visual Basic)**

```
rate = rateTable( smokingStatus, gender, maritalStatus, age )
```

Основное преимущество этого подхода — в замене сложной логики табличным поиском. Такой код удобней читать и проще изменять.

## Пример гибкого формата сообщения

Таблицу можно использовать для реализации такой логики, которая слишком динамична для представления в коде. В примерах по классификации символов, количеству дней в месяцах и страховым ставкам вы хотя бы знали, что можете в случае необходимости написать длинную строку условий *if*. Однако иногда данные слишком сложны, чтобы жестко закодировать их с помощью операторов *if*.

Если вам кажется, что вы поняли принцип работы таблиц с прямым доступом, можете пропустить следующий пример. Тем не менее он немного сложнее предыдущих и продолжает демонстрацию мощности табличных подходов.

Допустим, вы разрабатываете метод для печати сообщений, хранящихся в файле. Обычно файл содержит около 500 сообщений, которые бывают примерно 20 видов. Изначально сообщения поступают от бакенов и включают в себя информацию о температуре воды, расположении бакена и т. д.

Каждое сообщение имеет несколько полей и начинается с заголовка, содержащего идентификатор, позволяющий узнать, с каким из примерно 20 видов сообщений вы имеете дело (рис. 18-2).

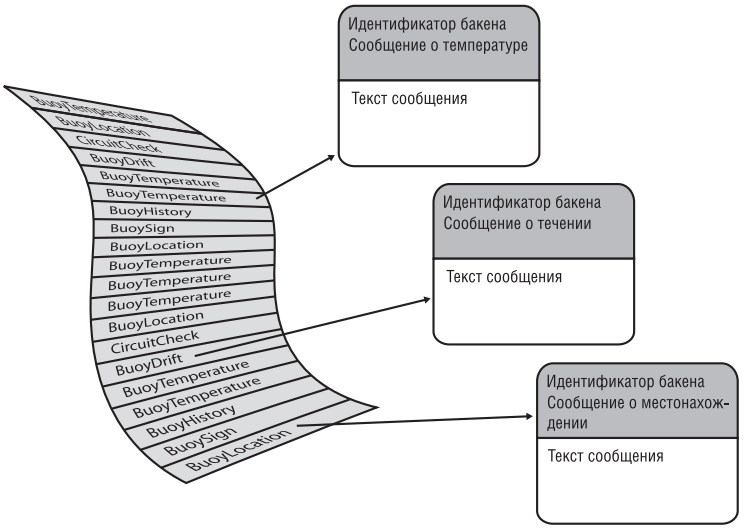



***Рис. 18-2.*** *Сообщения хранятся в произвольном порядке, каждое определяется идентификатором*

Сообщения имеют переменный формат, определяемый заказчиком, и вы не можете заставить его стабилизировать формат (рис. 18-3).

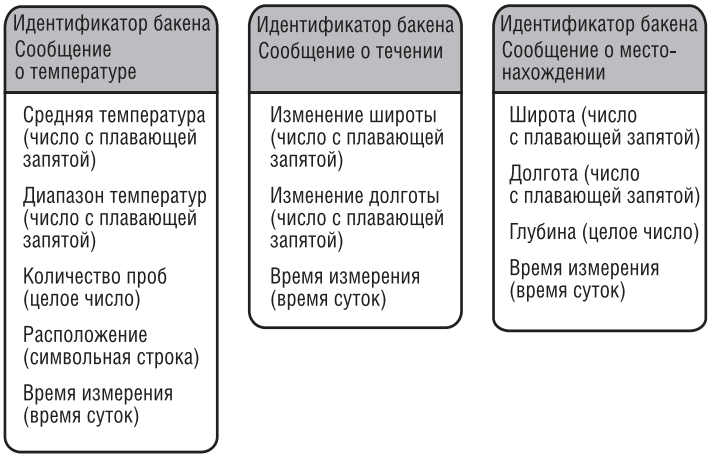



***Рис. 18-3.*** *За исключением идентификатора каждое сообщение имеет свой формат*

### Логический подход

Используя логический подход, вы, вероятно, прочитали бы каждое сообщение, проверили его идентификатор, а затем вызвали метод, разработанный для чтения, преобразования и печати каждого сообщения. Имей вы 20 типов сообщений, вы создали бы 20 методов. Для поддержки пришлось бы написать неизвестное количество методов более низкого уровня. Так, вы могли бы создать метод *PrintBuoyTemperatureMessage()* для печати сообщения о температуре. Объектно-ориентированный подход не дал бы никаких преимуществ: скорее всего вы задействовали бы абстрактный объект, представляющий сообщение, и породили от него подклассы для каждого типа сообщения.

При каждом изменении формата какого-нибудь сообщения вам пришлось бы менять логику в соответствующем классе или методе. Если в приведенном выше содержимом сообщения поле со средней температурой поменяло бы тип с плавающей запятой на иной, вам пришлось бы изменить логику метода *PrintBuoyTemperatureMessage()*. (А если бы изменился тип самого бакена, вам бы пришлось разрабатывать класс для нового!)

В логическом подходе метод для чтения сообщений состоит из цикла, читающего каждое сообщение, декодирующего его идентификатор, а затем вызывающего на основе этого идентификатора один из 20 методов. Вот пример псевдокода логического подхода:

```
Пока есть сообщения для чтения
   Прочитать заголовок сообщения
   Декодировать идентификатор сообщения из заголовка
   Если заголовок сообщения соответствует типу 1,
      то напечатать сообщение 1-го типа.
   Иначе, если заголовок сообщения соответствует типу 2,
      то напечатать сообщение 2-го типа.
   ...
```

**Перекрестная ссылка** Этот псевдокод низкого уровня используется в иных целях, нежели псевдокод, предназначенный для проектирования метода. О разработке с помощью псевдокода см. главу 9.

```
   Иначе, если заголовок сообщения соответствует типу 19,
      то напечатать сообщение 19-го типа.
   Иначе, если заголовок сообщения соответствует типу 20,
      то напечатать сообщение 20-го типа.
```

Текст этого псевдокода приводится не полностью — понять его смысл можно и без просмотра всех 20 вариантов.

### Объектно-ориентированный подход

При использовании механического объектно-ориентированного подхода логика была бы скрыта в структуре унаследованных объектов, но основная структура была бы столь же сложной:

```
Пока есть сообщения для чтения,
   прочитать заголовок сообщения.
   Декодировать идентификатор сообщения из заголовка.
   Если заголовок сообщения соответствует типу 1,
      то создать объект сообщения 1-го типа.
   Иначе, если заголовок сообщения соответствует типу 2,
      то создать объект сообщения 2-го типа.
   ...
   Иначе, если заголовок сообщения соответствует типу 19,
      то создать объект сообщения 19-го типа.
   Иначе, если заголовок сообщения соответствует типу 20,
      то создать объект сообщения 20-го типа.
   Конец Если
Конец цикла Пока
```

Независимо от того, будет ли логика написана непосредственно или реализована в специальных классах, каждое из 20 видов сообщений будет иметь собственный метод печати. Каждый такой метод тоже можно изобразить с помощью псевдокода. Вот его пример для метода, считывающего и печатающего сообщение о температуре бакена:

```
Напечатать «Сообщение о температуре бакена».

Прочитать значение с плавающей запятой.
Напечатать «Средняя температура».
Напечатать значение с плавающей запятой.

Прочитать значение с плавающей запятой.
Напечатать «Диапазон температур».
Напечатать значение с плавающей запятой.

Прочитать целое значение.
Напечатать «Количество проб».
Напечатать целое значение.

Прочитать символьную строку.
Напечатать «Местонахождение».
Напечатать символьную строку.
```

```
Прочитать время суток.
Напечатать «Время измерения».
Напечатать время суток.
```

Это код только для одного типа сообщений. Для каждого из оставшихся 19 типов нужно реализовать похожий код. И если будет добавлен 21-й тип сообщения, потребуется добавить 21-й метод или подкласс — в любом случае новый тип сообщения потребует изменения существующего кода.

### Табличный подход

Табличный подход экономичнее предыдущего. Метод чтения сообщений состоит из цикла, который считывает заголовок каждого сообщения, декодирует его идентификатор, находит описание сообщения в массиве *Message*, а затем всегда вызывает один и тот же метод для декодирования сообщения. Этот подход позволяет описать формат каждого сообщения в форме таблицы, а не задавать его жестко в логике программы. Это упрощает первоначальное программирование, создает меньше кода и облегчает сопровождение программы без изменения кода.

Применение этого подхода начинается с перечисления типов сообщений и типов полей. В C++ вы можете определить типы всех возможных полей таким образом:

#### Пример определения типов данных сообщения (C++)

```
enum FieldType {
    FieldType_FloatingPoint,
    FieldType_Integer,
    FieldType_String,
    FieldType_TimeOfDay,
    FieldType_Boolean,
    FieldType_BitField,
    FieldType_Last = FieldType_BitField
};
```

Вместо жестко закодированных методов печати каждого из 20 видов сообщений можно создать горстку функций для печати основных типов данных: чисел с плавающей точкой, целых чисел, символьных строк и т. д. Вы можете описать содержимое каждого типа сообщения в таблице (с указанием имени каждого поля), а затем декодировать все сообщения на основе этого описания. Элемент таблицы, содержащий сведения об одном типе сообщений, может выглядеть так:

#### Пример определения элемента таблицы, описывающего сообщение

```
Message Begin
   NumFields 5
    MessageName “Buoy Temperature Message”
    Field 1, FloatingPoint, “Average Temperature”
    Field 2, FloatingPoint, “Temperature Range”
    Field 3, Integer, “Number of Samples”
    Field 4, String, “Location”
    Field 5, TimeOfDay, “Time of Measurement”
Message End
```

Эта таблица может быть жестко закодирована в программе (в этом случае значения всех элементов будут присвоены переменным) или читаться из файла при запуске программы или позже.

Поскольку определения сообщений поступают в программу извне, то вместо внедрения информации в логику программы мы внедрили ее в данные. Данные обычно гибче программной логики: их легко изменять, если меняется формат сообщения. Если нужно добавить новый вид сообщений, вы можете просто добавить еще один элемент в таблицу данных.

Вот псевдокод цикла верхнего уровня для табличного подхода:

Первые три строки такие же, как и при логическом подходе.

```
Пока есть сообщения для чтения,
   прочитать заголовок сообщения,
   декодировать идентификатор сообщения из заголовка,
   найти описание сообщения в таблице описаний сообщений,
   прочитать поля сообщения и напечатать их, основываясь на описании сообщения.
Конец цикла Пока
```

В отличие от псевдокода при логическом подходе в этом случае псевдокод не сокращен, так как логика гораздо проще. Логика более низкого уровня содержит метод, который интерпретирует сообщение на основе таблицы описаний сообщений, считывает данные сообщения и печатает его. Этот метод более общего вида, чем методы печати сообщений при логическом подходе, но он не намного сложнее и заменяет собой все 20 методов:

```
Пока не все поля напечатаны,
   получить тип поля из описания сообщения.
   Выбор ( типа поля )
      вариант: ( число с плавающей запятой )
         прочитать значение с плавающей запятой,
         напечатать метку поля,
         напечатать значение с плавающей запятой.

      вариант: ( целое число )
         прочитать целое значение,
         напечатать метку поля,
         напечатать целое значение.

      вариант: ( символьная строка )
         прочитать символьную строку,
         напечатать метку поля,
         напечатать символьную строку.

      вариант: ( время суток )
         прочитать время суток,
         напечатать метку поля,
         напечатать время суток.

      вариант: ( логическое значение )
         прочитать значение флажка,
```

```
            напечатать метку поля,
            напечатать значение флажка.

         вариант: ( битовое поле )
            прочитать битовое поле,
            напечатать метку поля,
            напечатать битовое поле.
      Конец Выбора
   Конец цикла Пока
```

Нужно признать, что этот метод с шестью вариантами выбора длиннее, чем отдельный метод для печати температуры. Но это единственный метод, который вам необходим. Вам не нужны остальные 19 функций для остальных 19 типов сообщений. Данный метод обрабатывает шесть типов полей, и обслуживает все виды сообщений.

Этот метод также иллюстрирует наиболее сложный способ реализации табличного поиска, так как использует оператор *case*. Другой подход — создание абстрактного класса *AbstractField* и последующее наследование от него подклассов для каждого типа поля. Тогда вам не понадобится оператор *case*, вы сможете вызывать метод-член соответствующего объектного типа.

Вот как можно создать такие объекты на C++:

#### Пример создания объектных типов (C++)

```
class AbstractField {
   public:
   virtual void ReadAndPrint( string, FileStatus & ) = 0;
}

class FloatingPointField : public AbstractField {
   public:
   virtual void ReadAndPrint( string, FileStatus & ) {
   ...
   }
}

class IntegerField ...
class StringField ...
...
```

Этот фрагмент объявляет во всех классах метод, принимающий строковый параметр и параметр типа *FileStatus*.

Следующий шаг — объявление массива для хранения набора объектов. Этот массив и есть таблица для поиска. Вот как она выглядит:

#### Пример создания таблицы для хранения объектов каждого типа (C++)

```
AbstractField* field[ Field_Last ];
```

Последний шаг в настройке таблицы объектов — заполнение массива *Field* конкретными объектами:

**Пример заполнения списка объектов (C++)**

```
field[ Field_FloatingPoint ] = new FloatingPointField();
field[ Field_Integer ] = new IntegerField();
field[ Field_String ] = new StringField();
field[ Field_TimeOfDay ] = new TimeOfDayField();
field[ Field_Boolean ] = new BooleanField();
field[ Field_BitField ] = new BitFieldField();
```

В этом коде предполагается, что *FloatingPointField* и другие идентификаторы с правой стороны выражений присваивания — это имена объектов, унаследованных от *AbstractField*. Присваивание объектов элементам массива означает, что вы сможете вызвать правильную версию метода *ReadAndPrint()*, обращаясь к элементу массива, а не используя конкретный тип объекта напрямую.

Подготовив таблицу методов, можно обрабатывать поле сообщения с помощью простого обращения к таблице объектов и вызова одного из методов-членов этих объектов. Код может выглядеть так:

**Пример выбора объектов и их методов из таблицы (C++)**

```
Это строки — служебный код, необходимый для обработки каждого поля в сообщении.
fieldIdx = 1;
while ( ( fieldIdx <= numFieldsInMessage ) and ( fileStatus == OK ) ) {
    fieldType = fieldDescription[ fieldIdx ].FieldType;
    fieldName = fieldDescription[ fieldIdx ].FieldName;

Это — обращение к таблице, в результате которого будет вызван метод, зависящий от типа поля:
он просто выбирается в таблице объектов.
    field[ fieldType ].ReadAndPrint( fieldName, fileStatus );
}
```

Помните первоначальные 34 строки псевдокода табличного поиска, содержащего оператор *case*? Если вы замените оператор *case* таблицей объектов, то это весь код, который вам нужен для обеспечения той же функциональности. Невероятно, но это также весь код, необходимый для замены всех 20 отдельных методов, применяемых при логическом подходе. Более того, если описания сообщений читаются из файла, то новые типы сообщений не потребуют изменений кода, если только не будут содержать новых типов полей.

Вы можете использовать такой подход в любом объектно-ориентированном языке. Он менее подвержен ошибкам, легче в сопровождении и эффективнее длинных выражений *if*, операторов *case* или огромного количества подклассов.

Сам факт, что проект использует наследование и полиморфизм, не делает его хорошим проектом. Механический объектно-ориентированный дизайн, описанный в разделе «Объектно-ориентированный подход», потребовал бы такого же большого объема кода, как и механический функциональный дизайн, а может, и больше. Такой подход скорее усложнил бы решение, чем упростил. В данном случае основная суть проектного решения не в объектной и не в функциональной ориентации, а в использовании хорошо продуманной таблицы поиска.

## Подгонка значений ключа

Во всех трех предыдущих примерах вы могли использовать данные в качестве ключа для прямого обращения к таблице. То есть можно было указать переменную *messageID* как ключ без всяких изменений, переменную *month* в примере количества дней в месяцах, а также *gender*, *maritalStatus* и *smokingStatus* в примере ставок страхования.

Было бы хорошо всегда обращаться к таблице напрямую, потому что это просто и быстро. Однако не всегда данные для этого годятся. В примере со ставками страхования переменная *age* не очень удобна в качестве ключа. Первоначальная логика определяла одну ставку для лиц моложе 18 лет, индивидуальные ставки для возрастов от 18 до 65 и одну ставку для людей старше 65. Это означает, что для возрастов от 0 до 17 и от 66 и выше нельзя использовать возраст как ключ напрямую, если таблица хранит только один набор ставок для нескольких лет.

Это приводит к обсуждению вопроса подгонки значений ключа в таблице поиска. Подогнать ключ можно несколькими способами.

***Продублировать информацию, чтобы использовать ключ напрямую*** Один прямолинейный способ заставить *age* работать ключом в таблице ставок — продублировать все ставки для лиц, моложе 18, для каждого возраста от 0 до 17, а затем использовать возраст для прямого обращения к таблице. То же самое можно сделать и для возрастов от 66 лет и старше. Преимущество этого подхода в том, что структура таблицы остается простой и доступ к данным так же прост. Если нужно добавить специальное значение ставки для некоторого возраста, меньшего 17, вы можете просто изменить табличное значение. Недостаток этого метода в том, что дублирование приведет к напрасным затратам на хранение избыточной информации, а также увеличит вероятность появления ошибок в таблице хотя бы потому, что таблица будет содержать избыточные данные.

***Преобразовать ключ, чтобы использовать его напрямую*** Второй способ задействовать *Age* в качестве прямого ключа — применить к переменной *Age* некоторую функцию, которая позволит это делать. В данном случае такая функция должна преобразовывать все возрасты от 0 до 17 к какому-то одному значению, скажем, 17, а возрасты старше 66 — к другому, например, 66. В данном случае такое преобразование легко выполнить с помощью функций *min()* и *max()*. Так, для создания табличного ключа в диапазоне от 17 до 66 можно использовать выражение:

```
max( min( 66, Age ), 17 )
```

Реализация функции трансформации требует хорошего понимания структуры данных, которые вы хотите применить как ключ, и это не всегда так просто, как использование функций *min()* и *max()*. Допустим, в этом примере ставки меняются через интервалы не в 5 лет, а в 1 год. Если только вы не хотите дублировать все данные по пять раз, вам придется написать функцию, которая делит *Age* на 5 и использует методы *min()* и *max()*.

***Изолируйте преобразование ключа в собственном методе*** Если вам нужно подгонять данные для использования в качестве табличного ключа, поместите операции, трансформирующие данные в ключ, в отдельный метод. Его использование исключает возможность применения разных преобразований в разных

местах. Это упростит модификацию при изменении функции преобразования. Хорошее имя процедуры, такое как *KeyFromAge()*, также прояснит и задокументирует назначение математических махинаций.

Если ваша среда предоставляет готовые варианты преобразования ключа, используйте их. Например, класс *HashMap* в языке Java позволяет создавать пары «ключ-значение».

## 18.3. Таблицы с индексированным доступом

Иногда простого математического преобразования недостаточно для перехода от таких данных, как *Age* к значению ключа. Некоторые из таких случаев подходят для схем с индексным доступом.

Применяя индексы, вы используете исходные данные для поиска ключа в индексной таблице, а затем значение из этой таблицы служит для поиска интересующих вас данных.

Допустим, вы заведуете складом, и у вас около 100 наименований товара. Далее предположим, что каждый товар имеет четырехзначный номер в диапазоне от 0000 до 9999. В этом случае, если вы захотите задействовать номер товара в качестве ключа для прямого доступа к таблице, описывающей какой-то признак каждого товара, вам придется создать индексный массив с 10 000 записей (от 0 до 9999). Этот массив в основном будет пустым за исключением 100 элементов, соответствующих номерам товаров на вашем складе. Как показано на рис. 18-4, эти элементы указывают на таблицу с описанием товаров, содержащую гораздо менее 10 000 записей.

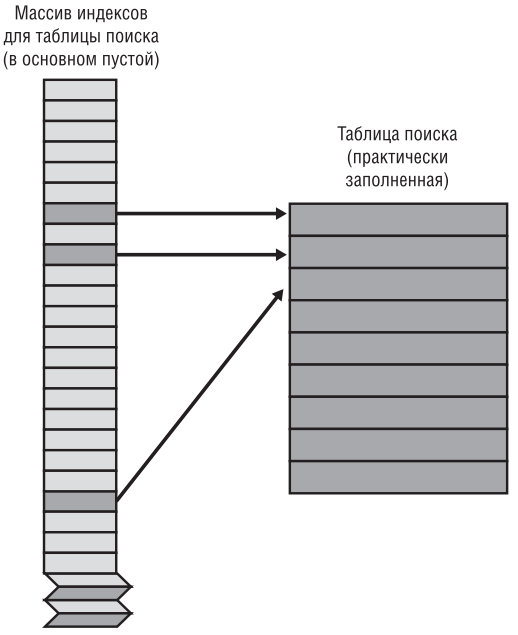



***Рис. 18-4.*** *В отличие от таблиц с прямым доступом для обращения к таблице с индексным доступом используется промежуточный индекс*

У схем с индексным доступом два главных преимущества. Во-первых, если каждый элемент главной таблицы поиска имеет большой размер, создание индексного массива с большим количеством пустых ячеек потребует гораздо меньше места, чем создание самой таблицы поиска с большим количеством пустых ячеек. Пусть, например, элемент основной таблицы занимает 100 байт, а элемент индексной таблицы — 2 байта. Далее предположим, что главная таблица содержит 100 записей, а данные, необходимые для обращения к ней, могут принимать 10 000 возможных значений. В этом случае выбор осуществляется между использованием индексной таблицы с 10 000 записей или только одной главной таблицы с 10 000 записей. Если вы используете индекс, общий объем требуемой памяти равен 30 000 байт. Если вы откажетесь от индекса и будете попусту тратить память в основной таблице, то общий объем используемой памяти составит 1 000 000 байт.

Вторым преимуществом индексного доступа (даже если вам не удастся сэкономить память с его помощью) является то, что иногда дешевле манипулировать элементами индекса, чем элементами основной таблицы. Так, если у вас есть таблица с именами работников, датами их приема на работу и окладами, вы можете создать один индекс для доступа к таблице по имени работника, второй — для доступа на основе даты приема, и третий — для доступа на основе размера зарплаты.

И последнее преимущество индексной схемы доступа — общее для всех таблиц поиска удобство сопровождения. Данные, закодированные в таблицах, легче сопровождать, чем внедренные в код. Для увеличения гибкости поместите код индексного доступа в отдельный метод и вызывайте его, когда вам нужно получить значение ключа на основе номера товара. Когда понадобится изменить таблицу, вы можете решить изменить схему индексного доступа или даже перейти к другому способу доступа к таблице поиска. Схему доступа будет легче поменять, если код доступа по индексу не разбросан по всей программе.

## 18.4. Таблицы со ступенчатым доступом

Еще один способ табличного доступа — ступенчатый метод. Этот метод не такой прямой, как индексная структура, но позволяет не терять такого большого объема памяти.

Основная идея ступенчатой структуры в том, что записи в таблице соответствуют некоторому диапазону данных, а не отдельным элементам (рис. 18-5).

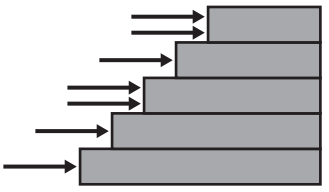


***Рис. 18-5.*** *Ступенчатый подход классифицирует каждый элемент, определяя уровень, на котором он наталкивается на «лестницу». «Ступенька», в которую этот элемент упирается, определяет его категорию*

Например, если вы пишете аттестационную программу, диапазон оценки «B» может быть в пределах 75–90%. Вот список оценок, которые вам однажды, возможно, придется запрограммировать:

| | |
|---|---|
| ≥ 90.0% | A |
| < 90.0% | B |
| < 75.0% | C |
| < 65.0% | D |
| < 50.0% | F |

Это ужасный диапазон для табличного поиска, потому что вы не можете написать простую функцию преобразования данных для соответствия буквам от *A* до *F*. Индексная схема неудобна, так как используются числа с плавающей запятой. Вы можете предложить конвертировать числа с плавающей запятой в целые, что для данного случая вполне допустимо, однако в целях иллюстрации этот пример будет придерживаться чисел с плавающей запятой.

Применяя ступенчатый метод, вы помещаете верхнюю границу каждого диапазона в таблицу, а затем пишете цикл для сравнения количества баллов с этой верхней границей. Обнаружив точку, в которой сумма баллов в первый раз превысит заданный предел, вы узнаете оценку. Применяя ступенчатую методику, надо следить за правильной обработкой граничных точек диапазона. Вот пример кода Visual Basic, присваивающей оценки группе студентов, основываясь на данных этого примера:

**Пример ступенчатого поиска в таблице (Visual Basic)**

```
' Подготавливаем данные для таблицы оценок.
Dim rangeLimit() As Double = { 50.0, 65.0, 75.0, 90.0, 100.0 }
Dim grade() As String =      { "F",  "D",  "C",  "B",   "A"  }
maxGradeLevel = grade.Length - 1
...

' Присваиваем оценки, основываясь на количестве баллов, набранных студентом.
gradeLevel = 0
studentGrade = "A"
While ( ( studentGrade = "A" ) and ( gradeLevel < maxGradeLevel ) )
   If ( studentScore < rangeLimit( gradeLevel ) ) Then
      studentGrade = grade( gradeLevel )
   End If
   gradeLevel = gradeLevel + 1
Wend
```

Хотя это и простой пример, вы легко можете его обобщить для работы с несколькими студентами или несколькими системами оценок (например, введя разные оценки для различных уровней сложности выполняемых заданий), а также для изменения системы оценок.

Преимущество этого подхода перед другими табличными методами в том, что он хорошо работает с нестандартными данными. Пример с оценками прост с той точки зрения, что, хотя оценки и присваиваются через неодинаковые промежут-

ки, сами рассматриваемые числа — «круглые», оканчивающиеся на 5 или 0. Ступенчатый способ столь же хорошо подходит и для данных, не заканчивающихся на 5 или 0. Вы можете применять эту методику и в статистических задачах для работы с такими примерами вероятностных распределений, как этот:

| Вероятность | Сумма страхового иска |
|---|---|
| 0,458747 | $0,00 |
| 0,547651 | $254,32 |
| 0,627764 | $514,77 |
| 0,776883 | $747,82 |
| 0,893211 | $1 042,65 |
| 0,957665 | $5 887,55 |
| 0,976544 | $12 836,98 |
| 0,987889 | $27 234,12 |
| ... | |

Такие ужасные числа сводят на нет все попытки создать функцию для их точного преобразования в табличные ключи. В этом случае следует использовать ступенчатый метод.

Этот подход также позволяет оценить главные преимущества табличных методов: гибкость и модифицируемость. Если диапазоны баллов в примере с оценками надо изменить, программу легко исправить, изменив элементы массива *RangeLimit*. Вы легко можете обобщить метод выставления отметок так, чтобы он принимал извне таблицы с отметками и соответствующими граничными значениями баллов. При выставлении оценок не обязательно использовать баллы, выраженные в процентах, их можно поменять на обычные единицы, и это не потребует больших изменений в программе.

Вот несколько тонкостей, которые надо принимать во внимание, применяя ступенчатый метод.

***Следите за граничными точками*** Убедитесь, что вы учитываете верхнюю границу каждого диапазона. Выполните ступенчатый поиск так, чтобы он находил значения, соответствующие любому интервалу, кроме самого верхнего, а затем задайте несколько элементов, попадающих в этот верхний интервал. Иногда требуется создать искусственное значение для верхней границы последнего интервала.

Не ошибитесь с операциями < или <=! Убедитесь, что при значениях, попадающих в верхний диапазон, цикл корректно завершается, а также что границы интервалов обрабатываются правильно.

***Рассмотрите вопрос использования бинарного поиска вместо последовательного*** В примере с оценками цикл, присваивающий отметки, последовательно проходит по списку предельных значений баллов. Если у вас будет более длинный список, затраты на последовательный поиск могут чрезмерно возрасти. В этом случае можно воспользоваться квази-бинарным поиском. «Квази» он потому, что цель большинства бинарных поисков — нахождение значения. В данном случае вы не намерены найти значение — вы ищете правильную категорию для этого значения. Алгоритм бинарного поиска должен корректно определить, куда это

значение попадает. Не забывайте также рассматривать граничные точки как специальный случай.

***Рассмотрите вопрос использования индексного доступа вместо ступенчатого метода*** Схема с индексным доступом (см. раздел 18.3) может быть хорошей альтернативой ступенчатому подходу. Поиск, выполняющийся в ступенчатом методе, может увеличить накладные расходы, и если, скорость выполнения имеет значение, вы, возможно, захотите пожертвовать объемом памяти и затратами на дополнительную индексную структуру, чтобы получить преимущество в скорости, обусловленное более прямым методом доступа.

Очевидно, эта альтернатива во всех случаях не годится. В примере с оценками вы, возможно, захотите ее использовать. Если у вас всего 100 дискретных процентных значений, расходы на дополнительную память для индексного массива не будут чрезмерными. С другой стороны, если вы работаете с данными вероятностного распределения, приведенными выше, вы не сможете применить индексную схему, потому что не сможете задействовать в качестве ключей к таблице данных такие числа, как 0,458747 и 0,547651.

**Перекрестная ссылка** О правильных подходах к выбору альтернативных способов проектирования см. главу 5.

В некоторых случаях подойдет любой из нескольких вариантов. И задачей проектирования является выбор одного из нескольких хороших вариантов для этого конкретного случая. Не слишком волнуйтесь по поводу выбора наилучшего. Как говорит Батлер Лемпсон, выдающийся инженер из Microsoft, лучше стремиться к хорошему решению и избегать неудач, чем пытаться найти наилучшее решение (Lampson, 1984).

***Поместите ступенчатый поиск в таблице в отдельный метод*** Когда вы создаете функцию преобразования, которая трансформирует такое значение, как *StudentGrade*, в табличный ключ, поместите ее в отдельный метод.

## 18.5. Другие примеры табличного поиска

Несколько примеров табличного поиска встречаются в других разделах этой книги. Они используются в процессе обсуждения других методик, и поэтому в контексте не подчеркивается их принадлежность к табличному поиску. Вот где вы можете их найти:

- поиск ставок в таблице страхования: раздел 16.3;
- применение таблиц решения для замены сложной логики: подраздел «Используйте таблицы решений для замены сложных условий» раздела 19.1;
- расходы на страничную организацию памяти при табличном поиске: раздел 25.3;
- комбинации логических значений (A или B или C): подраздел «Замена сложных выражений на обращение к таблице» раздела 26.1;
- предварительное вычисление значения в таблице погашения ссуд: раздел 26.4.

**Контрольный список: табличные методы**

http://cc2e.com/1872

- ❑ Рассмотрены ли табличные методы в качестве альтернативы сложной логике?
- ❑ Рассмотрены ли табличные методы в качестве альтернативы сложным структурам с наследованием?
- ❑ Рассмотрен ли вопрос размещения табличных данных отдельно от программы и их чтения во время выполнения, чтобы позволить обновлять данные без изменения кода?
- ❑ Если доступ к таблице нельзя осуществить напрямую с помощью простого индекса массива (как в примере с возрастами), помещены ли вычисления ключа доступа в отдельный метод, а не разбросаны по всему коду?

# Ключевые моменты

- Таблицы представляют собой альтернативу сложной логике и структурам с наследованием. Если вы понимаете, что сбиты с толку логикой программы или деревом наследования, спросите себя, не проще ли использовать таблицу поиска.
- Основной вопрос при использовании таблиц состоит в выборе способа доступа к таблице. Вы можете использовать прямой, индексный или ступенчатый доступ.
- Другой основной вопрос состоит в выборе того, что конкретно будет помещено в таблицу.

ГЛАВА 19

# Общие вопросы управления

http://cc2e.com/1978

**Содержание**



**Связанные темы**

- Последовательный код: глава 14
- Код с условными операторами: глава 15
- Код с циклами: глава 16
- Нестандартные управляющие структуры: глава 17
- Сложность в разработке ПО: подраздел «Главный Технический Императив Разработки ПО: управление сложностью» раздела 5.2

Обсуждение способов управления было бы неполным без углубления в несколько общих вопросов, касающихся управляющих структур. Большая часть этой главы посвящена подробностям их практического применения. Если вам интересней читать о теории управляющих структур, а не о мелких деталях, сосредоточьтесь на исторической перспективе структурного программирования (раздел 19.5), а затем на взаимодействии управляющих структур (раздел 19.6).

## 19.1. Логические выражения

Кроме простейших случаев, таких, в которых происходит последовательное выполнение операторов, все управляющие структуры зависят от вычисления логических выражений.

## Использование *true* и *false* в логических проверках

В логических выражениях следует использовать идентификаторы *true* и *false*, а не значения *0* и *1*. Большинство современных языков реализует логический тип данных и предоставляет предопределенные идентификаторы для значений «истина» и «ложь». В таких языках все просто: они даже не позволят вам присвоить логическим переменным другие значения, кроме *true* или *false*. В языках, не содержащих встроенного логического типа, от вас требуется большая дисциплинированность, чтобы сделать логические выражения читабельными. Приведем пример:

**Примеры использования неоднозначных флажков для логических значений (Visual Basic)**

```
Dim printerError As Integer
Dim reportSelected As Integer
Dim summarySelected As Integer
...
If printerError = 0 Then InitializePrinter()
If printerError = 1 Then NotifyUserOfError()

If reportSelected = 1 Then PrintReport()
If summarySelected = 1 Then PrintSummary()

If printerError = 0 Then CleanupPrinter()
```

Если использование флажков *0* и *1* — общая практика, то чем она плоха? При чтении кода неочевидно, когда выполняются вызовы функций: когда проверки условия истинны или когда они ложны. Ничего в этом фрагменте не говорит о том, представляет ли *1* «истину», а *0* — «ложь» или же все наоборот. Нельзя даже утверждать, что *1* и *0* служат в качестве значений «истина» и «ложь». Например, в строке *If reportSelected = 1* число *1* может легко обозначать первый отчет, *2* — второй, *3* — третий; ничто в коде не наводит на мысль, что *1* означает либо «истину», либо «ложь». Кроме того, легко можно написать *0*, имея в виду *1*, и наоборот.

Используйте термины *true* и *false* для проверки логических выражений. Если ваш язык не поддерживает их напрямую, создайте их с помощью макросов препроцессора или глобальных переменных. Вот как можно переписать предыдущий пример, используя встроенные идентификаторы *True* и *False* языка Visual Basic:

**Хорошие, но не превосходные примеры использования *True* и *False* вместо числовых значений для проверок условий (Visual Basic)**

```
Dim printerError As Boolean
Dim reportSelected As ReportType
Dim summarySelected As Boolean
...
If ( printerError = False ) Then InitializePrinter()
If ( printerError = True ) Then NotifyUserOfError()

If ( reportSelected = ReportType_First ) Then PrintReport()
If ( summarySelected = True ) Then PrintSummary()
```

**Перекрестная ссылка** Еще лучший подход к выполнению тех же проверок можно найти в следующем примере кода.

```
If ( printerError = False ) Then CleanupPrinter()
```

Применение констант *True* и *False* проясняет назначение кода. Вам не нужно помнить, что обозначают *1* и *0*, и вы не сможете их случайно перепутать. Более того, в переписанном коде стало понятно, что в некоторых случаях *1* и *0* в исходном примере на Visual Basic не являлись логическими флагами. Выражение *If reportSelected = 1* было не проверкой логического значения, а проверкой того, выбран ли первый отчет.

Этот подход сообщает читателю, что вы выполняете логическую проверку. Кроме того, сложнее написать *true*, подразумевая *false*, чем *1*, подразумевая *0*. Также вы избежите распространения магических чисел *0* и *1* по всему коду. Далее приведены советы по использованию *true* и *false* в логических проверках.

***Используйте неявное сравнение логических величин с true или false*** Вы сможете сделать проверку условия более понятной, если будете рассматривать проверяемые выражения как логические. Например, пишите:

```
while ( not done ) ...
while ( a > b ) ...
```

вместо:

```
while ( done = false ) ...
while ( (a > b) = true ) ...
```

Использование неявных сравнений уменьшает число элементов, которые придется помнить при чтении кода, и приближает получаемое выражение к разговорному английскому. Вот как улучшить стиль предыдущего примера:

#### Улучшенные примеры неявных проверок *True* или *False* (Visual Basic)

```
Dim printerError As Boolean
Dim reportSelected As ReportType
Dim summarySelected As Boolean
...
If ( Not printerError ) Then InitializePrinter()
If ( printerError ) Then NotifyUserOfError()

If ( reportSelected = ReportType_First ) Then PrintReport()
If ( summarySelected ) Then PrintSummary()

If ( Not printerError ) Then CleanupPrinter()
```

**Перекрестная ссылка** О логических переменных см. раздел 12.5.

Если ваш язык не поддерживает логические переменные и вам приходится их эмулировать, то, вероятно, вы не сможете использовать эту технологию, поскольку искусственные *true* и *false* не всегда могут проверяться в таких выражениях, как *while ( not done )*.

## Упрощение сложных выражений

Вы можете предпринять несколько описанных далее шагов для упрощения сложных выражений.

***Разбивайте сложные проверки на части с помощью новых логических переменных*** Вместо создания чудовищных условий с полудюжиной элементов присвойте значения этих элементов промежуточным переменным, что позволит выполнять упрощенную проверку.

***Размещайте сложные выражения в логических функциях*** Если какая-то проверка выполняется неоднократно или отвлекает от основного хода алгоритма программы, поместите ее код в отдельную функцию и проверяйте значение этой функции. Вот пример сложного условия:

**Пример проверки сложного условия (Visual Basic)**

```
If ( ( document.AtEndOfStream ) And ( Not inputError ) ) And _
   ( ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES ) ) And _
   ( Not ErrorProcessing(  ) ) Then
   ' Делаем то или иное.
   ...
End If
```

Это условие выглядит ужасно, и вам приходится его читать, даже если оно вам неинтересно. Поместив его в логическую функцию, вы сможете изолировать эту проверку и позволите читателю забыть о ней, пока она не понадобится. Вот как можно поместить условие *if* в функцию:

**Пример сложного условия, помещенного в логическую функцию и использующего для ясности новые промежуточные переменные (Visual Basic)**

```
Function DocumentIsValid( _
   ByRef documentToCheck As Document, _
   lineCount As Integer, _
   inputError As Boolean _
   ) As Boolean

   Dim allDataRead As Boolean
   Dim legalLineCount As Boolean

Промежуточные переменные добавлены для упрощения проверки в самой последней строке.

   allDataRead = ( documentToCheck.AtEndOfStream ) And ( Not inputError )
   legalLineCount = ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES )
   DocumentIsValid = allDataRead And legalLineCount And ( Not ErrorProcessing() )

End Function
```

**Перекрестная ссылка** О методике использования промежуточных переменных для прояснения логических проверок см. подраздел «Используйте логические переменные для документирования программы» раздела 12.5.

Здесь предполагается, что *ErrorProcessing()* — это некоторая логическая функция, определяющая текущее состояние обработки документа. Теперь, когда вы читаете основной ход алгоритма, вам не надо разбираться со сложным условием:

**Пример основного хода алгоритма, не содержащего сложное условие (Visual Basic)**

```
If ( DocumentIsValid( document, lineCount, inputError ) ) Then
   ' Делаем то или иное.
   ...
End If
```

Если вы проверяете условие только раз, вам может показаться, что его не стоит помещать в отдельный метод. Но вынесение условия в отдельную, хорошо названную функцию улучшит читабельность кода и упростит понимание того, что этот код делает. Это достаточная причина для создания функции. Имя нового метода привносит в программу абстракцию, которая документирует назначение проверки *прямо в коде*. Это даже лучше, чем документирование условия с помощью комментариев, потому что код будет читаться и модифицироваться с большей вероятностью, чем комментарии.

**Перекрестная ссылка** Об использовании таблиц для замены сложной логики см. главу 18.

***Используйте таблицы решений для замены сложных условий*** Иногда нужно проверять сложные условия, содержащие несколько переменных. В этом случае для выполнения проверки удобней применять таблицы решений, а не операторы *if* или *case*. Таблицу решений изначально проще кодировать — она требует пары строк кода и никаких изощренных управляющих структур. Такая минимизация сложности уменьшает возможность ошибок. При изменении данных вы можете изменить таблицу решений, не меняя код: вам всего лишь надо обновить содержимое структуры данных.

## Составление позитивных логических выражений

> Я не не нетупой.
>
> *Гомер Симпсон*
> (*Homer Simpson*)

Не немногие люди не имеют проблем с непониманием некоротких неположительных фраз, т. е. большинство людей имеют трудности с пониманием большого количества отрицаний. Вы можете предпринять какие-то действия, чтобы избежать кодирования сложных отрицательных логических выражений в программе.

***В операторах* if *заменяйте негативные выражения позитивными, меняя местами блоки* if *и* else** Вот пример негативного условия:

**Пример сбивающего с толку отрицательного логического условия (Java)**

```
Здесь оператор отрицания.
if ( !statusOK ) {
   // Делаем что-то.
   ...
}
else {
   // Делаем что-то еще.
   ...
}
```

Это условие можно заменить другим, выраженным положительно:

**Пример более понятного логического условия на Java**

```
Условие в этой строке было заменено на противоположное.
if ( statusOK ) {
   // Делаем что-то еще.
Код в этом блоке был поменян местами...
   ...
}
else {
...с кодом в этом блоке.
   // Делаем что-то.
   ...
}
```

Второй фрагмент кода логически совпадает с первым, но его легче читать, потому что отрицательное выражение было изменено на положительное.

В качестве альтернативы вы можете использовать другие имена переменных, которые изменят истинное значение условия на противоположное. В приведенном примере вы можете заменить переменную *statusOK* на *ErrorDetected*, которая будет истиной, когда *statusOK* будет ложно.

**Перекрестная ссылка** Рекомендация по составлению положительных логических выражений иногда идет вразрез с рекомендацией кодировать номинальный вариант в блоке *if*, а не в блоке *else* (см. раздел 15.1). В этом случае вам нужно взвесить преимущества каждого подхода и решить, какой вариант в вашей ситуации будет наилучшим.

***Применяйте теоремы Деморгана для упрощения логических проверок с отрицаниями*** Теоремы Деморгана позволяют эксплуатировать логическую взаимосвязь между некоторым выражением и версией этого выражения, обозначающего то же самое, благодаря использованию двойного отрицания. Рассмотрим фрагмент кода, содержащий следующее условие:

**Пример условия с отрицанием (Java)**

```
if ( !displayOK || !printerOK ) ...
```

Это условие логически эквивалентно следующему:

**Пример после применения теорем Деморгана (Java)**

```
if ( !( displayOK && printerOK ) ) ...
```

В данном случае вам не надо менять местами блоки *if* и *else* — выражения в двух последних фрагментах кода логически эквивалентны. Для применения теорем Деморгана к логическому оператору *and* или *or* и паре операндов вы инвертируете оба операнда, меняете местами операторы *and* и *or* и инвертируете все выражение целиком. Табл. 19-1 обобщает возможные преобразования в соответствии с теоремами Деморгана.

**Табл. 19-1. Преобразования логических переменных в соответствии с теоремами Деморгана**

| Исходное выражение | Эквивалентное выражение |
|---|---|
| not A and not B | not ( A or B ) |
| not A and B | not ( A or not B ) |
| A and not B | not ( not A or B ) |
| A and B | not ( not A or not B ) |
| not A or not B* | not ( A and B ) |
| not A or B | not ( A and not B ) |
| A or not B | not ( not A and B ) |
| A or B | not ( not A and not B ) |

* Это выражение и используется в примере.

## Использование скобок для пояснения логических выражений

**Перекрестная ссылка** Примеры применения скобок для прояснения других видов выражений см. в подразделе «Скобки» раздела 31.2.

Если у вас есть сложное логическое выражение, не надейтесь на порядок вычисления его операндов в языке программирования — используйте скобки, чтобы сделать ваши намерения понятными. Скобки уменьшают требования, предъявляемые к читателю кода, который может и не разобраться в тонкостях вычисления логических выражений в вашем языке программирования. Если вы благоразумны, то не будете полагаться на собственную или чью-то еще способность правильно запомнить приоритет вычислений, особенно если приходится переключаться между двумя или более языками. Использование скобок не похоже на отправку телеграммы: вы не платите за каждую букву — дополнительные символы бесплатны.

Вот пример выражения, содержащего слишком мало скобок:

**Пример выражения, содержащего слишком мало скобок (Java)**

```
if ( a < b == c == d ) ...
```

Начнем с того, что это выражение слишком запутано. Оно тем более сбивает с толку, что не ясно, хотел ли кодировщик проверить условие *( a < b ) == ( c == d )* или *( ( a < b ) == c ) == d*. Следующая версия все равно не идеальна, но скобки все же помогают:

**Пример выражения, частично улучшенного с помощью скобок (Java)**

```
if ( ( a < b ) == ( c == d ) ) ...
```

В этом случае скобки повышают удобство чтения и корректность программы, поскольку компилятор не истолковал бы первый фрагмент таким способом. Когда сомневаетесь, используйте скобки.

***Используйте простой метод подсчета для проверки симметричности скобок*** Если у вас возникают проблемы с поиском парных скобок, то вот простой способ подсчета. Начните считать, сказав «ноль». Двигайтесь вдоль выраже-

ния слева направо. Встретив открывающую скобку, скажите «один». Каждый раз при встрече открывающей скобки увеличивайте число. А встречая закрывающую скобку, уменьшайте это число. Если к концу выражения у вас опять получится 0, то ваши скобки симметричны.

**Перекрестная ссылка** Многие текстовые редакторы, ориентированные на программистов, предоставляют команды для поиска парных круглых, квадратных и фигурных скобок. О редакторах для программирования см. подраздел «Редактирование» раздела 30.2.

**Пример симметричных скобок (Java)**

```
Читаем это выражение.
if ( ( ( a < b ) == ( c == d ) ) && !done ) ...
   | | |       |    |        | |          |
0  1 2 3       2    3        2 1          0
Произносим эти числа.
```

В этом примере в конце получился 0, следовательно, скобки симметричны. В следующем примере количество отрывающих и закрывающих скобок не одинаково:

**Пример несимметричных скобок (Java)**

```
Читаем это выражение.
if ( ( a < b ) == ( c == d ) ) && !done ) ...
   | |       |    |        | |          |
0  1 2       1    2        1 0         -1
Произносим эти числа.
```

Значение 0, полученное до последней закрывающей скобки, подсказывает, что где-то до этой точки была пропущена скобка. Вы не должны получить 0, не достигнув последней скобки в выражении.

***Заключайте в скобки логическое выражение целиком*** Скобки ничего вам не стоят, а читабельность улучшают. Привычка заключать в скобки все логическое выражение целиком — хорошая практика программирования.

## Понимание правил вычисления логических выражений

Множество языков содержит неявную управляющую форму, которая начинает действовать при вычислении логических выражений. Компиляторы некоторых языков вычисляют каждый элемент логического выражения перед объединением всех этих элементов и вычисления значения всего выражения. Компиляторы других используют «короткозамкнутый» (или «ленивый») алгоритм, обрабатывая только необходимые элементы выражения. Это особенно важно, когда в зависимости от результатов первой проверки вы можете не захотеть выполнять следующий тест. Допустим, вы проверяете элементы массива с помощью следующего выражения:

**Пример псевдокода неправильной проверки условия**

```
while ( i < MAX_ELEMENTS and item[ i ] <> 0 ) ...
```

Если вычисляется выражение целиком, вы получите ошибку при последней итерации цикла. В этом случае переменная *i* равна *maxElements*, а значит, выражение *item[ i ]* эквивалентно *item[ maxElements ]*, что является недопустимым значением

индекса. Вы можете возразить, что это не имеет значения, поскольку вы только обращаетесь к элементу, а не изменяете его. Но это неряшество способно сбить с толку читателя вашего кода. Во многих средах этот код будет также генерировать ошибку выполнения или нарушение защиты.

Используя псевдокод, можно реструктурировать данное условие так, чтобы эта ошибка не возникала:

**Пример псевдокода правильно реструктурированной проверки условия**

```
while ( i < MAX_ELEMENTS )
   if ( item[ i ] <> 0 ) then
      ...
```

Этот вариант корректен, так как *item[ i ]* будет вычисляться, только когда *i* меньше, чем *maxElements*.

Многие современные языки предоставляют средства, которые изначально предотвращают возможность возникновения такой ошибки. Так, C++ использует короткозамкнутые вычисления: если значение первого операнда в операции *and* ложно, то второй операнд не вычисляется, потому что полное выражение в любом случае будет ложным. Иначе говоря, в C++ единственный элемент выражения:

```
if ( SomethingFalse && SomeCondition ) ...
```

который будет вычисляться, — это *SomethingFalse*. Обработка выражения завершается, поскольку значение *SomethingFalse* определяется как ложное.

Аналогичное укороченное вычисление будет производиться и для оператора *or*. В C++ и Java в выражении:

```
if ( somethingTrue || someCondition ) ...
```

вычисляется только *somethingTrue*. Обработка завершается, как только операнд *somethingTrue* определяется как истинный, так как все выражение будет истинным, если истинна хотя бы одна из его частей. В результате такого способа вычисления следующее выражение вполне допустимо:

**Пример условия, которое работает благодаря короткозамкнутому вычислению (Java)**

```
if ( ( denominator != 0 ) && ( ( item / denominator ) > MIN_VALUE ) ) ...
```

Если бы это выражение вычислялось целиком, то в случае, когда переменная *denominator* равна *0*, операция деления во втором операнде генерировала бы ошибку деления на 0. Но поскольку вторая часть не вычисляется, если значение первой ложно, то когда *denominator* равен *0*, вторая операция не выполняется, и ошибка деления на 0 не возникает.

С другой стороны, из-за того что операция *&& (and)* вычисляется слева направо, следующее логически эквивалентное выражение работать не будет:

**Пример условия, в котором короткозамкнутое вычисление не спасает от ошибки (Java)**

```
if ( ( ( item / denominator ) > MIN_VALUE ) && ( denominator != 0 ) ) ...
```

Здесь *item / denominator* вычисляется раньше, чем *denominator != 0*. Следовательно, в этом коде происходит ошибка деления на 0.

Язык Java еще более усложняет эту картину, предоставляя «логические» операторы. Логические операторы & и | языка Java гарантируют, что все элементы будут вычислены полностью независимо от того, определяется ли истинность или ложность выражения без его полного вычисления. Иначе говоря, в Java такое условие будет безопасно:

**Пример условия, которое работает благодаря короткозамкнутому (условному) вычислению (Java)**

```
if ( ( denominator != 0 ) && ( ( item / denominator ) > MIN_VALUE ) ) ...
```

А вот такое — нет:

**Пример условия, которое не будет работать, потому что короткозамкнутое вычисление не гарантируется (Java)**

```
if ( ( denominator != 0 ) & ( ( item / denominator ) > MIN_VALUE ) ) ...
```

Разные языки используют разные способы вычисления, и случается, что разработчики языка чересчур свободно обращаются с правилами вычисления выражений, поэтому обратитесь к руководству по вашей версии языка, чтобы выяснить, как в нем выполняются эти операции. Еще лучше (поскольку читатель может не обладать вашей сообразительностью) использовать вложенные условия, проясняющие ваши намерения, и не зависеть от порядка обработки выражений и короткозамкнутых вычислений.

## Упорядочение числовых выражений в соответствии со значениями на числовой прямой

Организуйте числовые условия так, чтобы они соответствовали порядку точек на числовой прямой. В общем случае структурируйте выражения так, чтобы сравнения выглядели следующим образом:

```
MIN_ELEMENTS <= i and i <= MAX_ELEMENTS
i < MIN_ELEMENTS or MAX_ELEMENTS < i
```

Идея в том, чтобы расположить элементы по порядку слева направо, от наименьших к наибольшим. В первой строке *MIN_ELEMENTS* и *MAX_ELEMENTS* — это две граничные точки, и поэтому они расположены по краям выражения. Подразумевается, что переменная *i* должна находиться между ними, и поэтому она расположена в середине. Во втором примере вы проверяете, находится ли *i* вне диапазона, поэтому *i* расположена по краям условия, а *MIN_ELEMENTS* и *MAX_ELEMENTS* — посредине. Этот подход позволяет легко создать визуальное представление сравнения (рис. 19-1):

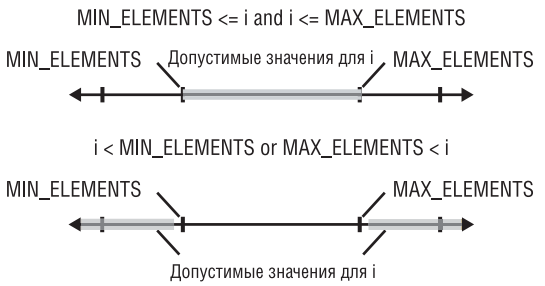



***Рис. 19-1.*** *Примеры упорядочения логических условий в соответствии с числовой прямой*

Если вы сравниваете значение *i* только с *MIN_ELEMENTS*, то расположение *i* зависит от того, каким должно быть *i* в случае успешной проверки условия. Если предполагается, что *i* меньше, то условие выглядит так:

```
while ( i < MIN_ELEMENTS ) ...
```

Но если *i* должно быть больше, вы получаете:

```
while ( MIN_ELEMENTS < i ) ...
```

Этот подход более очевиден, чем такие условия, как:

```
( i > MIN_ELEMENTS ) and ( i < MAX_ELEMENTS )
```

которые не позволяют читателю визуализировать проверяемое выражение.

## Общие принципы сравнения с *0*

Языки программирования используют *0* для нескольких целей. Это числовое значение. Это нулевой терминатор в строке. Это пустой указатель. Это значение первого элемента перечисления. Это *false* в логических выражениях. А посему вам следует писать такой код, чтобы конкретное назначение *0* было очевидно.

***Неявно сравнивайте логические переменные*** Как указывалось ранее, целесообразно писать логические выражения в виде:

```
while ( !done ) ...
```

Это неявное сравнение с *0* допустимо, потому что выполняется в логическом выражении.

***Сравнивайте числа с 0*** Хотя в логических выражениях неявное сравнение с *0* допустимо, числовые выражения следует сравнивать явно. Для чисел пишите:

```
while ( balance != 0 ) ...
```

а не:

```
while ( balance ) ...
```

***В языке C сравнивайте символы с нулевым терминатором ('\0') явно*** Символы, как и числа, не являются логическими выражениями. Поэтому для символов следует писать:

```
while ( *charPtr != '\0' ) ...
```

а не:

```
while ( *charPtr ) ...
```

Эта рекомендация расходится с широко распространенным соглашением языка C по обработке символьных данных (так, как показано во втором примере). Но первый способ усиливает идею, что выражение имеет дело с символьными, а не с логическими данными. Некоторые соглашения C не основываются на максимизации удобства чтения или сопровождения, и это — один из таких примеров. К счастью, весь этот вопрос растворяется в лучах заката, поскольку все больше кода пишется с использованием строк C++ и STL.

***Сравнивайте указатели с*** **NULL** Для указателей пишите:

```
while ( bufferPtr != NULL ) ...
```

а не:

```
while ( bufferPtr ) ...
```

Как и в случае с символьными данными, эта рекомендация расходится с устоявшимся соглашением языка C, но преимущество в читабельности ее оправдывает.

## Общие проблемы с логическими выражениями

Логические выражения содержат еще несколько ловушек, специфичных для некоторых языков программирования:

***В C-подобных языках, помещайте константы с левой стороны сравнений*** В C-подобных языках возникают некоторые специфические проблемы с логическими выражениями. Если время от времени вы набираете = вместо ==, подумайте об использовании соглашения по размещению констант и литералов с левой стороны выражений, например, так:

**Пример размещения константы с левой стороны выражения на C++. Это позволит компилятору обнаружить ошибку**

```
if ( MIN_ELEMENTS = i ) ...
```

В этом выражении компилятор обязан объявить одиночный знак = ошибкой, поскольку недопустимо присваивать какое-нибудь значение константе. А в следующем выражении компилятор, напротив, выдаст только предупреждение, и лишь в том случае, если у вас включены все предупреждения компилятора:

**Пример размещения константы с правой стороны выражения на C++. Эту ошибку компилятор может и не обнаружить**

```
if ( i = MIN_ELEMENTS ) ...
```

Эта рекомендация конфликтует с предложением упорядочивать значения на числовой прямой. Я лично предпочитаю использовать упорядочение, а компилятор пусть предупреждает меня о непреднамеренных присваиваниях.

***В C++ рассмотрите вопрос подстановки макросов препроцессора вместо операторов &&, || и == (но только как последнее средство)*** Если у вас возникают такие трудности, то можно создать макросы *#define* для логических операций *and* и *or* и использовать *AND* и *OR* вместо *&&* и ||. Точно так же очень легко написать =, имея в виду ==. Если вы часто с этим сталкиваетесь, можете создать макрос *EQUALS* для логического равенства(==).

Многие опытные программисты рассматривают этот подход как упрощающий работу тем разработчикам, которые еще не освоились с языком программирования. А тем, кто уже хорошо знает язык, эта методика будет только мешать. Кроме того, многие компиляторы генерируют предупреждающие сообщения, если использование присваивания и битовых операций выглядит как ошибка. Включение всех предупреждений компилятора часто является лучшим вариантом, чем создание нестандартных макросов.

***В языке Java учитывайте разницу между* a==b *и* a.equals(b)** В Java *a==b* проверяет, ссылаются ли *a* и *b* на один и тот же объект, тогда как *a.equals(b)* проверяет, имеют ли объекты одно и то же логическое значение. В общем случае программы на Java должны использовать выражения вроде *a.equals(b)*, а не *a==b*.

## 19.2. Составные операторы (блоки)

«Составной оператор» или «блок» — это набор операторов, который с точки зрения управления ходом программы рассматривается как один оператор. Составные операторы создаются с помощью добавления скобок *{* и *}* вокруг группы выражений в языках C++, C#, C и Java. Иногда они задаются с помощью ключевых слов команды, как, например, *For* и *Next* в Visual Basic.

**Перекрестная ссылка** Многие текстовые редакторы, ориентированные на программиста, содержат команды для поиска парных круглых, квадратных и фигурных скобок (см. подраздел «Редактирование» раздела 30.2).

***Пишите обе скобки одновременно*** Заполняйте внутреннее содержимое по-сле того, как напишете открывающую и закрывающую часть блока. Люди часто жалуются, как тяжело не ошибиться с парными скобками или словами *begin-end*, хотя это совершенно излишняя проблема. Если вы последуете этому совету, у вас никогда больше не будет трудностей в поиске соответствующих пар скобок.

Сначала напишите:

```
for ( i = 0; i < maxLines; i++ )
```

Потом:

```
for ( i = 0; i < maxLines; i++ ) {   }
```

И наконец:

```
for ( i = 0; i < maxLines; i++ ) {
   // Все что угодно   ...
}
```

Это относится ко всем блоковым структурам, включая операторы *if, for* и *while* в C++ и Java и сочетания *If-Then-Else*, *For-Next* и *While-Wend* в Visual Basic.

***Используйте скобки для пояснения условных операторов*** Довольно сложно разобраться в коде условного оператора, не разбираясь в действиях, выполняемых в результате проверки условия *if*. Размещение единственного оператора после *if* иногда выглядит эстетически правильным, но в процессе сопровождения такие выражения часто превращаются в более сложные блоки, и одиночный оператор в таких случаях может привести к ошибке.

Для пояснения ваших намерений используйте блок независимо от того, сколько в нем строк кода: 1 или 20.

## 19.3. Пустые выражения

В C++ допустимо использования пустого оператора, состоящего исключительно из точки с запятой, например:

**Пример традиционного пустого выражения (C++)**

```
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() )
   ;
```

В C++ после *while* должно следовать выражение, но оно может быть пустым. Отдельная точка с запятой и является пустым оператором. Далее приведены принципы работы с пустыми выражениями в C++:

***Привлекайте внимание к пустым выражениям*** Пустые выражения не являются широко распространенными, поэтому сделайте их очевидными. Один из способов — выделить точке с запятой, представляющей собой пустой оператор, отдельную строку. Этот подход показан в предыдущем примере. В качестве альтернативы можно использовать пустые скобки, чтобы подчеркнуть это выражение. Приведем два примера:

**Перекрестная ссылка** Возможно, лучший способ обрабатывать пустые операторы — это избегать их (см. подраздел «Избегайте пустых циклов» раздела 16.2).

**Примеры выделения пустых выражений (C++)**

```
Это один из способов выделить пустое выражение.
while ( recordArray.Read( index++ ) ) != recordArray.EmptyRecord() ) {}

Это еще один способ сделать это.
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() ) {
   ;
}
```

***Создайте для пустых выражений макрос препроцессора или встроенную функцию*** **DoNothing()** Это выражение не делает ничего, кроме бесспорного подтверждения того факта, что никакие действия предприниматься не должны. Это похоже на пометку пустых страниц документа фразами «Эта страница намеренно оставлена пустой». На самом деле страница не совсем пустая, но вы знаете, что ничего другого на ней быть не должно.

Вот как создать собственный пустой оператор в C++ с помощью *#define*. (Вы также можете создать *inline*-функцию, которая дает тот же эффект.)

**Пример пустого выражения, выделенного с помощью *DoNothing()* (C++)**

```
#define DoNothing()
...
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() ) {
   DoNothing();
}
```

В дополнение к использованию *DoNothing()* в пустых циклах *while* и *for* можно задействовать ее в несущественных вариантах оператора *switch* — добавление *DoNothing()* делает очевидным тот факт, что вариант был рассмотрен и никаких действий предприниматься не должно.

Если ваш язык не поддерживает макросы препроцессора или встроенные функции, вы можете создать обычный метод *DoNothing()*, который сразу будет возвращать управление вызывающей стороне.

***Подумайте, не будет ли код яснее с непустым телом цикла*** Большая часть циклов с пустым телом полагается на побочный эффект в управляющем выражении цикла. В большинстве случаев код будет читабельнее, если эти побочные действия будут выполняться явно, например:

**Пример более очевидного цикла с непустым телом (C++)**

```
RecordType record = recordArray.Read( index );
index++;
while ( record != recordArray.EmptyRecord() ) {
   record = recordArray.Read( index );
   index++;
}
```

Этот подход требует дополнительной переменной, управляющей циклом, а также большего количества строк, но он делает акцент на простоте программирования, а не на остроумном использовании побочных эффектов. В промышленном коде такой акцент предпочтительней.

# 19.4. Укрощение опасно глубокой вложенности

Чрезмерные отступы (или «вложенность») осуждаются в компьютерной ли тературе уже на протяжении 25 лет и все еще являются главными обвиняемыми в создании запутанного кода. В работах Ноума Чомски и Джеральда Вейнберга (Noam Chomsky and Gerald Weinberg) высказывалось предположение, что немногие люди способны понять более трех уровней вложенных *if* (Yourdon, 1986a), и многие исследователи рекомендуют избегать вложенности, превышающей три или четыре уровня (Myers, 1976; Marca, 1981; Ledgard and Tauer, 1987a). Глубокая вложенность противоречит описанному в главе 5 Главному Техническому Императиву ПО (управлению сложностью). Это достаточная причина для отказа от глубокой вложенности.

Избавиться от глубокой вложенности несложно. Для этого вы можете переписать проверки условий, выполняемые в блоках *if* и *else*, или разбить код на более простые методы.

***Упростите вложенные* if *с помощью повторной проверки части условия***
Если вложенность становится слишком глубокой, вы можете уменьшить количество ее уровней, повторно проверив некоторые условия. Глубина вложенности в этом примере кода является достаточным основанием для его реструктуризации:

**Пример плохого, глубоко вложенного кода (C++)**

**Перекрестная ссылка** Повторная проверка части условия для уменьшения сложности аналогична повторному тестированию статусной переменной. Такой способ демонстрируется в подразделе «Обработка ошибок и операторы *goto*» раздела 17.3.

```
if ( inputStatus == InputStatus_Success ) {
   // Много кода.
   ...
   if ( printerRoutine != NULL ) {

      // Много кода.
      ...
      if ( SetupPage() ) {
         // Много кода.
         ...
         if ( AllocMem( &printData ) ) {
            // Много кода.
            ...
         }
      }
   }
}
```

Этот пример придуман для демонстрации уровней вложенности. Части, обозначенные как *// Много кода*, подразумевают, что метод содержит достаточно много строк и простирается на нескольких экранах или нескольких страницах напечатанного листинга. Вот как можно видоизменить этот код, используя повторные проверки, а не вложенность:

**Пример кода, милосердно избавленного от вложенности с помощью повторных проверок (C++)**

```
if ( inputStatus == InputStatus_Success ) {
   // Много кода.
   ...
   if ( printerRoutine != NULL ) {
      // Много кода.
      ...
   }
}

if ( ( inputStatus == InputStatus_Success ) &&
   ( printerRoutine != NULL ) && SetupPage() ) {
   // Много кода.
   ...
```

```
   if ( AllocMem( &printData ) ) {
      // Много кода.
      ...
   }
}
```

Это чрезвычайно реалистичный пример, так как показывает, что вы не можете уменьшить уровень вложенности безнаказанно, взамен вам придется формировать более сложный условия. Однако уменьшение с четырех до двух уровней вложенности дает большое улучшение в читабельности, поэтому такой способ стоит принять во внимание.

***Упростите вложенные* if *с помощью блока с выходом*** Альтернативой к только что описанному подходу будет создание фрагмента кода, который будет выполняться как блок. Если одно из условий в середине блока не выполнится, остаток блока будет пропущен.

#### Пример использования блока с выходом (C++)

```
do {
   // Начало блока с выходом.
   if ( inputStatus != InputStatus_Success ) {
      break; // Выходим из блока.
   }
   // Много кода.
   ...
   if ( printerRoutine == NULL ) {
      break; // Выходим из блока.
   }

   // Много кода.
   ...
   if ( !SetupPage() ) {
      break; // Выходим из блока.
   }

   // Много кода.
   ...
   if ( !AllocMem( &printData ) ) {
      break; // Выходим из блока.
   }

   // Много кода.
   ...
} while (FALSE); // Конец блока с выходом
```

Этот способ довольно необычен, поэтому его следует использовать, только если вся ваша команда разработчиков с ним знакома и он одобрен в качестве подходящей практики кодирования.

***Преобразуйте вложенные* if *в набор* if-then-else** Если вы критически относитесь к вложенными условиями *if*, вам будет интересно узнать, что вы можете ре-

организовать эти конструкции так, чтобы использовать операторы *if-then-else* вместо вложенных *if*. Допустим, у вас есть развесистое дерево решений вроде этого:

**Пример заросшего дерева решений (Java)**

```
if ( 10 < quantity ) {
   if ( 100 < quantity ) {
      if ( 1000 < quantity ) {
         discount = 0.10;
      }
      else {
         discount = 0.05;
      }
   }
   else {
      discount = 0.025;
   }
}
else {
   discount = 0.0;
}
```

Этот фрагмент имеет много недостатков, один из которых в том, что проверяемые условия избыточны. Когда вы удостоверились, что значение *quantity* больше *1000*, вам не нужно дополнительно проверять, что оно больше *100* и больше *10*. А значит, вы можете преобразовать этот код таким образом:

**Пример вложенных *if*, сконвертированных в набор *if-then-else* (Java)**

```
if ( 1000 < quantity ) {
   discount = 0.10;
}
else if ( 100 < quantity ) {
   discount = 0.05;
}
else if ( 10 < quantity ) {
   discount = 0.025;
}
else {
   discount = 0;
}
```

Это решение проще, чем могло бы быть, потому что закономерность увеличения чисел проста. Вот как изменить вложенные *if*, если бы числа не были так упорядочены:

**Пример вложенных *if*, преобразованных в набор *if-then-else*, для случая, когда числа не упорядочены (Java)**

```
if ( 1000 < quantity ) {
   discount = 0.10;
}
```

```
else if ( ( 100 < quantity ) && ( quantity <= 1000 ) ) {
   discount = 0.05;
}
else if ( ( 10 < quantity ) && ( quantity <= 100 ) ) {
   discount = 0.025;
}
else if ( quantity <= 10 ) {
   discount = 0;
}
```

Основное различие между этим и предыдущим примером в том, что выражения в условиях *else-if* не полагаются на предыдущие проверки. Этот код не требует выполнения блока *else*, и проверки фактически могут выполняться в любом порядке. Фрагмент мог бы состоять из четырех *if* и не включать ни одного *else*. Единственная причина, по которой версия с *else* предпочтительней, — это отказ от ненужных повторных проверок.

***Преобразуйте вложенные* if *в оператор* case** Некоторые виды проверок условий, особенно использующие целые числа, можно перекодировать, применяя оператор *case* вместо последовательностей *if* и *else*. В некоторых языках вы не сможете использовать эту методику, но там, где это возможно, она очень удобна. Вот как преобразовать тот же пример на Visual Basic:

**Пример преобразования вложенных *if* к оператору *case* (Visual Basic)**

```
Select Case quantity
   Case 0 To 10
      discount = 0.0
   Case 11 To 100
      discount = 0.025
   Case 101 To 1000
      discount = 0.05
   Case Else
      discount = 0.10
End Select
```

Пример читается, как стихи. Если вы сравните его с двумя приведенными ранее — многочисленными отступами, он покажется особенно понятным решением.

***Факторизуйте глубоко вложенный код в отдельный метод*** Если глубокая вложенность создается внутри цикла, вы зачастую можете улучшить ситуацию, переместив содержимое цикла в отдельный метод. Это особенно эффективно, если вложенность является результатом как проверок условий, так и итераций. Оставьте блоки *if-then-else* в основном цикле, чтобы показать ветвление решения, а содержимое этих блоков переместите в новые методы. Следующий код нуждается в такой модификации:

**Пример вложенного кода, требующего разбиения на методы (C++)**

```
while ( !TransactionsComplete() ) {
   // Читаем транзакционную запись.
   transaction = ReadTransaction();
```

```
    // Обрабатываем транзакцию в зависимости от ее типа.
    if ( transaction.Type == TransactionType_Deposit ) {
        // Обрабатываем транзакцию-вклад.
        if ( transaction.AccountType == AccountType_Checking ) {
            if ( transaction.AccountSubType == AccountSubType_Business )
                MakeBusinessCheckDep( transaction.AccountNum, transaction.Amount );
            else if ( transaction.AccountSubType == AccountSubType_Personal )
                MakePersonalCheckDep( transaction.AccountNum, transaction.Amount );
            else if ( transaction.AccountSubType == AccountSubType_School )
                MakeSchoolCheckDep( transaction.AccountNum, transaction.Amount );
        }
        else if ( transaction.AccountType == AccountType_Savings )
            MakeSavingsDep( transaction.AccountNum, transaction.Amount );
        else if ( transaction.AccountType == AccountType_DebitCard )
            MakeDebitCardDep( transaction.AccountNum, transaction.Amount );
        else if ( transaction.AccountType == AccountType_MoneyMarket )
            MakeMoneyMarketDep( transaction.AccountNum, transaction.Amount );
        else if ( transaction.AccountType == AccountType_Cd )
            MakeCDDep( transaction.AccountNum, transaction.Amount );
    }
    else if ( transaction.Type == TransactionType_Withdrawal ) {
        // Обрабатываем снятие денег.
        if ( transaction.AccountType == AccountType_Checking )
            MakeCheckingWithdrawal( transaction.AccountNum, transaction.Amount );
        else if ( transaction.AccountType == AccountType_Savings )
            MakeSavingsWithdrawal( transaction.AccountNum, transaction.Amount );
        else if ( transaction.AccountType == AccountType_DebitCard )
            MakeDebitCardWithdrawal( transaction.AccountNum, transaction.Amount );
    }
```

Вот транзакция перевода — *TransactionType_Transfer*.

```
    else if ( transaction.Type == TransactionType_Transfer ) {
        MakeFundsTransfer(
            transaction.SourceAccountType,
            transaction.TargetAccountType,
            transaction.AccountNum,
            transaction.Amount
        );
    }
    else {
        // Обрабатываем неизвестный тип транзакции.
        LogTransactionError( "Unknown Transaction Type", transaction );
    }
}
```

Этот код сложен, но бывает и хуже. Он имеет всего четыре уровня вложенности, содержит комментарии и логические отступы, а его функциональная декомпозиция достаточно адекватна, особенно для типа транзакции *TransactionType_Transfer*. И все же вы можете улучшить этот код, вынеся содержимое внутренних *if*-проверок в отдельные методы.

**Пример правильно вложенного кода после декомпозиции на методы (C++)**

```
while ( !TransactionsComplete() ) {
   // Читаем транзакционную запись.
   transaction = ReadTransaction();

   // Обрабатываем транзакцию в зависимости от ее типа.
   if ( transaction.Type == TransactionType_Deposit ) {
      ProcessDeposit(
         transaction.AccountType,
         transaction.AccountSubType,
         transaction.AccountNum,
         transaction.Amount
      );
   }
   else if ( transaction.Type == TransactionType_Withdrawal ) {
      ProcessWithdrawal(
         transaction.AccountType,
         transaction.AccountNum,
         transaction.Amount
      );
   }
   else if ( transaction.Type == TransactionType_Transfer ) {
      MakeFundsTransfer(
         transaction.SourceAccountType,
         transaction.TargetAccountType,
         transaction.AccountNum,
         transaction.Amount
      );
   }
   else {
      // Обрабатываем неизвестный тип транзакции.
      LogTransactionError("Unknown Transaction Type", transaction );
   }
}
```

**Перекрестная ссылка** Этот способ функциональной декомпозиции особенно прост, если вы изначально строили методы по методике, описанной в главе 9. О принципах функциональной декомпозиции см. подраздел «Разделяй и властвуй» раздела 5.4.

Код новых методов просто был изъят из исходного фрагмента и оформлен в виде новых методов (они здесь не показаны). Такой код имеет несколько преимуществ. Во-первых, двухуровневая вложенность делает структуру проще и понятнее. Во-вторых, вы можете читать, исправлять и отлаживать более короткий цикл *while*, помещающийся на одном экране — не требуется переходить между экранами или страницами напечатанного текста. В-третьих, при вынесении функциональности в методы *ProcessDeposit()* и *ProcessWithdrawal()* приобретаются все остальные преимущества модульности. В-четвертых, теперь легко можно увидеть, что этот код может быть преобразован в оператор *case*, что еще более упростит чтение:

**Пример правильно вложенного кода после декомпозиции и использования оператора *case* (C++)**

```
while ( !TransactionsComplete() ) {
   // Читаем транзакционную запись.
   transaction = ReadTransaction();

   // Обрабатываем транзакцию в зависимости от ее типа.
   switch ( transaction.Type ) {
      case ( TransactionType_Deposit ):
         ProcessDeposit(
            transaction.AccountType,
            transaction.AccountSubType,
            transaction.AccountNum,
            transaction.Amount
            );
         break;

      case ( TransactionType_Withdrawal ):
         ProcessWithdrawal(
            transaction.AccountType,
            transaction.AccountNum,
            transaction.Amount
            );
         break;

      case ( TransactionType_Transfer ):
         MakeFundsTransfer(
            transaction.SourceAccountType,
            transaction.TargetAccountType,
            transaction.AccountNum,
            transaction.Amount
            );
         break;

      default:
         // Обрабатываем неизвестный тип транзакции.
         LogTransactionError("Unknown Transaction Type", transaction );
         break;
   }
}
```

***Используйте более объектно-ориентированный подход*** Самый прямой подход для упрощения именно этого кода в объектно-ориентированной среде состоит в создании абстрактного базового класса *Transaction* и его подклассов *Deposit*, *Withdrawal* и *Transfer*.

**Пример хорошего кода, использующего полиморфизм (C++)**

```
TransactionData  transactionData;
Transaction  *transaction;

while ( !TransactionsComplete() ) {
   // Читаем транзакционную запись.
   transactionData = ReadTransaction();

   // Создаем объект транзакции в зависимости от ее типа.
   switch ( transactionData.Type ) {
      case ( TransactionType_Deposit ):
         transaction = new Deposit( transactionData );
         break;

      case ( TransactionType_Withdrawal ):
         transaction = new Withdrawal( transactionData );
         break;

      case ( TransactionType_Transfer ):
         transaction = new Transfer( transactionData );
         break;

      default:
         // Обрабатываем неизвестный тип транзакции.
         LogTransactionError("Unknown Transaction Type", transactionData );
         return;
   }
   transaction->Complete();
   delete transaction;
}
```

В системах любого размера оператор *switch* можно заменить вызовом специального метода фабрики объекта, который может повторно использоваться в любом месте, где нужно создать объект типа *Transaction*. Если бы этот код принадлежал такой системе, то он мог бы стать еще проще:

**Пример хорошего кода, использующего полиморфизм и фабрику объекта (C++)**

```
TransactionData  transactionData;
Transaction  *transaction;

while ( !TransactionsComplete() ) {
   // Читаем транзакционную запись и выполняем транзакцию.
   transactionData = ReadTransaction();
   transaction = TransactionFactory.Create( transactionData );
   transaction->Complete();
   delete transaction;
}
```

Чтобы вам было понятно, код метода *TransactionFactory.Create()* представляет собой простую адаптацию оператора *switch* из предыдущего примера:

**Перекрестная ссылка** Дополнительные полезные улучшения кода наподобие этого см. в главе 24.

**Пример хорошего кода для фабрики объекта (C++)**

```
Transaction  *TransactionFactory::Create(
   TransactionData transactionData
   ) {

   // Создаем объект транзакции на основе ее типа.
   switch ( transactionData.Type ) {
      case ( TransactionType_Deposit ):
         return new Deposit( transactionData );
         break;

      case ( TransactionType_Withdrawal ):
         return new Withdrawal( transactionData );
         break;

      case ( TransactionType_Transfer ):
         return new Transfer( transactionData );
         break;

      default:
         // Обрабатываем неизвестный тип транзакции.
         LogTransactionError( "Unknown Transaction Type", transactionData );
         return NULL;
   }
}
```

***Перепроектируйте глубоко вложенный код*** Некоторые эксперты утверждают, что операторы *case* в объектно-ориентированном программировании практически всегда сигнализируют о плохо факторизованном коде и должны использоваться как можно реже (Meyer, 1997). И показанное преобразование из операторов *case*, вызывающих методы, к фабрике объекта с вызовами полиморфных методов — один из таких примеров.

Обобщая, можно сказать, что сложный код — это признак того, что вы недостаточно хорошо понимаете свою программу, чтобы сделать ее простой. Глубокая вложенность — это знак, предупреждающий о том, что нужно добавить вызов метода или перепроектировать сложную часть кода. Это не значит, что вы обязаны переписать весь метод, но у вас должна быть веская причина не делать этого.

## Сводка методик уменьшения глубины вложенности

Далее перечислены способы, позволяющие уменьшить вложенность. Рядом указаны ссылки на разделы этой книги, в которых эти способы обсуждаются:

- повторная проверка части условия (этот раздел);
- конвертирование в блоки *if-then-else* (этот раздел);

- преобразование к оператору *case* (этот раздел);
- факторизация глубоко вложенного кода в отдельный метод (этот раздел);
- использование объектной и полиморфной диспетчеризации (этот раздел);
- изменение кода с целью использования статусной переменной (раздел 17.3);
- использование сторожевых операторов для выхода из метода и пояснения номинального хода алгоритма (раздел 17.1);
- использование исключений (раздел 8.4);
- полное перепроектирование глубоко вложенного кода (этот раздел).

## 19.5. Основа программирования: структурное программирование

Термин «структурное программирование» был введен в исторической статье «Structured Programming», представленной Эдсжером Дейкстрой на конференции НАТО по разработке ПО в 1969 году (Dijkstra, 1969). С тех самых пор термин «структурный» применялся к любой деятельности в области разработки ПО, включая структурный анализ, структурный дизайн и структурное валяние дурака. Различные структурные методики не имели между собой ничего общего, кроме того, что все они создавались в то время, когда слово «структурный» придавало им большую значимость.

Суть структурного программирования состоит в простой идее: программа должна использовать управляющие конструкции с одним входом и одним выходом. Такая конструкция представляет собой блок кода, в котором есть только одно место, где он может начинаться, и одно — где может заканчиваться. У него нет других входов и выходов. Структурное программирование — это не то же самое, что и структурное проектирование сверху вниз. Оно относится только к уровню кодирования.

Структурная программа пишется в упорядоченной, дисциплинированной манере и не содержит непредсказуемых переходов с места на место. Вы можете читать ее сверху вниз, и практически так же она выполняется. Менее дисциплинированные подходы приводят к такому исходному коду, который содержит менее понятную и удобную для чтения картину того, как программа выполняется. Меньшая читабельность означает худшее понимание и в конце концов худшее качество программы.

Главные концепции структурного программирования, касающиеся вопросов использования *break, continue, throw, catch, return* и других тем, применимы до сих пор.

### Три компонента структурного программирования

В следующих разделах описаны три конструкции, составляющие основу структурного программирования.

#### Последовательность

**Перекрестная ссылка** Об использовании последовательностей см. главу 14.

Последовательность — это набор операторов, выполняющихся по порядку. Типичные последовательные операторы содержат присваивания и вызовы методов. Вот два примера:

### Примеры последовательного кода (Java)

```
// Последовательность операторов присваивания.
a = “1”;
b = “2”;
c = “3”;

// Последовательность вызовов методов.
System.out.println( a );
System.out.println( b );
System.out.println( c );
```

## Выбор

**Перекрестная ссылка** Об использовании выбора см. главу 15.

Выбор — это такая управляющая конструкция, которая заставляет операторы выполняться избирательно. Наиболее частый пример — выражение *if-then-else*. Выполняется либо блок *if-then*, либо *else*, но не оба сразу. Один из блоков «выбирается» для выполнения.

Оператор *case* — другой пример управляющего элемента выбора. Оператор *switch* в C++ и Java, оператор *select* — все это примеры *case*. В каждом случае для выполнения выбирается один из вариантов. Концептуально операторы *if* и *case* похожи. Если ваш язык не поддерживает операторы *case*, вы можете эмулировать их с помощью набора *if*. Вот два примера выбора:

### Пример выбора (Java)

```
// Выбор в операторе if.
if ( totalAmount > 0.0 ) {
   // Делаем что-то.
   ...
}
else {
   // Делаем что-то еще.
   ...
}

// Выбор в операторе case.
switch ( commandShortcutLetter ) {
   case ‘a’:
      PrintAnnualReport();
      break;
   case ‘q’:
      PrintQuarterlyReport();
      break;
   case ‘s’:
      PrintSummaryReport();
      break;
   default:
      DisplayInternalError( “Internal Error 905: Call customer support.” );
}
```

### Итерация

**Перекрестная ссылка** Об использовании итераций см. главу 16.

Итерация — это управляющая структура, которая заставляет группу операторов выполняться несколько раз. Итерацию обычно называют «циклом». К итерациям относятся структуры *For-Next* в Visual Basic и *while* и *for* в C++ и Java. Этот фрагмент кода содержит примеры итераций на Visual Basic:

#### Примеры итераций на Visual Basic

```
' Пример итерации в виде цикла For.
For index = first To last
   DoSomething( index )
Next

' Пример итерации в виде цикла while.
index = first
While ( index <= last )
   DoSomething ( index )
   index = index + 1
Wend

' Пример итерации в виде цикла с выходом.
index = first
Do
   If ( index > last ) Then Exit Do
   DoSomething ( index )
   index = index + 1
Loop
```

Основной тезис структурного программирования гласит, что любая управляющая логика программы может быть реализована с помощью трех конструкций: последовательности, выбора и итерации (Böhm Jacopini, 1966). Программисты иногда предпочитают языковые конструкции, увеличивающие удобство, но программирование, похоже, развивается во многом благодаря ограничению того, что мы можем делать на наших языках программирования. До введения структурного программирования использовать *goto* представлялось очень удобным, но код, написанный таким образом, оказался малопонятным и не поддающимся сопровождению. Я считаю, что использование любых управляющих структур, отличных от этих трех стандартных конструкций, т. е. *break, continue, return, throw-catch* и т. д., должны рассматриваться под критическим углом зрения.

## 19.6. Управляющие структуры и сложность

Одна из причин, по которой столько внимания уделялось управляющим структурам, заключается в том, что они вносят большой вклад в общую сложность программы. Неправильное применение управляющих структур увеличивает сложность, правильное — уменьшает ее.

> Делайте вещи настолько простыми, насколько это возможно, но не проще.
>
> *Альберт Эйнштейн*

Одной из единиц измерения «программной сложности» является число воображаемых объектов, которые вам приходится одновременно держать в уме, чтобы разобраться в программе. Это умственное жонглирование — один из самых сложных аспектов программирования и причина того, что программирование требует большей сосредоточенности, чем другие виды деятельности. По этой причине программисты не любят, когда их «ненадолго прерывают» — такие перерывы равносильны просьбе жонглеру продолжать подкидывать три мяча и подержать вашу сумку с продуктами.

Интуитивно понятно, что сложность программы во многом определяется количеством усилий, требуемых для ее понимания. Том Маккейб (Tom McCabe) опубликовал важную статью, утверждающую, что сложность программы определяется ее управляющей логикой (1976). Другие исследователи обнаружили дополнительные факторы, кроме предложенного Маккейбом цикломатического показателя сложности (например, количество переменных, используемых в программе), но они согласны, что управляющая логика — одна из главных составляющих сложности, если не самая главная.

## Насколько важна сложность?

**Перекрестная ссылка** О сложности см. подраздел «Главный Технический Императив ПО: управление сложностью» раздела 5.2.

Исследователи в области вычислительной техники уже на протяжении двух десятилетий осознают важность проблемы сложности. Много лет назад Дейкстра предупреждал об опасности сложности: «Компетентный программист полностью осознает строго ограниченные размеры своего черепа, поэтому подходит к задачам программирования со всей возможной скромностью» (Dijkstra, 1972). Из этого не следует, что вам нужно увеличить объем вашего черепа, чтобы иметь дело с невероятной сложностью. Это предполагает, что вы можете никогда не связываться с чрезмерной сложностью и всегда должны предпринимать шаги для ее уменьшения.

Сложность управляющей логики имеет большое значение, потому что она коррелирует с низкой надежностью и частыми ошибками (McCabe, 1976, Shen et al., 1985). Вильям Т. Уорд (William T. Ward) сообщал о значительном выигрыше в надежности ПО, полученном в Hewlett-Packard в результате применения показателя сложности Маккейба (1989b). Этот показатель использовался для идентификации проблемных участков в одной программе длиной 77 000 строк. Коэффициент ошибок после выпуска этой программы составил 0,31 ошибку на 1000 строк кода. Коэффициент ошибок в программе длиной 125 000 строк не превышал 0,02 ошибки на 1000 строк кода. Ворд отметил, что из-за своей меньшей сложности обе программы имели значительно меньше дефектов, чем другие программы в Hewlett-Packard. Моя компания Construx Software в 2000 г. получила похожие результаты при использовании средств измерения сложности для поиска проблемных методов.

## Общие принципы уменьшения сложности

Вы можете бороться со сложностью двумя способами. Во-первых, вы можете улучшить свои способности к умственному жонглированию, выполняя специальные упражнения. Но программирование само по себе — уже хорошее упражнение, а люди, похоже, сталкиваются с трудностями при жонглировании бóльшим количеством, чем от пяти до девяти воображаемых сущностей (Miller, 1956). Так что потенциал улучшения невелик. Во-вторых, вы можете уменьшить сложность ваших программ и количество усилий, прилагаемых для их понимания.

### Как измерить сложность

**Дополнительные сведения** Описанный здесь подход основан на важной статье Тома Маккейба «A Complexity Measure» (1976).

У вас, возможно, есть интуитивное ощущение того, что делает метод более или менее сложным. Исследователи пытаются формализовать свои чувства и приводят несколько способов измерения сложности. Возможно, самый важный способ предложил Том Маккейб, Сложность в нем измеряется с помощью подсчета количества «точек принятия решения» в методе (табл. 19-2):

**Табл. 19-2. Способы подсчета точек принятия решения в методе**

| | |
|---|---|
| 1. | Начните считать с 1 на некотором участке кода. |
| 2. | Добавляйте 1 для каждого из следующих ключевых слов или их эквивалентов: *if while repeat for and or*. |
| 3. | Добавляйте 1 для каждого варианта в операторе *case*. |

Приведем пример:

```
if ( ( (status = Success) and done ) or
     ( not done and ( numLines >= maxLines ) ) ) then ...
```

В этом фрагменте вы начинаете считать с 1, получаете 2 для *if*, 3 — для *and*, 4 — для *or* и 5 — для *and*. Таким образом, этот фрагмент содержит всего пять точек принятия решения.

### Что делать с этим измерением сложности

Посчитав количество точек принятия решения, вы можете использовать это число для анализа сложности вашего метода:

| | |
|---|---|
| 0–5 | Этот метод, возможно, в порядке. |
| 6–10 | Начинайте думать о способах упрощения метода. |
| 10+ | Вынесите часть кода в отдельный метод и вызывайте его. |

Перенос части метода в другой метод не упрощает программу — он просто перемещает точки принятия решения. Но он уменьшает сложность, с которой вам приходится иметь дело в каждый момент времени. Поскольку одна из главных целей состоит в минимизации количества элементов, которыми приходится мысленно жонглировать, то уменьшение сложности отдельного метода дает свой результат.

Максимум в 10 точек принятия решения не является абсолютным ограничением. Используйте количество этих точек как сигнал, предупреждающий о том, что метод, возможно, стоит перепроектировать. Не считайте его неколебимым правилом. Оператор *case* со многими вариантами может иметь более 10 элементов, но в зависимости от назначения *case* может быть глупо разбивать его на части.

## Другие виды сложности

Измерение сложности, предложенное Маккейбом, — не единственный значимый показатель, но он наиболее широко обсуждался в компьютерной литературе и особенно полезен при рассмотрении управляющей логики. Другие показатели включают количество используемых данных, число уровней вложенности в управляющих конструкциях, число строк кода, число строк между успешными обращениями к переменной («диапазон»), число строк, в которых используется переменная («время жизни») и объем ввода и вывода. Некоторые исследователи разработали составные показатели сложности, основанные на сочетании перечисленных простых вариантов.

**Дополнительные сведения** Отличное обсуждение показателей сложности см. в «Software Engineering Metrics and Models» (Conte, Dunsmore and Shen, 1986).

### Контрольный список: вопросы по управляющим структурам

http://cc2e.com/1985

- ❑ Используют ли выражения идентификаторы *true* и *false*, а не *1* и *0*?
- ❑ Сравниваются ли логические значения с *true* и *false* неявно?
- ❑ Сравниваются ли числовые значения со своими тестовыми значениями явно?
- ❑ Выполнено ли упрощение выражений с помощью введения новых логических переменных, использования логических функций и таблиц решений?
- ❑ Составлены ли логические выражения позитивно?
- ❑ Сбалансированы ли пары скобок?
- ❑ Используются ли скобки везде, где они необходимы для большей ясности?
- ❑ Заключены ли логические выражения в скобки целиком?
- ❑ Написаны ли условия в соответствии с расположением чисел на числовой прямой?
- ❑ Используются ли в программах на Java выражения вида *a.equals(b)*, а не *a == b* там, где это необходимо?
- ❑ Очевидно ли применение пустых операторов?
- ❑ Выполнено ли упрощение глубоко вложенных выражений с помощью повторной проверки части условия, преобразования в операторы *if-then-else* или *case*, перемещения части кода в отдельные методы, преобразования с использованием более обеъктно-ориентированной модели или они были улучшены как-то иначе?
- ❑ Если метод содержит более 10 точек принятия решения, есть ли хорошая причина, чтобы не перепроектировать его?

## Ключевые моменты

- Упрощение и облегчение чтения логических выражений вносит существенный вклад в качество вашего кода.
- Глубокая вложенность затрудняет понимание метода. К счастью, вы сравнительно легко можете ее избежать.
- Структурное программирование — это простая, но все еще злободневная идея: вы можете построить любую программу с помощью комбинации последовательностей, выборов и итераций.
- Уменьшение сложности — ключ к написанию высококачественного кода.

Часть V

# УСОВЕРШЕНСТВОВАНИЕ КОДА

ГЛАВА 20

# Качество ПО

http://cc2e.com/2036

**Содержание**



**Связанные темы**

- Совместное конструирование: глава 21
- Тестирование, выполняемое разработчиками: глава 22
- Отладка: глава 23
- Предварительные условия конструирования: главы 3 и 4
- Актуальность предварительных условий для современных программных проектов: соответствующий подраздел раздела 3.1

В этой главе мы рассмотрим методики повышения качества ПО в контексте конструирования. Конечно, вся эта книга посвящена повышению качества ПО, но в данной главе вопросы качества и контроля качества обсуждаются более целенаправленно. Темой этой главы являются скорее общие вопросы, а не практические методики контроля качества. Практические советы, касающиеся совместной разработки, а также тестирования и отладки, можно найти в трех следующих главах.

## 20.1. Характеристики качества ПО

Качество ПО имеет внешние и внутренние характеристики. К внешним характеристикам относятся свойства, которые осознает пользователь программы. Они описаны ниже.

- **Корректность** — отсутствие/наличие дефектов в спецификации, проекте и реализации системы.
- **Практичность** — легкость изучения и использования системы.

- **Эффективность** — степень использования системных ресурсов. Эта характеристика учитывает такие факторы, как быстродействие приложения и требуемый им объем памяти.
- **Надежность** — способность системы выполнять необходимые функции в предопределенных условиях; средний интервал между отказами.
- **Целостность** — способность системы предотвращать неавторизованный или некорректный доступ к своим программам и данным. Идея целостности подразумевает ограничение доступа к системе для неавторизованных пользователей, а также обеспечение правильности доступа к данным, т. е. одновременное изменение взаимосвязанных данных, хранение только допустимых значений и т. д.
- **Адаптируемость** — возможность использования системы без ее изменения в тех областях или средах, на которые она не была ориентирована непосредственно.
- **Правильность** — степень безошибочности системы, особенно в отношении вывода количественных данных. Правильность характеризует выполнение системой ее функций, а не то, создана ли она корректно. Этим правильность отличается от корректности.
- **Живучесть** — способность системы продолжать работу при вводе недопустимых данных или в напряженных условиях.

Некоторые из этих характеристик перекрываются, однако каждая имеет свои отличительные черты, которые в одних случаях выражены сильнее, а в других слабее.

Внешние характеристики — единственная категория свойств ПО, которая волнует пользователей. Пользователей беспокоит легкость работы с ПО, а не легкость его изменения. Их заботит корректность ПО, а не удобочитаемость или структированность кода.

Программистов волнуют и внешние характеристики ПО, и внутренние. Раз уж эта книга посвящена программированию, основное внимание в ней уделяется внутренним характеристикам качества, которые перечислены ниже.

- **Удобство сопровождения** — легкость изменения программной системы с целью реализации дополнительных возможностей, повышения быстродействия, исправления дефектов и т. д.
- **Гибкость** — возможный масштаб изменения системы с целью использования ее в тех областях или средах, на которые она не была непосредственно ориентирована.
- **Портируемость** — легкость изменения системы с целью использования в средах, на которые она не была ориентирована непосредственно.
- **Возможность повторного использования** — масштабность и легкость использования частей системы в других системах.
- **Удобочитаемость** — легкость чтения и понимания исходного кода системы, особенно на детальном уровне отдельных операторов.
- **Тестируемость** — возможная степень выполнения блочного и системного тестирования программы и проверки ее соответствия требованиям.

- **Понятность** — легкость понимания системы и на уровне общей организации, и на детальном уровне отдельных операторов. Понятность характеризует согласованность системы на более общем уровне, чем удобочитаемость.

Как и внешние, некоторые из этих внутренних характеристик качества перекрываются, но при этом каждая из них имеет важные отличительные черты.

Внутренние характеристики качества ПО — главная тема этой книги, и в этой главе мы их подробнее рассматривать не будем.

Различие между внутренними и внешними характеристиками качества размыто, потому что на некотором уровне первые влияют на вторые. Если программа недостаточно понятна или неудобна в сопровождении, в ней трудно исправлять дефекты, что в свою очередь влияет на такие внешние характеристики, как корректность и надежность. ПО, страдающее от недостатка гибкости, нельзя улучшить в ответ на запросы пользователей, что сказывается на его практичности. Суть сказанного в том, что одни характеристики качества облегчают жизнь пользователям, а другие — программистам. Вам следует знать, какие из них какие, и понимать, когда и как эти характеристики взаимодействуют.

| Как концентрация на факторе из списка внизу влияет на фактор из списка справа | Корректность | Практичность | Эффективность | Надежность | Целостность | Адаптируемость | Правильность | Живучесть |
|---|---|---|---|---|---|---|---|---|
| Корректность | ↑ | | ↑ | ↑ | | | ↑ | ↓ |
| Практичность | | ↑ | | | | ↑ | ↑ | |
| Эффективность | ↓ | | ↑ | ↓ | ↓ | ↓ | ↓ | |
| Надежность | ↑ | | | ↑ | ↑ | | ↑ | ↓ |
| Целостность | | | ↓ | ↑ | ↑ | | | |
| Адаптируемость | | | | | ↓ | ↑ | | ↑ |
| Правильность | ↑ | | ↓ | ↑ | | ↓ | ↑ | ↓ |
| Живучесть | ↓ | ↑ | ↓ | ↓ | ↓ | ↑ | ↓ | ↑ |

↑ Усиливает
↓ Ослабляет

***Рис. 20-1.*** *Концентрация на одной внешней характеристике качества ПО может влиять на другую характеристику положительно, отрицательно, а может и не влиять*

Улучшение некоторых аспектов качества неизбежно приводит к ухудшению других. Необходимость согласования решения с рядом противоречащих целей делает разработку ПО поистине инженерной дисциплиной. Взаимосвязь внешних характеристик качества пояснена на рис. 20-1. Подобные отношения имеют место и между внутренними характеристиками.

Самый интересный аспект этой диаграммы в том, что концентрация на одной конкретной характеристике не всегда предполагает компромисс с другой характеристикой. Одни характеристики связаны между собой обратным отношением, другие — прямым, а третьи вообще не зависят друг от друга. Так, корректность характеризует степень, в которой работа системы соответствует спецификации.

Живучесть характеризует способность системы продолжать работу даже в непредвиденных условиях. Стремление повысить корректность приводит к снижению живучести и наоборот. В то же время концентрация на адаптируемости повышает живучесть и наоборот.

Диаграмма, показанная на рис. 20.1 иллюстрирует только типичные отношения между характеристиками качества. В конкретном проекте две характеристики могут быть связаны и нетипичным отношением. Поэтому при размышлении о качестве полезно думать о конкретных целевых характеристиках и о том, способствует одна из них достижению другой или препятствует.

## 20.2. Методики повышения качества ПО

Контроль качества ПО — это планомерная и систематичная программа действий, призванная гарантировать, что система обладает желательными характеристиками. Вероятно, самый лучший способ разработки высококачественного приложения — сосредоточиться на самом приложении, однако при контроле качества нужно также концентрироваться на процессе разработки.

***Целевые характеристики качества ПО*** Одной из эффективных методик повышения качества ПО является явное определение целевых внешних и внутренних характеристик. Не имея явных целей, программисты могут сосредоточиться не на тех характеристиках качества, на которых следовало бы.

***Явный контроль качества*** Одна распространенная проблема, связанная с контролем качества, заключается в том, что качество воспринимается как вторичная цель. В некоторых организациях быстрое и «грязное» программирование является скорее правилом, чем исключением. Программисты вроде Глобального Гарри, которые быстро «завершают» работу над программами, наполняя при этом их дефектами, получают большее вознаграждение, чем такие программисты, как Высококачественный Генри, которые пишут прекрасные программы, убеждаясь в их практичности. Не следует удивляться тому, что в подобных организациях программисты не считают качество главным приоритетом. Руководители должны показать программистам, что качество — одна из важнейших целей. Явный контроль качества делает это очевидным, и программисты отвечают соответствующим образом.

***Стратегия тестирования*** Тестирование программы может предоставить детальную оценку ее надежности. Частью контроля качества является разработка стратегии тестирования, учитывающей требования к системе, ее архитектуру и проект. Довольно часто разработчики рассматривают тестирование как главный метод и оценки, и повышения качества. В оставшейся части главы будет показано, что это слишком объемная задача, чтобы ее можно было выполнить исключительно путем тестирования.

**Перекрестная ссылка** О тестировании см. главу 22.

***Принципы разработки ПО*** Технический характер ПО во время его разработки должен контролироваться рядом принципов. Такие принципы используются на всех этапах разработки ПО, включая определение проблемы, выработку требований, конструирование и тестирование системы. Принципы разработки ПО, описываемые в этой книге, в некотором смысле являются принципами конструирования.

**Перекрестная ссылка** Один класс принципов разработки ПО, актуальных для конструирования, обсуждается в разделе 4.2.

***Неформальные технические обзоры*** Многие разработчики сами выполняют обзор своей работы до проведения формального обзора. Неформальный обзор может выражаться в самостоятельной проверке проекта или кода, а также в анализе кода вместе с коллегами.

**Перекрестная ссылка** Об обзорах и инспекциях см. главу 21.

***Формальные технические обзоры*** Важный аспект управления процессом разработки ПО — исправление проблем на этапе «наименьших затрат», т. е. когда на разработку системы потрачен минимальный объем средств и когда решение проблемы окажется наиболее дешевым. Для достижения этой цели служат «ворота качества» — периодические тесты или обзоры, позволяющие определить, достаточным ли качеством обладает система на конкретном этапе разработки, чтобы перейти к следующему этапу. Ворота качества обычно используются при переходе от определения требований к разработке архитектуры, от разработки архитектуры к конструированию, а также от конструирования к тестированию системы. «Воротами» может быть инспекция, обзор системы, выполняемый коллегами или заказчиками, а также аудит.

**Перекрестная ссылка** О зависимости подходов к разработке ПО от типа проекта см. раздел 3.2.

«Ворота качества» не предполагают, что этапы выработки требований или разработки архитектуры нужно выполнить на 100%; они позволяют определить, достаточно ли хороши требования или архитектура, чтобы разработчики могли перейти к следующим этапам. В одних случаях «ворота качества» могут требовать определения только самых важных аспектов архитектуры, а в других — определения почти всех деталей. Разумеется, это зависит от природы конкретного проекта.

***Внешний аудит*** Внешний аудит — это отдельный тип технического обзора, служащий для определения статуса проекта или качества разрабатываемой системы. Аудит выполняют специалисты другой организации, которые сообщают результаты своей работы тому, кто оплачивает аудит, обычно руководителям.

## Процесс разработки

**Дополнительные сведения** Разработка ПО как процесс обсуждается в книге «Professional Software Development» (McConnell, 1994).

Каждый из упомянутых элементов явно связан с контролем качества и неявно — с процессом разработки ПО. Методики разработки, включающие действия, направленные на контроль качества, способствуют созданию более качественного ПО. Другие процессы, не являющиеся сами по себе аспектами контроля качества, также влияют на качество ПО.

**Перекрестная ссылка** О контроле изменений см. раздел 28.2.

***Процедуры контроля изменений*** Повышению качества ПО часто препятствуют неконтролируемые изменения. Неконтролируемые изменения требований могут снижать эффективность проектирования и кодирования. Неконтролируемые изменения проекта могут приводить к несоответствию кода и требований, несогласованности отдельных фрагментов кода и лишней трате времени на вынужденное изменение кода. Неконтролируемые изменения самого кода могут приводить к появлению в нем внутренних противоречий и затрудняют слежение за тем, какой код был под-

вергнут полному обзору и тестированию, а какой — нет. Изменения естественным образом вызывают дестабилизацию системы и снижение ее качества, поэтому эффективный контроль изменений — важнейшее условие достижения высокого качества ПО.

***Оценка результатов*** Только оценив результаты выполнения плана контроля качества, вы сможете узнать, эффективен ли он. Оценка позволяет не только узнать эффективность плана, но и изменить процесс разработки контролируемым образом с целью его улучшения. Кроме того, вы можете извлечь дополнительную выгоду, оценивая сами атрибуты качества: корректность, практичность, эффективность и т. д. Обсуждение оценки атрибутов качества см. в главе 9 книги «Principles of Software Engineering» (Gilb, 1988).

***Прототипирование*** Прототипированием называют разработку реалистичных моделей важных функций системы. Так, разработчики могут использовать прототипирование для определения удобства пользовательского интерфейса, времени выполнения важных вычислений или объема памяти, нужного для хранения типичных наборов данных. Анализ 16 опубликованных и 8 неопубликованных исследований конкретных случаев, посвященный сравнению прототипирования с традиционными методиками разработки, основанными на спецификациях, показал, что прототипирование повышает эффективность проектирования, способствует более полному удовлетворению потребностей пользователей и облегчает сопровождение ПО (Gordon and Bieman, 1991).

## Задание целей

Явное задание целевых аспектов качества — простой и очевидный способ повышения качества ПО, но его легко упустить. Будут ли программисты на самом деле преследовать явные заданные цели? Да, будут, если цели будут им известны и если цели будут разумны. Программисты не могут стремиться к достижению постоянно изменяющихся или недостижимых .

Джеральд Вайнберг и Эдвард Шульман провели один очень интересный эксперимент, посвященный изучению влияния задания целевых аспектов качества на производительность труда программистов (Weinberg and Schulman, 1974). Они попросили пять групп разработчиков написать одну и ту же программу и указали одни и те же пять целевых характеристик качества, но каждой группе было сказано оптимизировать разные характеристики: одной — минимизировать объем используемой программой памяти, второй — обратить максимальное внимание на ясность выводимых данных, третьей — создать как можно более удобочитаемый код, четвертой — сделать программу максимально компактной, а пятой — написать программу за минимальное время. В табл. 20-1 показано, какое место заняла каждая группа по каждому целевому показателю.

**Табл. 20-1. Места, занятые каждой группой по каждому целевому показателю**

| Целевая характеристика качества, на которую группа должна была обратить максимальное внимание | Минимальный объем памяти | Читабельность выводимых данных | Читабельность кода | Компактность кода | Минимальное время разработки |
|---|---|---|---|---|---|
| Минимальный объем памяти | 1 | 4 | 4 | 2 | 5 |
| Читабельность выводимых данных | 5 | 1 | 1 | 5 | 3 |
| Читабельность кода | 3 | 2 | 2 | 3 | 4 |
| Компактность кода | 2 | 5 | 3 | 1 | 3 |
| Минимальное время разработки | 4 | 3 | 5 | 4 | 1 |

**Источник:** «Goals and Performance in Computer Programming» (Weinberg and Schulman, 1974).

Результаты впечатляют: четыре из пяти групп выполнили поставленные перед ними задачи лучше всех остальных групп. Оставшаяся группа заняла в своей категории второе место. Никакой группе не удалось преуспеть в достижении всех целей.

Что из этого следует? То, что люди на самом деле делают то, о чем вы их просите. Программисты действительно стремятся к достижению поставленных целей, но они должны знать, что это за цели. И еще одно следствие: как и ожидалось, целевые характеристики качества конфликтуют, поэтому преуспеть в достижении всех целей почти невозможно.

## 20.3. Относительная эффективность методик контроля качества ПО

Не все методики контроля качества имеют одинаковую эффективность. Эффективность многих методик в плане нахождения и устранения дефектов уже известна. В данном разделе мы обсудим этот и некоторые другие аспекты «эффективности» методик контроля качества.

### Эффективность обнаружения дефектов

> Если бы строители возводили здания так, как программисты пишут программы, первый же дятел уничтожил бы цивилизацию.
>
> *Джеральд Вайнберг* (*Gerald Weinberg*)

Некоторые методики более эффективны в обнаружении дефектов, чем другие, к тому же разные методики приводят к обнаружению разных дефектов. Одним из способов оценки методик нахождения дефектов является определение процента обнаруженных дефектов из общего числа дефектов, имеющихся в программе на конкретном этапе. Показатели эффективности нахождения дефектов при использовании некоторых популярных методик указаны в табл. 20-2.

**Табл. 20-2. Эффективность нахождения дефектов при использовании разных методик**

| Методика устранения дефектов | Минимальная эффективность | Типичная эффективность | Максимальная эффективность |
|---|---|---|---|
| Неформальные обзоры проекта | 25% | 35% | 40% |
| Формальные инспекции проекта | 45% | 55% | 65% |
| Неформальные обзоры кода | 20% | 25% | 35% |
| Формальные инспекции кода | 45% | 60% | 70% |
| Моделирование или прототипирование | 35% | 65% | 80% |
| Самостоятельная проверка кода | 20% | 40% | 60% |
| Блочное тестирование | 15% | 30% | 50% |
| Тестирование новых функций (компонентов) | 20% | 30% | 35% |
| Интеграционное тестирование | 25% | 35% | 40% |
| Регрессивное тестирование | 15% | 25% | 30% |
| Тестирование системы | 25% | 40% | 55% |
| Ограниченное бета-тестирование (менее чем в 10 организациях) | 25% | 35% | 40% |
| Крупномасштабное бета-тестирование (более чем в 1000 организаций) | 60% | 75% | 85% |

**Источники:** «Programming Productivity» (Jones, 1986a), «Software Defect-Removal Efficiency» (Jones, 1996) и «What We Have Learned About Fighting Defects» (Shull et al., 2002).

Самое интересное в этих данных то, что типичная эффективность обнаружения дефектов при использовании любой методики не превышает 75% и что в среднем она равна примерно 40%. Более того, самые популярные методики — блочное тестирование и интеграционное тестирование — позволяют найти обычно только около 30–35% дефектов. Как правило, используется подход, основанный на интенсивном тестировании, что позволяет устранить лишь около 85% дефектов. Ведущие организации используют более широкий диапазон методик, достигая при этом 95%-ой или более высокой эффективности устранения дефектов (Jones, 2000).

Итак, если разработчики хотят достигнуть более высокой эффективности обнаружения дефектов, они должны полагаться на комбинацию методик. Одно из подтверждений этого вывода было получено в классическом исследовании Гленфорда Майерса (Myers, 1978b). Участниками исследования были программисты, обладавшие минимум 7-, а в среднем — 11-летним опытом. Исследуемая программа

содержала 15 известных ошибок. Майерс попросил каждого программиста найти эти ошибки, используя одну из следующих методик:

- тестирование выполнения программы по спецификации;
- тестирование выполнения программы по спецификации с возможностью изучения исходного кода;
- анализ/инспекция с использованием и спецификации, и исходного кода.

Различия эффективности обнаружения дефектов оказались очень большими: программисты нашли от 1 до 9 дефектов. Средний показатель был равен 5,1, или 1/3 от общего числа известных дефектов.

При использовании одной методики никакая из них не имела статистически значимого преимущества над любой другой. В то же время любая комбинация двух методик — в том числе использование одной методики двумя независимыми группами — приводила к увеличению общего числа найденных дефектов почти вдвое. В исследованиях, проведенных в Лаборатории проектирования ПО NASA, компании Boeing и других компаниях, было обнаружено, что разные программисты находят разные дефекты. Только примерно каждую пятую ошибку, обнаруженную в ходе инспекций, находят двое или более разработчиков (Kouchakdjian, Green, and Basili, 1989; Tripp, Struck, and Pflug, 1991; Schneider, Martin, and Tsai, 1992).

Майерс обращает внимание на то, что поиск одних видов ошибок оказывается более эффективным при непосредственном участии людей (например, при инспекции или анализе кода), а других видов — при компьютерном тестировании (Myers, 1979). Этот вывод подтвердился в более позднем исследовании, показавшем, что чтение кода способствует нахождению дефектов интерфейса, а функциональное тестирование — нахождению дефектов управляющих структур (Basili, Selby, and Hutchens, 1986). Гуру тестирования Борис Бейзер (Boris Beizer) сообщает, что неформальные подходы к тестированию обычно позволяют достигнуть покрытия кода тестами лишь на 50–60%, если только вы не используете анализатор покрытия (Johnson, 1994).

Таким образом, методики поиска дефектов лучше применять в комбинации. Джонс (Jones) также подтверждает этот вывод. Используя исключительно тестирование, высоких результатов добиться невозможно. Джонс сообщает, что комбинация блочного тестирования, функционального тестирования и тестирования системы часто приводит к обнаружению менее 60% дефектов, что обычно неприемлемо для конечного продукта.

Эти данные также помогают понять, почему программисты, начинающие применять дисциплинированную методику устранения дефектов, такую как экстремальное программирование, добиваются более высокой степени устранения дефектов. Как показывает табл. 20-3, набор методик устранения дефектов, применяемых в экстремальном программировании, позволяет устранить около 90% дефектов в обычной ситуации и 97% в лучшем случае, что гораздо выше среднего для отрасли показателя, равного 85%. Некоторые программисты связывают этот факт с синергичными отношениями между методиками экстремального программирования, но на самом деле это просто предсказуемый результат использования конкретного набора методик устранения дефектов. Эффективность других комбинаций методик может оказаться такой же или даже более высокой, поэтому выбор кон-

кретных методик устранения дефектов, позволяющих достичь желаемого уровня качества, является одним из аспектов эффективного планирования проекта.

**Табл. 20-3. Эффективность обнаружения дефектов, характерная для экстремального программирования**

| Методика устранения дефектов | Минимальная эффективность | Типичная эффективность | Максимальная эффективность |
|---|---|---|---|
| Неформальные обзоры проекта (парное программирование) | 25% | 35% | 40% |
| Неформальные обзоры кода (парное программирование) | 20% | 25% | 35% |
| Самостоятельная проверка кода | 20% | 40% | 60% |
| Блочное тестирование | 15% | 30% | 50% |
| Интеграционное тестирование | 25% | 35% | 40% |
| Регрессивное тестирование | 15% | 25% | 30% |
| Общая эффективность устранения дефектов | ~74% | ~90% | ~97% |

## Стоимость нахождения дефектов

Некоторые методики обнаружения дефектов дороже других. Наиболее экономичные при прочих равных условиях имеют наименьшую стоимость в расчете на один обнаруженный дефект. Равенством прочих условий пренебрегать нельзя, поскольку стоимость методики в расчете на один дефект зависит от общего числа обнаруженных дефектов, этапа обнаружения каждого дефекта и других факторов, не связанных с экономическими аспектами конкретной методики.

Как правило, эксперименты показывают, что инспекции обходятся дешевле, чем тестирование. В исследовании, проведенном в Лаборатории проектирования ПО, было обнаружено, что при чтении кода число дефектов, находимых в час, было примерно на 80% более высоким, чем при тестировании (Basili and Selby, 1987). В другой организации поиск дефектов проектирования с использованием блочного тестирования был вшестеро дороже, чем при использовании инспекций (Ackerman, Buchwald, and Lewski, 1989). Более позднее исследование, проведенное в IBM, показало, что на обнаружение каждой ошибки разработчики тратили 3,5 человеко-часа в случае инспекций кода и 15–25 в случае тестирования (Kaplan, 1995).

## Стоимость исправления дефектов

Стоимость нахождения дефектов — только одна часть уравнения. Другой частью является стоимость их исправления. На первый взгляд, методика обнаружения дефектов не играет роли: стоимость их исправления всегда будет одинаковой.

Это неверно, потому что чем дольше дефект остается в системе, тем больше средств придется потратить на его устранение. Следовательно, методика, способствующая раннему обнаружению ошибок, снижает стоимость их исправления. Еще важнее

**Перекрестная ссылка** О зависимости стоимости исправления дефектов от срока их присутствия в системе см. раздел «Обращение к данным» раздела 3.1. Сами ошибки более подробно обсуждаются в разделе 22.4.

то, что одни методики — такие как инспекции — позволяют определить и симптомы, и причины дефектов за один этап; другие — например, тестирование — указывают на симптомы дефекта, но требуют выполнения дополнительной работы для диагностики и устранения его причины. В итоге одноэтапные методики оказываются гораздо более дешевыми, чем двухэтапные.

В одном из подразделений Microsoft обнаружили, что при использовании инспекции кода — одноэтапной методики — на нахождение и исправление дефекта уходит 3 часа, тогда как при использовании тестирования — двухэтапной методики — на это требуется 12 часов (Moore, 1992). Коллофелло и Вудфилд сообщили, что при разработке программы из 700 000 строк, над которой работало более 400 программистов, обзоры кода имели гораздо более высокую экономическую эффективность, чем тестирование: прибыль на инвестированный капитал была равной 1,38 и 0,17 соответственно (Collofello and Woodfield, 1989).

Суть сказанного в том, что эффективная программа контроля качества должна включать комбинацию методик, применяемых на всех стадиях разработки. Для достижения высокого качества ПО можно использовать следующую комбинацию:

- формальные инспекции всех требований, всех аспектов архитектуры и всех проектов критических частей системы;
- моделирование или прототипирование;
- чтение или инспекции кода;
- тестирование выполнения программы.

## 20.4. Когда выполнять контроль качества ПО?

**Перекрестная ссылка** Контроль качества предварительных действий — например, определения требований и разработки архитектуры — в этой книге не рассматривается. Информацию по этим темам можно найти в книгах, указанных в разделе «Дополнительные ресурсы» в конце этой главы.

Как было отмечено в главе 3, чем раньше ошибка внедряется в приложение, тем сильнее она переплетается с другими частями приложения и тем больше средств придется потратить на ее устранение. Дефект в требованиях может вылиться в один или несколько дефектов в проекте, которые могут привести к появлению множества дефектов в коде. Ошибка в требованиях может привести к разработке дополнительных компонентов архитектуры или подтолкнуть к неудачным архитектурным решениям. Дополнительные архитектурные компоненты требуют написания дополнительного кода, тестов и документации. С другой стороны, ошибка в требованиях может привести к выбрасыванию частей архитектуры, кода и тестов. Если идея устранения ошибок из чертежей дома перед заливкой фундамента бетоном кажется вам разумной, то вы согласитесь и с тем, что дефекты требований и архитектуры также следует устранять до того, как они повлияют на более поздние этапы разработки.

Кроме того, ошибки в требованиях или архитектуре обычно имеют более широкие следствия, чем ошибки конструирования. Одна ошибка в архитектуре может затронуть несколько классов и десятки методов, тогда как одна ошибка конструи-

ирования скорее всего повлияет только на один метод или класс. Это еще одно убедительное обоснование как можно более раннего нахождения ошибок.

Дефекты проникают в ПО на всех стадиях разработки, поэтому контролю качества следует уделять должное внимание на всех этапах проекта, начиная с самых ранних. Контроль качества нужно внести в планы в начале работы над программой; его следует выполнять по мере прогресса; наконец, он должен подчеркивать удачное завершение работы над проектом.

## 20.5. Главный Закон Контроля Качества ПО

Ни в одном ресторане посетителей не кормят бесплатно, и даже если б кормили, никто не смог бы поручиться за качество блюд. Однако разработка ПО — совсем не кулинарное искусство, и качество ПО имеет одну важную необычную особенность. Главный Закон Контроля Качества ПО заключается в том, что повышение качества системы снижает расходы на ее разработку.

В основе этого закона лежит одно важное наблюдение: лучшим способом повышения производительности труда программистов и качества ПО является минимизация времени, затрачиваемого на исправление кода, чем бы оно ни объяснялось: изменениями требований, изменениями проекта или отладкой. Средняя для отрасли производительность труда программистов эквивалентна примерно 10–50 строкам кода на одного человека в день (с учетом всех затрат, не связанных с кодированием). Написание 10–50 строк кода требует нескольких минут, — на что же уходит остальное время?

Такая, казалось бы, низкая производительность труда частично объясняется тем, что в подобных средних показателях учитывается время, не связанное непосредственно с программированием. Время тестировщиков, руководителей, секретарей — все эти факторы включены в данный показатель. Определение требований, разработка архитектуры и другие действия, не относящиеся к кодированию, также отражены в «строках кода в день». Однако основные временные затраты объясняются не этим.

**Перекрестная ссылка** О различиях между написанием отдельной программы и созданием программного продукта см. подраздел «Программы, продукты, системы и системные продукты» раздела 27.5.

Самый длительный этап в большинстве проектов — отладка и исправление неправильного кода. При традиционном цикле разработки ПО эти действия занимают около 50% времени (см. раздел 3.1). Сокращение потребности в отладке, достигаемое благодаря предотвращению ошибок, повышает производительность труда. Следовательно, наиболее очевидный метод сокращения графика разработки — повышение качества ПО и снижение объема времени, уходящего на его отладку и исправление.

Этот анализ подтверждается реальными данными. В обзоре 50 проектов, потребовавших более 400 человеколет и включивших почти 3 000 000 строк кода, проведенном в Лаборатории проектирования ПО NASA, было обнаружено, что повышенное внимание к контролю качества позволяло снизить уровень ошибок, но не повышало общие расходы на разработку (Card, 1987).

В исследовании, проведенном в IBM, были получены аналогичные результаты:

Программным проектам с наименьшими уровнями дефектов соответствовали самые короткие графики разработки и максимальные показатели производительности труда... устранение дефектов на самом деле — самый дорогой и длительный этап разработки ПО (Jones, 2000).

Это верно и для противоположного края шкалы. В одном исследовании 1985 года ученые попросили 166 профессиональных программистов написать программы по одной и той же спецификации. Итоговые программы содержали в среднем 220 строк, а на их написание ушло в среднем чуть меньше 5 часов. Результаты оказались поистине удивительными: программисты, работавшие над своими программами средний объем времени, допустили наибольшее число ошибок. Программисты, которым потребовалось больше или меньше времени, допустили значительно меньше ошибок (DeMarco and Lister, 1985). Результаты показаны на рисунке 20-2.

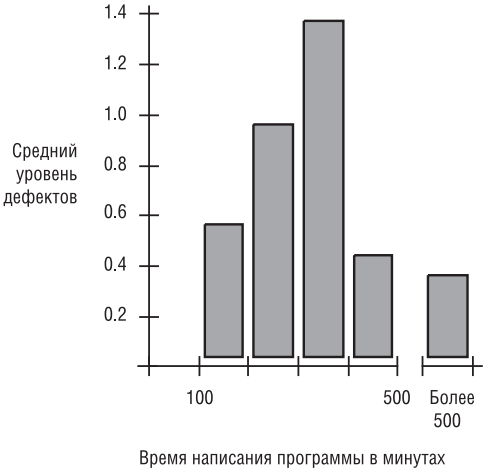



***Рис. 20-2.*** *Ни при самом быстром, ни при самом медленном подходе к разработке ПО не наблюдается наибольший уровень дефектов*

В сравнении с самой быстрой группой двум самым медленным группам понадобилось примерно в 5 раз больше времени для достижения результата с примерно тем же уровнем дефектов. Таким образом, на создание ПО без дефектов не всегда уходит больше времени, чем на написание ПО с дефектами.

Вероятно, в некоторых случаях контроль качества требует значительных затрат. Если вы пишете приложение управления космическим кораблем или медицинской системой жизнеобеспечения, высокие требования к надежности ПО делают проект более дорогим.

В сравнении с традиционным циклом «кодирование — тестирование — отладка» улучшенная программа контроля качества ПО оказывается более экономичной. Она перенаправляет ресурсы от отладки и рефакторинга к предварительным этапам контроля качества. Предварительные этапы влияют на качество системы больше, чем последующие, поэтому время, потраченное на предварительных этапах, позволяет сэкономить больше времени потом. Результатом является снижение уровня

дефектов, сокращение сроков разработки и снижение затрат. В трех следующих главах вы найдете еще несколько примеров, иллюстрирующих Главный Закон Контроля Качества ПО.

**Контрольный список: план контроля качества**

http://cc2e.com/2043

- ❑ Идентифицировали ли вы специфические характеристики качества, имеющие особую важность в вашем проекте?
- ❑ Сообщили ли вы другим программистам целевые характеристики качества?
- ❑ Провели ли вы различие между внешними и внутренними характеристиками качества?
- ❑ Обдумали ли вы, как некоторые характеристики могут усиливать или ослаблять другие?
- ❑ Призывает ли ваш проект к использованию нескольких методик обнаружения ошибок, ориентированных на поиск разных видов ошибок?
- ❑ Составили ли вы план контроля качества, охватывающий все этапы разработки ПО?
- ❑ Оцениваете ли вы качество системы каким-нибудь образом, чтобы можно было определить, повышается оно или понижается?
- ❑ Понимают ли руководители, что контроль качества требует дополнительных расходов в начале проекта, но зато позволяет добиться общей экономии средств?

# Дополнительные ресурсы

http://cc2e.com/2050

Составить список книг для этой главы несложно, потому что методики повышения качества ПО и производительности труда описываются почти во всех трудах, посвященных эффективным методологиям разработки ПО. Сложность в том, чтобы выделить книги, касающиеся непосредственно качества ПО. Ниже я указал две такие работы.

Ginac, Frank P. *Customer Oriented Software Quality Assurance*. Englewood Cliffs, NJ: Prentice Hall, 1998. В этой очень краткой книге описаны атрибуты качества, метрики качества, программы контроля качества, роль тестирования в контроле качества, а также известные программы повышения качества, в том числе модель CMM, разработанная в институте Software Engineering Institute, и стандарты ISO серии 9000.

Lewis, William E. *Software Testing and Continuous Quality Improvement*, 2d ed. Auerbach Publishing, 2000. В этой книге можно найти подробное обсуждение цикла контроля качества, а также методик тестирования. Кроме того, в ней вы найдете много контрольных форм и списков.

## Соответствующие стандарты

http://cc2e.com/2057

*IEEE Std 730-2002* — стандарт IEEE планирования контроля качества ПО.

*IEEE Std 1061-1998* — стандарт IEEE методологии метрик качества ПО.

*IEEE Std 1028-1997* — стандарт обзоров ПО.

*IEEE Std 1008-1987 (R1993)* — стандарт блочного тестирования ПО.

*IEEE Std 829-1998* — стандарт документирования тестов ПО.

## Ключевые моменты

- Высокого качества можно достичь без дополнительных затрат, но для этого вы должны перераспределить ресурсы и предотвращать дефекты вместо того, чтобы их исправлять.
- Стремление к одним характеристикам качества препятствует достижению других. Четко определите цели, имеющие для вас первостепенную важность, и сообщите об этом всем членам группы.
- Никакая методика обнаружения дефектов не является достаточно эффективной. Тестирование само по себе — не самый лучший способ устранения ошибок. Составляя программу контроля качества, предусмотрите применение нескольких методик, позволяющих обнаружить разные виды ошибок.
- Существуют многие эффективные методики контроля качества, применяемые как во время конструирования, так и до его начала. Чем раньше вы обнаружите дефект, тем слабее он переплетется с остальным кодом и тем меньше вреда он успеет принести.
- В мире программирования контроль качества ориентирован на процесс. В отличие от промышленного производства разработка ПО не включает повторяющегося этапа, влияющего на конечный продукт, поэтому качество результата определяется процессом, используемым для разработки ПО.

ГЛАВА 21

# Совместное конструирование

**Содержание**

http://cc2e.com/2185



**Связанные темы**

- Качество ПО: глава 20
- Тестирование, выполняемое разработчиками: глава 22
- Отладка: глава 23
- Предварительные условия конструирования: главы 3 и 4

Вероятно, вам знакома одна довольно распространенная ситуация. Вы подходите к столу другого программиста и говорите: «Не мог бы ты взглянуть на этот код? Он не работает». Вы начинаете объяснять: «Причиной не может быть вот это, потому что я сделал то-то и то-то. Причиной также не может быть это, потому что я сделал вот это. Кроме того, причиной не может быть... подожди... Это *может* быть причиной. Спасибо!» Вы решили проблему, хотя ваш «помощник» не произнес ни слова.

Так или иначе все методики совместного конструирования направлены на формализацию процесса проверки вашей работы другими программистами с целью устранения ошибок.

Если вы уже читали об инспекциях и парном программировании, вы найдете в этой главе мало нового. Возможно, вас заинтересуют данные об эффективности инспекций (раздел 21.3), а также методика чтения кода (раздел 21.4). Вы также можете взглянуть на табл. 21-1 «Сравнение методик совместного конструирования» в конце главы. Если ваши знания основаны только на опыте, читайте дальше! Каждый человек имеет собственный опыт, поэтому некоторые идеи окажутся для вас новыми.

# 21.1. Обзор методик совместной разработки ПО

«Совместным конструированием» можно называть парное программирование, формальные инспекции, неформальные технические обзоры, чтение документации, а также другие методики, подразумевающие разделение ответственности за те или иные результаты работы между несколькими программистами. В моей компании термин «совместное конструирование» ввел в обиход Мэтт Пелокуин (Matt Peloquin) где-то в 2000 году. По-видимому, примерно тогда же независимо от нас этот термин стали использовать и в других компаниях.

Все методики совместного конструирования основаны на идее, что разработчики плохо находят определенные дефекты в своей работе и что каждый человек имеет свои недостатки, поэтому качество работы повысится, если ее проверит кто-то другой. Исследования, проведенные в Институте разработки ПО (Software Engineering Institute), показали, что разработчики допускают в среднем от 1 до 3 дефектов в час при проектировании и от 5 до 8 дефектов в час при кодировании (Humphrey, 1997). Ясно, что устранение этих дефектов — обязательное условие эффективного конструирования.

## Совместное конструирование дополняет другие методики контроля качества

Главной целью совместного конструирования является повышение качества ПО. Как уже отмечалось в главе 20, само по себе тестирование ПО имеет довольно невысокую эффективность: средний уровень определения дефектов равен примерно 30% при блочном тестировании, 35% при интеграционном тестировании и 35% при ограниченном бета-тестировании. В то же время средняя эффективность инспекций проектов и кода равна соответственно 55% и 60% (Jones, 1996). Дополнительное преимущество совместного конструирования состоит в том, что оно сокращает время разработки, что в свою очередь снижает расходы.

Предварительные отчеты о результатах парного программирования говорят о том, что оно позволяет достигнуть примерно такого же качества кода, что и формальные инспекции (Shull et al., 2002). Затраты на разработку при применении только парного программирования оказываются примерно на 10–25% выше, чем при программировании в одиночку, но зато сокращение сроков разработки составляет около 45%, что может оказаться решающим преимуществом над разработкой в одиночку (Boehm and Turner, 2004), хотя не над инспекциями, которые приводят к похожим результатам.

Технические обзоры изучаются гораздо дольше, чем парное программирование, и результаты проведенных исследований впечатляют.

■ В IBM обнаружили, что каждый час инспекции предотвращал около 100 часов аналогичной работы (тестирования и исправления дефектов) (Holland, 1999).

- Принятие инициативы, основанной на инспекциях, позволило компании Raytheon снизить объем затрат на исправление дефектов с 40% общей стоимости проектов примерно до 20% (Haley, 1996).
- Специалисты Hewlett-Packard сообщили, что благодаря программе инспекций они добились экономии примерно 21,5 млн долларов в год (Grady and Van Slack, 1994).
- В компании Imperial Chemical Industries обнаружили, что затраты на сопровождение пакета, состоящего примерно из 400 программ, составляли лишь около 10% от затрат на сопровождение аналогичного пакета программ, которые не были подвергнуты инспекции (Gilb and Graham, 1993).
- Исследование крупных программ показало, что каждый час инспекций предотвращал в среднем 33 часа сопровождения программ и что инспекции иногда были аж в 20 раз эффективнее тестирования (Russell, 1991).
- В организации, занимающейся сопровождением ПО, до введения обзоров кода изменения одной строки оказывались ошибочными в 55% случаев. После введения обзоров этот показатель снизился до 2% (Freedman and Weinberg, 1990). В целом после введения обзоров программисты стали правильно выполнять с первого раза 95% изменений. До введения обзоров с первого раза правильно выполнялись менее 20% изменений.
- Одна группа программистов разработала 11 программ. Первые пять, разработанные без выполнения обзоров, содержали в среднем 4,5 ошибки на 100 строк кода. Другие шесть программ, подвергавшиеся инспекциям, содержали в среднем 0,82 ошибки на 100 строк кода. Иначе говоря, проведение обзоров снижало уровень ошибок более чем на 80% (Freedman and Weinberg, 1990).
- Кейперс Джонс сообщает, что во всех изученных им программных проектах с 99%-ым или более высоким уровнем устранения дефектов использовались формальные инспекции. С другой стороны, ни в одном проекте с 75%-ой или более низкой эффективностью устранения дефектов формальные инспекции не проводились (Jones, 2000).

Многие из этих исследований подтверждают Главный Закон Контроля Качества ПО, согласно которому уменьшение числа дефектов в программе приводит к сокращению времени ее разработки.

Самые разные исследования показали, что методики совместной разработки не только обеспечивают более высокую эффективность нахождения ошибок, чем тестирование, но и позволяют находить типы ошибок, на которые тестирование указать не может (Myers 1978; Basili, Selby, and Hutchens, 1986). Как сказал Карл Вигерс, «человек, выполняющий обзор, может обратить внимание на неясные сообщения об ошибках, неадекватные комментарии, жестко закодированные значения переменных и повторяющиеся фрагменты кода, которые следует объединить. При тестировании все это невозможно» (Wiegers, 2002). Кроме того, программисты, знающие, что их работа будет подвергнута обзору, выполняют ее более добросовестно. Таким образом, даже при высокой эффективности тестирования в программу контроля качества следует включить обзоры или другие методики совместной разработки.

## Совместное конструирование способствует усвоению корпоративной культуры и обмену опытом программирования

Неформальные процедуры обзоров передавались от человека человеку в общей культуре программирования задолго до того, как информация об этом стала появляться в печатных материалах. Необходимость обзоров была настолько очевидна лучшим программистам, что они редко упоминали об этом в статьях и книгах, тогда как худшие программисты считали, что они настолько хороши, что их работа не нуждается в обзорах.

*Дениел Фридмен и Джеральд Вайнберг (Daniel Freedman and Gerald Weinberg)*

Стандарты программирования можно выразить на бумаге и распространить среди сотрудников, но если их не обсуждать и не поощрять их применение, никто не будет их соблюдать. Обзоры — важный механизм предоставления программистам обратной связи, касающейся их кода. Код, стандарты и причины стандартизации кода — все эти темы достойны обсуждения во время обзоров.

Программисты должны получать информацию не только о том, насколько хорошо они следуют стандартам, но и о более субъективных аспектах программирования, таких как форматирование, использование комментариев, имен переменных, локальных и глобальных переменных, методик проектирования и т. д. Начинающие программисты нуждаются в советах более опытных коллег, а более опытные и, как правило, более занятые — в мотивации, которая подтолкнула бы их к передаче опыта. Обзоры — тот механизм, который создает начинающим и опытным программистам все условия для обсуждения технических вопросов. Таким образом, обзоры способствуют повышению качества не только текущего кода, но и будущих программ.

В одной группе было обнаружено, что использование формальных инспекций быстро повысило компетентность всех разработчиков до уровня самых лучших (Tackett and Van Doren, 1999).

## Все формы совместного конструирования предполагают совместное владение результатами работы

**Перекрестная ссылка** Все методики совместного конструирования объединяет идея совместного владения. В некоторых моделях разработки код принадлежит его автору, а возможность изменения кода других программистов ограничивается писаными или неписаными правилами. Совместное владение предъявляет более высокие требования к координации труда, особенно к управлению конфигурацией ПО (см. раздел 28.2).

При совместном владении весь код принадлежит группе, а не отдельным программистам, поэтому изучать и изменять его могут разные члены группы. Это обеспечивает несколько важных преимуществ:

- увеличение числа программистов, разрабатывающих и анализирующих конкретный код, способствует повышению его качества;
- уход одного из участников проекта не приводит к серьезным последствиям, потому что каждый фрагмент кода известен нескольким программистам;
- возможность поручить исправление ошибок любому из нескольких программистов позволяет ускорить исправление дефектов.

В некоторых методологиях, таких как экстремальное программирование, рекомендуется формально объединять программистов в пары и чередовать задания, назначенные конкретным парам. В моей компании обнаружили, что и без фор-

мальной организации пар программисты могут по ходу дела хорошо ознакомиться с кодом своих коллег. Для этого мы комбинируем формальные и неформальные технические обзоры, используем в случае надобности парное программирование и поручаем исправление дефектов поочередно разным программистам.

### Сотрудничество возможно не только во время конструирования, но и до и после него

Эта книга посвящена конструированию, поэтому основное внимание в этой главе уделяется сотрудничеству при детальном проектировании и кодировании. Однако большинство принципов совместного конструирования, описываемых в этой главе, можно использовать и на этапах оценки, планирования, определения требований, разработки архитектуры, тестирования и сопровождения программы. Изучив ресурсы, указанные в конце этой главы, вы сможете применять методики совместной разработки на большинстве этапов создания ПО.

## 21.2. Парное программирование

При парном программировании один программист печатает код на клавиатуре, а второй следит за тем, чтобы в программу не вкрались ошибки, и думает о правильности кода в стратегическом масштабе. Первоначально парное программирование приобрело популярность благодаря экстремальному программированию (Beck, 2000), но теперь парное программирование используется более широко (Williams and Kessler, 2002).

### Условия успешности парного программирования

Базовая идея парного программирования проста, но из него можно извлечь еще большую выгоду, следуя нескольким советам.

***Поддерживайте парное программирование стандартами кодирования*** Парное программирование не будет эффективным, если члены пары будут тратить время на споры о стиле кодирования. Попытайтесь стандартизовать то, что в главе 5 было названо «несущественными атрибутами» программирования, чтобы программисты могли сосредоточиться на стоящей перед ними «существенной» задаче.

***Не позволяйте парному программированию превратиться в наблюдение*** Член пары, не занимающийся непосредственно написанием кода, должен быть активным участником программирования. Он должен анализировать код, думать о том, что реализовать в следующую очередь, оценивать проект программы и планировать тестирование кода.

***Не используйте парное программирование для реализации простых фрагментов*** Члены одной группы, использовавшие парное программирование для написания наиболее сложного кода, обнаружили, что выгоднее посвятить 15 минут детальному проектированию на доске и затем программировать поодиночке (Manzo, 2002). В большинстве организаций, пробующих парное программирование, в итоге приходят к выводу, что в парах лучше выполнять не все, а только некоторые части работы (Boehm and Turner, 2004).

***Регулярно меняйте состав пар и назначаемые парам задачи*** Как и при других методиках совместной разработки, при парном программировании выгода объясняется тем, что каждый из программистов изучает разные части системы. Регулярно меняйте состав пар для стимуляции «перекрестного опыления» — некоторые эксперты рекомендуют выполнять это каждый день (Reifer, 2002).

***Объединяйте в пару людей, предпочитающих одинаковый темп работы*** Если один партнер работает слишком быстро, парное программирование начинает терять смысл. Более быстрый член пары должен снизить темп, или пару следует разбить и сформировать в другом составе.

***Убедитесь, что оба члена пары видят экран*** Эффективность парного программирования могут снижать даже такие, казалось бы, банальные вопросы, как неправильное расположение монитора и слишком мелкий шрифт.

***Не объединяйте в пару людей, которые не нравятся друг другу*** Эффективность работы в паре зависит от соответствия характеров двух программистов. Бессмысленно объединять в пару людей, которые плохо ладят друг с другом (Beck, 2000; Reifer, 2002).

***Не составляйте пару из людей, которые ранее не программировали в паре*** Парное программирование приносит максимальную выгоду, если хотя бы один из партнеров имеет опыт работы в паре (Larman, 2004).

***Назначьте лидера группы*** Если все члены вашей группы хотят выполнить все программирование в парах, возложите на кого-то ответственность за распределение задач, контроль результатов и связь с людьми, не участвующими в проекте.

## Достоинства парного программирования

Парное программирование имеет целый ряд достоинств.

- В сравнении с одиночным программированием оно позволяет программистам успешнее противостоять стрессу. Члены пар поощряют друг друга поддерживать высокое качество кода даже в напряженных условиях, подталкивающих к быстрому написанию «грязного» кода.
- Оно повышает качество кода. Удобочитаемость и понятность кода всех программистов повышаются до уровня кода лучшего программиста группы.
- Оно ускоряет разработку системы. Как правило, пары пишут код быстрее, допуская при этом меньше ошибок. Соответственно в конце проекта группе приходится тратить меньше времени на исправление дефектов.
- Оно обеспечивает все остальные общие преимущества совместного конструирования, такие как распространение корпоративной культуры, обучение неопытных программистов и содействие совместному владению результатами работы.

**Контрольный список: эффективное парное программирование**

http://cc2e.com/2192

- ❑ Утвердили ли вы стандарт кодирования, позволяющий парам сосредоточиться на программировании и не тратить время на философские диспуты о стиле кодирования?
- ❑ Оба ли члена пары принимают активное участие в программировании?
- ❑ Не используете ли вы парное программирование для реализации всех аспектов системы? Определяете ли вы задачи, которые действительно целесообразно решать в парах?
- ❑ Не забываете ли вы регулярно менять состав пар и назначаемые им задачи?
- ❑ Соответствуют ли члены пар друг другу по темпу работы и характеру?
- ❑ Назначили ли вы лидера группы, координирующего действия участников проекта и отвечающего за связь с людьми, не участвующими в проекте?

## 21.3. Формальные инспекции

**Дополнительные сведения** Первой работой, посвященной инспекциям, является статья «Design and Code Inspections to Reduce Errors in Program Development» (Fagan, 1976).

Инспекцией называют специфический вид обзора, обеспечивающий очень высокую эффективность обнаружения дефектов и требующий меньших затрат, чем тестирование. Инспекции были разработаны Майклом Фаганом (Michael Fagan) и уже использовались в IBM за несколько лет до того, как Фаган опубликовал свою работу, которая сделала их доступными широким массам. Хотя любой обзор предполагает изучение проектов или кода, инспекция отличается от обыкновенного обзора несколькими важными аспектами:

- используемые при инспекциях контрольные списки концентрируют внимание инспекторов на областях, с которыми ранее были связаны проблемы;
- главной целью инспекции является обнаружение, а не исправление дефектов;
- люди, выполняющие инспекцию, готовятся к инспекционному собранию заблаговременно и прибывают на него со списком обнаруженных ими проблем;
- всем участникам инспекции назначаются конкретные роли;
- координатор инспекции не является автором продукта, подвергающегося инспекции;
- координатор прошел специальное обучение координированию инспекций;
- инспекционное собрание проводится, только если все участники адекватно к нему подготовились;
- данные, полученные при каждой инспекции, используются для улучшения будущих инспекций;
- руководители не посещают инспекционных собраний, если только вы не инспектируете план проекта или другие организационные материалы; участие технических лидеров в инспекционных собраниях допускается.

## Каких результатов можно ожидать от инспекций?

Отдельные инспекции обычно приводят к обнаружению около 60% дефектов, что превышает эффективность других методик за исключением прототипирования и крупномасштабного бета-тестирования. Эти результаты многократно подтверждены в таких организациях, как Harris BCSD, National Software Quality Experiment, Software Engineering Institute, Hewlett Packard и многих других (Shull et al., 2002).

Комбинация инспекций проекта и кода обычно позволяет устранить из продукта 70–85 или более процентов дефектов (Jones, 1996). Инспекции способствуют раннему определению подверженных ошибкам классов, и Кейперс Джонс сообщает, что при использовании инспекций число дефектов в расчете на 1000 строк кода оказывается на 20–30% более низким, чем при использовании менее формальных методик обзора. Участвуя в инспекциях, разработчики, занимающиеся проектированием и кодированием, учатся улучшать свою работу и достигают повышения производительности труда примерно на 20% (Fagan, 1976; Humphrey, 1989; Gilb and Graham, 1993; Wiegers, 2002). Инспекции проектов и кода системы требуют около 10–15% бюджета проекта и обычно снижают общую сумму расходов на реализацию системы.

Инспекции позволяют также оценить прогресс, но только технический. Как правило, для этого нужно узнать, выполняются ли технические аспекты работы и *хорошо* ли они выполняются. Ответы на оба этих вопроса являются побочными продуктами формальных инспекций.

## Роли участников инспекции

Один из важнейших аспектов инспекции состоит в том, что каждый ее участник играет свою особую роль.

***Координатор*** Координатор должен поддерживать темп инспекции достаточно высоким, чтобы она была продуктивной, и в то же время достаточно низким, чтобы участники инспекции могли найти максимум ошибок. Координатор должен обладать адекватной технической компетентностью: он может не быть экспертом в конкретном фрагменте инспектируемого проекта или кода, но обязан понимать соответствующие детали. Координатор также управляет другими аспектами инспекции, такими как распределение фрагментов проекта или кода между инспекторами, распространение контрольных списков инспекции, организация собрания, отчет о результатах инспекции и контроль решения задач, поставленных на инспекционном собрании.

***Автор*** Автор проекта или кода играет во время инспекции относительно небольшую роль. Одной из целей инспекции как раз и является гарантия того, что проект или код говорит сам за себя. Если проект или код, подвергающийся инспекции, кажется неясным, автору поручают его улучшить. Иначе автор должен объяснить не совсем ясные части проекта или кода и, если нужно, рассказать, почему аспекты, которые кажутся ошибочными, на самом деле такими не являются. Если инспекторы плохо знакомы с конкретной частью системы, автор может предоставить им полезную информацию при подготовке к инспекционному собранию.

***Инспектор*** Инспектор имеет прямое отношение к проекту или коду, но не является его автором. Инспектором проекта может быть программист, который будет отвечать за его реализацию. Тестировщики или разработчики высокоуровневой архитектуры также могут быть инспекторами. Задача инспекторов — найти дефекты. Как правило, инспекторы ищут дефекты при подготовке к собранию, однако при обсуждении проекта или кода на собрании группа должна найти значительно больше дефектов.

***Секретарь*** Во время инспекционного собрания секретарь регистрирует обнаруженные ошибки и запланированные действия. Ни автор, ни координатор не должны играть роль секретаря.

***Руководители*** Как правило, руководители не должны участвовать в инспекции. Инспекция ПО — исключительно технический обзор. Присутствие руководителей изменяет все отношения между участниками инспекции: люди, чувствующие, что их оценивают, начинают беспокоиться не об обзоре кода, а совсем о других вещах, в результате чего инспекция становится не техническим, а политическим мероприятием. Однако руководители имеют право знать результаты инспекции, поэтому им следует предоставлять соответствующие отчеты.

Ни при каких обстоятельствах не используйте результаты инспекции для оценки производительности труда. Не режьте курицу, которая несет золотые яйца. Инспектируемый код все еще находится на стадии разработки. Оценка производительности труда должна быть основана на окончательных, а не промежуточных результатах работы.

В инспекции должны участвовать не менее трех человек, потому что роли координатора, автора и инспектора объединять не следует. В то же время к инспекции не следует привлекать более шести человек, потому что группой большего размера слишком трудно управлять. Ученые обнаружили, что наличие более двух-трех инспекторов обычно не повышает эффективность обнаружения дефектов (Bush and Kelly, 1989; Porter and Votta, 1997). Однако это лишь обобщенные данные: по-видимому, оптимальная методика проведения инспекции зависит от типа инспектируемого материала (Wiegers, 2002). Проанализируйте свой опыт и приспособьте эти рекомендации к своей ситуации.

## Общая процедура инспекции

Инспекция включает несколько отдельных этапов.

***Планирование*** Автор проекта или кода предоставляет его координатору. Координатор решает, кто будет выполнять обзор, определяет дату и место проведения инспекционного собрания, после чего предоставляет инспекторам проект или код с контрольным списком, концентрирующим внимание инспекторов на тех или иных аспектах. Инспектируемый материал следует распечатать с номерами строк, чтобы участникам инспекции было проще ориентироваться в нем во время собрания.

***Обзор*** Если инспекторы плохо знакомы с инспектируемым фрагментом системы, автор может посвятить примерно один час описанию технической среды, в которой создан проект или код. Однако наличие подобного обзора может приводить к нежелательным результатам, подталкивая к неверному толкованию не-

ясных моментов инспектируемого проекта или кода. Как я уже отмечал, проект и код должны говорить сами за себя.

**Перекрестная ссылка** Перечень контрольных списков, помогающих повысить качество кода, приведен после содержания книги.

***Подготовка*** Каждый инспектор сам ищет в проекте или коде ошибки, руководствуясь при этом полученным контрольным списком.

Выполняя обзор прикладного кода, написанного на высокоуровневом языке, инспектор может проанализировать за час около 500 строк. При обзоре системного кода, также написанного на высокоуровневом языке, производительность труда инспектора составляет лишь около 125 строк в час (Humphrey, 1989). Наиболее эффективный темп подготовки может колебаться в широком диапазоне, поэтому храните данные о быстроте подготовки к инспекциям в вашей организации — это поможет вам лучше готовиться к будущим инспекциям.

В некоторых организациях было обнаружено, что эффективность инспекций повышается, если поручить каждому инспектору рассмотреть проблему под определенным углом. Так, инспектора можно попросить подойти к инспекции с точки зрения программиста, который будет сопровождать программу, клиента или проектировщика. Пока что эта методика изучена недостаточно полно, но имеющиеся данные говорят о том, что такие обзоры позволяют найти больше ошибок, чем общие обзоры.

Еще одной разновидностью подготовки к инспекции является выполнение каждым инспектором одного или нескольких сценариев. Сценарии могут включать конкретные вопросы, на которые инспектор должен дать ответ, например: «Есть ли требования, которым не удовлетворяет этот проект?» Сценарий может также ставить перед инспектором определенную задачу, такую как составление списка требований, которым удовлетворяет конкретный проект. Вы также можете поручить нескольким инспекторам проанализировать материал с начала до конца, в обратном порядке или «вдоль и поперек».

***Инспекционное собрание*** Координатор поручает одному из участников (не автору) начать изложение проекта или чтение кода (Wiegers, 2003) с объяснением всей логики, в том числе всех ветвей каждой логической структуры. Во время этой презентации секретарь записывает обнаруженные ошибки, но как только участники приходят к выводу, что они нашли ошибку, ее обсуждение прекращается. Секретарь регистрирует тип и серьезность ошибки, и инспекция продолжается. Если инспекция теряет фокус, координатору следует привлечь внимание группы и вернуть обсуждение в нужное русло.

Темп рассмотрения проекта или кода не должен быть ни слишком медленным, ни слишком быстрым. Если темп слишком низок, участники инспекции теряют концентрацию, и продуктивность работы снижается. Если темп слишком высок, группа может упустить ошибки, которые в противном случае были бы обнаружены. Как и темп подготовки, оптимальный темп инспекции зависит от конкретной среды. Храните соответствующие данные, чтобы со временем вы могли определить самую эффективную скорость инспекции в своей организации. В некоторых компаниях было обнаружено, что оптимальная скорость инспекции системного кода равна 90 строкам в час. При инспекции прикладного кода скорость может дости-

гать 500 строк в час (Humphrey, 1989). Если вы только начинаете проводить инспекции, можете ориентироваться на анализ 150–200 непустых и не являющихся комментариями строк исходного кода в час (Wiegers, 2002).

Не обсуждайте на собраниях способы решения проблем. Группа должна сосредоточиться на обнаружении дефектов. В некоторых группах участникам инспекций даже запрещают обсуждать, действительно ли дефект является дефектом. Эти разработчики исходят из того, что любой аспект проекта, кода или документации, который хоть кому-то кажется дефектом, нуждается в пояснении.

Как правило, собрание не должно продолжаться более двух часов. Конечно, это не значит, что по окончании двух часов вы должны подать ложный сигнал пожарной тревоги, но опыт IBM и других компаний показывает, что инспекторы не могут поддерживать нужную концентрацию более двух часов. По этой же причине неразумно проводить более одной инспекции в день.

***Отчет об инспекции*** В день проведения инспекционного собрания координатор составляет отчет об инспекции (электронное письмо или что-либо подобное), указывая в нем все найденные дефекты, их тип и серьезность. Отчет об инспекции помогает гарантировать исправление всех дефектов и облегчает создание контрольного списка, обращающего внимание на проблемы, специфические для организации. Если вы храните данные о времени, затраченном на инспекции, и о числе обнаруженных ошибок, вы сможете подтвердить эффективность инспекций достоверными данными. Иначе вы сможете лишь сказать, что инспекции кажутся оптимальным вариантом. Разумеется, это не убедит сторонников тестирования или других методик. Благодаря отчетам вы также сможете узнать, что инспекции в вашей среде не работают. В этом случае вы можете изменить инспекции или отказаться от них. Сбор данных важен и потому, что любая новая методология должна оправдывать свое использование.

***Исправление дефектов*** Координатор поручает кому-нибудь — обычно автору — исправить все дефекты, указанные в составленном списке.

***Контроль*** Координатор отвечает за контроль решения всех задач, поставленных во время инспекции. В зависимости от числа обнаруженных ошибок и их серьезности вы можете поручить инспекторам провести повторную инспекцию в полном объеме или проинспектировать только исправленные фрагменты. Кроме того, вы можете позволить автору исправить дефекты без всякого контроля.

***Дополнительные собрания*** Хотя во время инспекции участникам не дозволяется обсуждать решения обнаруженных проблем, некоторые разработчики могут испытывать такое желание. Вы можете провести неформальное дополнительное собрание, позволяющее заинтересованным сторонам обсудить решения проблем по окончании официальной инспекции.

### Оптимизация инспекций

Накопив опыт проведения инспекций «по книге», вы скорее всего обнаружите несколько способов их улучшения. Однако изменять инспекции следует дисциплинированно. Адаптируйте процесс выполнения инспекций так, чтобы вы могли узнать, приносят ли изменения выгоду.

Устранение или объединение каких-нибудь этапов инспекции требует затрат, которые часто не оправдываются получаемой выгодой (Fagan, 1986). Если вы испытываете соблазна изменить процесс инспекции без оценки результатов изменения, не делайте этого. Если вы выполнили оценку и увидели, что измененная инспекция более эффективна, — вперед!

Проводя инспекции, вы заметите, что определенные типы ошибок встречаются чаще других. Создайте контрольный список, обращающий внимание инспекторов на эти типы ошибок. Обнаружив новые типы частых ошибок, добавляйте их в список. Если некоторые ошибки из первоначального списка стали встречаться реже, удалите их из списка. После нескольких инспекций вы получите контрольный список, отражающий особенности вашей организации и указывающий на слабые стороны, над которыми следует поработать вашим программистам. Ограничьте контрольный список одной страницей. Более объемные списки трудно использовать на нужном при инспекции уровне детальности.

## Личностные аспекты инспекций

**Дополнительные сведения** Обсуждение обезличенного программирования (egoless programming) можно найти в книге «The Psychology of Computer Programming, 2d ed.» (Weinberg, 1998).

Суть инспекции заключается в обнаружении дефектов в проекте или коде, а не в изучении альтернатив, не в спорах о том, кто прав, а кто виноват, и уж ни в коем случае *не* в критике автора проекта или кода. И для автора, и для других участников инспекция должна быть положительным опытом, укрепляющим командный дух и способствующим обучению. Она не должна вызывать у автора неприязни к членам группы или подталкивать его к поиску новой работы. Комментарии вроде «любому, кто знает Java, известно, что эффективнее организовать цикл от *0* до *num-1*, а не от *1* до *num*» абсолютно неуместны, и в случае надобности координатор должен ясно указать на это.

Критика проекта или кода по вполне понятным причинам будет задевать самолюбие его автора. Автор должен быть готов к тому, что инспекторы могут указать ему на дефекты, которые на самом деле не являются дефектами. Однако автору следует принять к сведению каждый предполагаемый дефект. Это не значит, что автор должен согласиться с сутью критики. Просто во время обзора автору не надо защищать свою работу. После обзора он может обдумать каждое замечание и решить, обосновано ли оно.

Инспекторы должны помнить, что именно автору принадлежит право принятия окончательного решения о том, что делать с дефектом. Пусть инспекторы ищут дефекты (и предлагают решения после обзора), но не покушаются на права автора.

## Инспекции и «Совершенный код»

При подготовке второго издания «Совершенного кода» я на собственном опыте убедился в эффективности инспекций. Работая над первым изданием книги, я писал черновой вариант главы, оставлял его в ящике стола на пару недель, после чего перечитывал и исправлял найденные ошибки. Далее я раздавал проверенную главу примерно десятку коллег, которые выполняли ее обзор, причем некоторые подходили к этому весьма ответственно. В итоге я исправил все обнаруженные ими ошибки. Через несколько недель я самостоятельно выполнил еще один об-

зор и исправил еще ряд ошибок. Наконец я отправил рукопись издателю, и ее проверили литературный редактор, технический редактор и корректор. Книга продается уже более 10 лет, и за это время читатели нашли в ней около 200 неточностей и ошибок.

Трудно поверить, что в книге, прошедшей столько обзоров, оказалось так много ошибок. Увы, это так. Работая над вторым изданием, я решил определить области, на которые нужно обратить особое внимание, и провел формальные инспекции первого издания. Группы из трех-четырех инспекторов подготовились к ним в соответствии с принципами, описанными в этой главе. К моему удивлению, в ходе формальных инспекций мы нашли еще несколько сотен ошибок, которые, несмотря на все приложенные в свое время усилия, все-таки просочились в первое издание.

Если до этого ценность формальных инспекций и вызывала у меня хоть какие-то сомнения, при работе над вторым изданием книги они развеялись.

## Инспекции: резюме

Используемые при инспекциях контрольные списки поддерживают концентрацию разработчиков на важных задачах. Стандартные контрольные списки и стандартные роли обеспечивают систематичность процесса инспекции. Кроме того, наличие петли формальной обратной связи, используемой для улучшения контрольных списков и слежения за темпом подготовки к инспекциям и темпом их проведения, делает процесс инспекции самоорганизующимся. Как бы инспекция ни начиналась, благодаря постоянной оптимизации и высокой эффективности управления процессом инспекции она быстро становится эффективным способом устранения дефектов и ошибок.

Специалисты Института разработки ПО (Software Engineering Institute, SEI) создали модель зрелости процессов (Capability Maturity Model, CMM), позволяющую оценить эффективность процесса разработки ПО (SEI, 1995). Процесс инспекции соответствует самому высокому уровню эффективности. Он является систематичным, повторяется и самооптимизируется на основе обратной связи, поддающейся оценке. Эти идеи вы можете приспособить ко многим методикам, описываемым в данной книге. При распространении на всю компанию эти идеи позволят добиться максимально возможного уровня качества и продуктивности.

**Дополнительные сведения** О разработанной в SEI концепции зрелости процесса разработки см. работу «Managing the Software Process» (Humphrey, 1989).

**Контрольный список: эффективные инспекции**

http://cc2e.com/2199

- ❑ Создали ли вы контрольные списки, обращающие внимание инспекторов на области, с которыми ранее были связаны проблемы?
- ❑ Посвящена ли инспекция обнаружению, а не исправлению дефектов?
- ❑ Рассмотрели ли вы целесообразность использования разных сценариев, помогающих инспекторам сосредоточиться на важных аспектах подготовки?
- ❑ Достаточный ли объем времени предоставляется инспекторам для подготовки к инспекционным собраниям? Все ли инспекторы адекватно готовятся к собраниям?

- ❑ Назначили ли вы каждому участнику инспекции отдельную роль: координатора, инспектора, секретаря и т. д.?
- ❑ Продуктивен ли темп собрания?
- ❑ Ограничили ли вы время собрания двумя часами?
- ❑ Обладают ли все участники инспекции специальными навыками проведения инспекций? Обладает ли координатор навыками координирования инспекций?
- ❑ Собираете ли вы данные о типах ошибок, обнаруженных при каждой инспекции, для адаптации контрольных списков к особенностям своей организации?
- ❑ Собираете ли вы данные о темпе подготовки к инспекции и проведения самой инспекции для оптимизации этих процессов в будущем?
- ❑ Контролируется ли решение задач, поставленных во время инспекции, непосредственно координатором или при помощи повторной инспекции?
- ❑ Понимают ли руководители, что им не следует посещать инспекционные собрания?
- ❑ Составили ли вы план контроля вносимых исправлений, гарантирующий их корректность?

## 21.4. Другие методики совместной разработки ПО

Другие типы совместной разработки исследованы хуже, чем инспекции или парное программирование, поэтому мы рассмотрим их менее подробно. В число методик совместной разработки, описываемых в этом разделе, входят анализ проекта или кода, чтение кода и презентация.

### Анализ проекта или кода

Анализ (walk-through) проекта или кода — популярный тип обзора. Этот термин не имеет точного определения и по крайней мере некоторую часть его популярности можно приписать тому факту, что почти любой тип обзора можно назвать «анализом».

Из-за отсутствия точного определения трудно сказать, что такое анализ. Несомненно, анализ предполагает участие двух или более человек, обсуждающих проект или код. Он может быть совсем неформальным, таким как разговор у доски без какой бы то ни было подготовки. В то же время он может быть очень формализован: примером может служить запланированное собрание с просмотром презентации, подготовленной отделением дизайна, и отправкой формального отчета руководителям. В некотором смысле единственным необходимым условием проведения анализа является «присутствие двух-трех человек в одном месте». Сторонникам анализа нравится расплывчатость такого определения, поэтому я просто укажу некоторые главные аспекты анализа и позволю вам разбираться с остальными деталями самостоятельно:

- за проведение и координацию анализа обычно отвечает автор проекта или кода, подвергающегося обзору;

- предметом анализа являются технические вопросы — это рабочее собрание;
- все участники готовятся к анализу, изучая проект или код и выискивая в нем ошибки;
- анализ позволяет опытным программистам передавать опыт и дух корпоративной культуры молодым программистам; с другой стороны, молодые программисты во время анализа могут предложить новые методологии и подвергнуть сомнению старые и, возможно, уже неэффективные методики;
- как правило, анализ длится от 30 до 60 минут;
- главная цель анализа — обнаружение ошибок, а не их исправление;
- руководители в анализе не участвуют;
- идея анализа обладает значительной гибкостью и легко адаптируется к специфическим потребностям организации.

### Каких результатов ждать от анализа?

При грамотном и дисциплинированном использовании анализ может принести результаты, похожие на результаты инспекции, т. е. в типичной ситуации он позволяет обнаружить в программе 20–40% ошибок (Myers, 1979; Boehm, 1987b; Yourdon, 1989b; Jones, 1996). Тем не менее обычно анализ значительно менее эффективен, чем инспекция (Jones, 1996).

При неразумном использовании анализ невыгоден. Нижняя граница эффективности анализа — 20% — не заслуживает особого внимания, к тому же по крайней мере в одной организации (Boeing Computer Services) взаимные обзоры кода оказались «очень дорогими». Специалисты Boeing обнаружили, что участников проекта было трудно убедить согласованно применять методики анализа, а при повышенном давлении внешних факторов проведение анализа стало практически невозможным (Glass, 1982).

Опыт оказания консалтинговых услуг, накопленный в моей компании за последние 10 лет, заставил меня более критично относиться к анализу. Я обнаружил, что если люди плохо отзывались о технических обзорах, то почти всегда это было связано с неформальными методиками (такими как анализ), а не с формальными инспекциями. По сути обзор является собранием, а проводить собрания дорого. Если вы хотите потратиться на проведение собрания, его следует организовать как формальную инспекцию. Если подвергающийся обзору материал не оправдывает затрат на проведение формальной инспекции, он не оправдывает затрат на проведение собрания вообще. В таких случаях вам лучше использовать чтение документации или другой менее интерактивный подход.

Итак, инспекции обеспечивают более высокую эффективность устранения ошибок, чем анализ. Какая же причина может заставить кого-то использовать анализ?

Если обзор выполняется большой группой разработчиков, анализ — хороший вариант обзора, потому что он позволяет изучить проблему под многими разными углами зрения. Если каждый участник анализа приходит к выводу, что решение не имеет серьезных недостатков, скорее всего так оно и есть.

Если вы привлекаете к обзору специалистов из других организаций, анализ также может быть предпочтительным вариантом. Роли, назначаемые при инспекции, более формализованы, и их эффективное исполнение требует некоторой прак-

тики. Разработчики, не имеющие опыта участия в инспекциях, не смогут показать все, на что они способны. Если вы хотите, чтобы они внесли свой вклад в общее дело, анализ может оказаться оптимальной методикой обзора.

Инспекция более целенаправленна, чем анализ, и обычно более выгодна. Следовательно, если вы выбираете стандартную методику обзора для своей организации, выберите сначала инспекцию, если только у вас нет убедительной причины поступить иначе.

## Чтение кода

Чтение кода — альтернатива инспекции и анализу. Данная методика подразумевает, что вы читаете исходный код в поисках ошибок, обращая при этом внимание и на его качественные аспекты, такие как проект, стиль, удобочитаемость, удобство сопровождения и эффективность.

Исследование, проведенное в Лаборатории проектирования ПО NASA, показало, что при чтении кода разработчики находили около 3,3 дефекта в час, а при тестировании — около 1,8 ошибки в час (Card, 1987). Кроме того, на всем протяжении проекта чтение кода позволяло найти на 20–60% больше ошибок, чем разные виды тестирования.

Как и анализ, чтение кода не имеет точного определения. Обычно при чтении кода двое или более человек независимо изучают код, после чего обсуждают его вместе с автором. Методика чтения кода описана ниже.

- В начале подготовки к собранию автор кода раздает листинги участникам обзора. Листинги включают от 1000 до 10 000 строк; типичный объем — 4000 строк.
- Двое или более разработчиков читают код. Чтобы поддержать дух соревнования, привлекайте к чтению кода минимум двух человек. Если в чтении кода участвуют три человека или более, оценивайте вклад каждого из них, чтобы вы знали, как дополнительные участники влияют на результаты.
- Участники обзора читают код независимо друг от друга. Обычно продуктивность составляет примерно 1000 строк в день.
- После того как участники обзора завершили чтение кода, автор кода проводит собрание. Собрание продолжается один-два часа и фокусируется на проблемах, обнаруженных при чтении кода. Никто не пытается анализировать код строку за строкой. Строго говоря, собрание даже не является необходимостью.
- Автор кода исправляет обнаруженные проблемы.

Различие между чтением кода и инспекцией и анализом в том, что чтение кода в большей степени ориентировано на индивидуальный обзор кода, а не на собрание. Это позволяет каждому участнику обзора уделять больше времени непосредственно поиску проблем. Меньше времени тратится на проведение собраний, предполагающих, что каждый участник активен только часть времени, и требующих значительных усилий для координации действий группы. Меньше времени тратится на отсрочку собрания до тех пор, пока каждый член группы не найдет два свободных часа. Чтение кода особенно полезно, если участников обзора разделяют большие расстояния.

Изучив результаты 13 обзоров, специалисты AT&T обнаружили, что важность собраний, посвященных обзору, была относительно невысокой: 90% дефектов разработчики находили при подготовке к собраниям и только около 10% во время самих собраний (Votta, 1991; Glass, 1999).

### Презентация

Презентацией (dog-and-pony show) называют обзор, при котором система демонстрируется заказчику. Такие обзоры — обычное дело при разработке ПО для правительственных организаций, которые часто требуют выполнения обзоров требований, проектов и кода. Цель презентации — показать заказчику, что работа продвигается успешно, так что это обзор, выполняемый руководителями, а не технический обзор.

Не рассматривайте презентации как способ повышения технического качества продукции. Подготовка к ним может оказывать косвенное влияние на техническое качество, но обычно при этом больше времени тратится на подготовку эффектных слайдов, а не на повышение качества ПО. Для повышения технического качества ПО используйте инспекции, анализ или чтение кода.

## 21.5. Сравнение методик совместного конструирования

Каковы различия между разными методиками совместного конструирования? Основные характеристики каждой методики указаны в табл. 21-1.

**Табл. 21-1. Сравнение методик совместного конструирования**

| Характеристика | Парное программирование | Формальная инспекция | Неформальный обзор (анализ) |
|---|---|---|---|
| Назначаются ли участникам конкретные роли? | Да | Да | Нет |
| Проводится ли формальное обучение исполнению конкретных ролей? | Возможно | Да | Нет |
| Кто «управляет» сотрудничеством? | Разработчик, сидящий за клавиатурой | Координатор | Обычно автор |
| Фокус сотрудничества | Проектирование, кодирование, тестирование и исправление дефектов | Только обнаружение дефектов | Разный |
| Сфокусирован ли обзор? Направляются ли усилия на поиск наиболее частых типов ошибок? | Действия фокусируются неформально, если вообще фокусируются | Да | Нет |
| Выполняется ли контроль исправления найденных ошибок? | Да | Да | Нет |
| Предоставляется ли отдельным программистам подробная обратная связь для сокращения числа ошибок в будущем? | Это второстепенный фактор | Да | Это второстепенный фактор |

(*см. след. стр.*)

**Табл. 21-1.** *(окончание)*

| Характеристика | Парное программирование | Формальная инспекция | Неформальный обзор (анализ) |
|---|---|---|---|
| Используется ли анализ результатов для повышения эффективности процесса? | Нет | Да | Нет |
| Полезна ли методика не на этапе конструирования? | Возможно | Да | Да |
| Типичный процент обнаруживаемых дефектов | 40–60% | 45–70% | 20–40% |

В отличие от формальных инспекций эффективность парного программирования не подтверждена десятилетиями исследований, но первоначальные данные свидетельствуют о том, что оно примерно эквивалентно инспекциям; отзывы компаний, использующих парное программирование, также положительны.

Если парное программирование и формальные инспекции примерно эквивалентны по показателям качества итогового кода, стоимости и быстроты разработки, при выборе одной из этих двух методик можно руководствоваться личными предпочтениями, а не техническими аспектами. Кому-то нравится работать в одиночку, лишь иногда отказываясь от этого режима для проведения инспекционных собраний. Кто-то предпочитает сотрудничество с коллегами. Выбрать методику можно на основе предпочтений конкретных членов группы, а подгруппам можно позволить самим выбрать способ выполнения большей части работы. Можете комбинировать разные методики, если это целесообразно.

# Дополнительные ресурсы

http://cc2e.com/2106

Ниже я указал ряд работ, посвященных методикам совместного конструирования.

## Парное программирование

Williams, Laurie and Robert Kessler. *Pair Programming Illuminated*. Boston, MA: Addison Wesley, 2002. В этой книге подробно рассматриваются все аспекты парного программирования, в том числе соответствие личностей (например, эксперт и новичок, интроверт и экстраверт) и другие вопросы реализации.

Beck, Kent. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison Wesley, 2000. Кент Бек вкратце рассматривает парное программирование и показывает, как его улучшить, дополнив другими методиками, такими как определение стандартов кодирования, частая интеграция и регрессивное тестирование.

Reifer, Donald. «How to Get the Most Out of Extreme Programming/Agile Methods», *Proceedings, XP/Agile Universe 2002*. New York, NY: Springer; pp. 185–196. В этой статье обобщен опыт использования экстремального программирования и методик гибкой разработки, а также указаны условия успешности парного программирования.

## Инспекции

Wiegers, Karl. *Peer Reviews in Software: A Practical Guide*. Boston, MA: Addison Wesley, 2002. В этой книге подробно рассматриваются разные виды обзоров, в том числе формальные инспекции и менее формальные методики. Она основана на тщательных исследованиях, имеет практическую направленность и легко читается.

Gilb, Tom and Dorothy Graham. *Software Inspection*. Wokingham, England: Addison-Wesley, 1993. В этой книге подробно обсуждаются инспекции, выполнявшиеся в начале 1990-х. Она имеет практическую направленность и включает исследования конкретных инспекционных программ, проводившихся в нескольких организациях.

Fagan, Michael E. «Design and Code Inspections to Reduce Errors in Program Development». *IBM Systems Journal*, 15, no. 3 (1976): 182–211.

Fagan, Michael E. «Advances in Software Inspections». *IEEE Transactions on Software Engineering*, SE-12, no. 7 (July 1986): 744–51. Две этих статьи написаны разработчиком методики инспекций. В них вы найдете суть того, что нужно знать для проведения инспекций, в том числе описание всех стандартных форм инспекций.

## Соответствующие стандарты

*IEEE Std 1028-1997* — стандарт обзоров ПО.

*IEEE Std 730-2002* — стандарт планирования контроля качества ПО.

# Ключевые моменты

- Как правило, методики совместной разработки позволяют находить больше дефектов, чем тестирование, и делать это более эффективно.
- Методики совместной разработки и тестирование приводят к обнаружению разных типов ошибок, поэтому программа контроля качества ПО должна включать и обзоры, и тестирование.
- Главными аспектами формальной инспекции, обеспечивающими максимальную эффективность обнаружения дефектов, являются использование контрольных списков, подготовка, назначение хорошо определенных ролей и постоянное улучшение процесса. Формальная инспекция — более эффективный способ обнаружения дефектов, чем анализ проекта или кода.
- Парное программирование и инспекции примерно эквивалентны по показателям расходов и качества итогового кода. Парное программирование может оказаться особенно полезным, если вы хотите сократить срок разработки системы. Некоторые разработчики предпочитают работать в паре, а не самостоятельно.
- Формальные инспекции можно использовать для проверки не только кода, но и других результатов труда, таких как требования, проекты и тесты.
- Анализ и чтение кода — альтернативы инспекциям. Чтение кода позволяет каждому участнику более эффективно использовать свое время.

ГЛАВА 22

# Тестирование, выполняемое разработчиками

http://cc2e.com/2261

**Содержание**



**Связанные темы**

- Качество ПО: глава 20
- Методики совместного конструирования: глава 21
- Отладка: глава 23
- Интеграция: глава 29
- Предварительные условия конструирования: глава 3

Тестирование — самая популярная методика повышения качества, подкрепленная многими исследованиями и богатым опытом разработки коммерческих приложений. Существует множество видов тестирования: одни обычно выполняют сами разработчики, а другие — специализированные группы. Виды тестирования перечислены ниже.

- *Блочным тестированием* называют тестирование полного класса, метода или небольшого приложения, написанного одним программистом или группой, выполняемое отдельно от прочих частей системы.
- *Тестирование компонента* — это тестирование класса, пакета, небольшого приложения или другого элемента системы, разработанного несколькими программистами или группами, выполняемое в изоляции от остальных частей системы.

- *Интеграционное тестирование* — это совместное выполнение двух или более классов, пакетов, компонентов или подсистем, созданных несколькими программистами или группами. Этот вид тестирования обычно начинают проводить, как только созданы два класса, которые можно протестировать, и продолжают до завершения работы над системой.
- *Регрессивным тестированием* называют повторное выполнение тестов, направленное на обнаружение дефектов в программе, уже прошедшей этот набор тестов.
- *Тестирование системы* — это выполнение ПО в его окончательной конфигурации, интегрированного с другими программными и аппаратными системами. Предметом тестирования в этом случае являются безопасность, производительность, утечка ресурсов, проблемы синхронизации и прочие аспекты, которые невозможно протестировать на более низких уровнях интеграции.

В этой главе «тестированием» я называю тестирование, выполняемое разработчиками, которое обычно включает три первых вида тестирования из приведенного списка, но иногда может включать также регрессивное тестирование и тестирование системы. Многие другие виды тестирования обычно выполняют не разработчики, а специализированный персонал: в качестве примеров можно привести бета-тестирование, тестирование системы на предмет одобрения заказчиком, тестирование производительности, тестирование конфигурации, платформенное тестирование, тестирование в стрессовом режиме, тестирование удобства использования и т. д. Эти виды тестирования мы рассматривать не будем.

Тестирование обычно разделяют на две обширных категории: «тестирование методом черного ящика» и «тестирование методом белого (прозрачного) ящика». В первом случае тестировщик не владеет сведениями о внутренней работе тестируемого элемента. Очевидно, что это не так, если вы тестируете собственный код. При тестировании методом белого ящика внутренняя реализация тестируемого элемента тестировщику известна. Тестируя собственный код, вы используете именно этот вид тестирования. Оба вида имеют свои плюсы и минусы; в данной главе основное внимание уделяется тестированию методом белого ящика, потому что именно его выполняют сами разработчики.

Порой термины «тестирование» и «отладка» используют взаимозаменяемо, но внимательные программисты различают два этих процесса. Тестирование — это средство обнаружения ошибок, тогда как отладка является средством поиска и устранения причин уже обнаруженных ошибок. Эта глава посвящена исключительно обнаружению ошибок. Об исправлении ошибок см. главу 23.

Тема тестирования не ограничивается тестированием во время конструирования. Информацию о тестировании системы, тестировании в стрессовом режиме, тестировании методом черного ящика и других вопросах, ориентированных на специалистов по тестированию, можно найти в работах, указанных в разделе «Дополнительные ресурсы» в конце главы.

# 22.1. Тестирование, выполняемое разработчиками, и качество ПО

**Перекрестная ссылка** Об обзорах ПО см. главу 21.

Тестирование — важная часть любой программы контроля качества, а зачастую и единственная. Это печально, так как разнообразные методики совместной разработки позволяют находить больше ошибок, чем тестирование, и в то же время обходятся более чем вдвое дешевле в расчете на одну обнаруженную ошибку (Card, 1987; Russell, 1991; Kaplan, 1995). Каждый из отдельных этапов тестирования (блочное тестирование, тестирование компонентов и интеграционное тестирование) обычно позволяет найти менее 50% ошибок. Комбинация этапов тестирования часто приводит к обнаружению менее 60% ошибок (Jones, 1998).

Программы — не люди, а ошибки — не микробы: программа не может нахвататься ошибок, общаясь с другими дефектными программами. Ошибки всегда допускают программисты.

*Харлан Миллз* (*Harlan Mills*)

Если у программиста спросить, какой из этапов разработки ПО менее всего похож на другие, он наверняка ответит: «Тестирование». По ряду описанных ниже причин большинство разработчиков испытывают при тестировании затруднения.

- Цель тестирования противоположна целям других этапов разработки. Его целью является нахождение ошибок. Успешным считается тест, нарушающий работу ПО. Все остальные этапы разработки направлены на предотвращение ошибок и недопущение нарушения работы программы.
- Тестирование никогда не доказывает отсутствия ошибок. Если вы тщательно протестировали программу и обнаружили тысячи ошибок, значит ли это, что вы нашли все ошибки или в программе все еще остались тысячи других ошибок? Отсутствие ошибок может указывать как на безупречность программы, так и на неэффективность или неполноту тестов.
- Тестирование не повышает качества ПО — оно указывает на качество программы, но не влияет на него. Стремление повысить качество ПО за счет увеличения объема тестирования подобно попытке снижения веса путем более частого взвешивания. То, что вы ели, прежде чем встать на весы, определяет, сколько вы будете весить, а использованные вами методики разработки ПО определяют, сколько ошибок вы обнаружите при тестировании. Если вы хотите снизить вес, нет смысла покупать новые весы — измените вместо этого свой рацион. Если вы хотите улучшить ПО, вы должны не тестировать больше, а программировать лучше.

- Тестирование требует, чтобы вы рассчитывали найти ошибки в своем коде. В противном случае вы, вероятно, на самом деле их не найдете, но это будет всего лишь самоисполняющимся пророчеством. Если вы запускаете программу в надежде, что она не содержит ошибок, их будет слишком легко не заметить. В исследовании, которое уже стало классическим, Гленфорд Майерс попросил группу опытных программистов протестировать программу, содержащую 15 известных дефектов. В среднем программисты нашли лишь 5 из 15 ошибок. Лучший программист обнаружил только 9. Главной причиной неэффективного обнаружения ошибок было недостаточно внимательно

ное изучение ошибочных выходных данных. Ошибки были видны, но программисты их не заметили (Myers, 1978).

Вы должны надеяться обнаружить ошибки в своем коде. Это может казаться неестественным, но лучше уж найти свои ошибки самому, иначе вам на них укажет кто-то другой.

Один из важнейших вопросов состоит в том, сколько времени разработчикам следует уделять тестированию в типичном проекте. Часто говорят, что на все тестирование уходит 50% времени разработки системы, но это может ввести в заблуждение. Во-первых, это число охватывает и тестирование, и отладку; само же тестирование занимает меньше времени. Во-вторых, это число отражает время, которое обычно тратят на тестирование, а не время, которое следует тратить. В-третьих, это число включает и тестирование, выполняемое разработчиками, и независимое тестирование.

В зависимости от объема и сложности проекта тестированию, выполняемому разработчиками, вероятно, следует посвящать от 8 до 25% общего времени работы над проектом. Это согласуется с данными многих авторов (рис. 22-1).

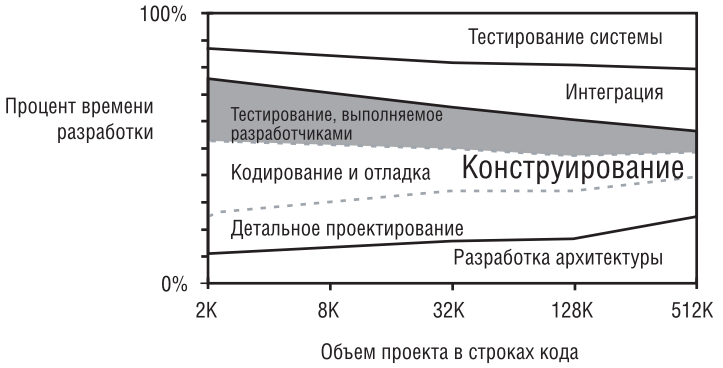



***Рис. 22-1.*** *По мере увеличения объема проекта тестирование, выполняемое разработчиками, отнимает все меньшую часть общего времени разработки. О влиянии размера программы на конструирование см. главу 27*

Другой важный вопрос: что делать с результатами тестирования, выполняемого разработчиками? Прежде всего вы можете сразу использовать эти результаты для оценки надежности разрабатываемой системы. Даже если вы не исправляете дефекты, обнаруженные при тестировании, оно характеризует надежность ПО. Кроме того, результаты тестирования позволяют определить необходимые исправления. Наконец, со временем регистрация дефектов, обнаруженных при тестировании, помогает определить наиболее частые типы ошибок. Вы можете использовать эту информацию для выбора и проведения подходящих обучающих курсов, при подготовке будущих технических обзоров и разработке будущих тестов.

## Тестирование во время конструирования

В большом мире тестирования тему этой главы — тестирование методом белого (или прозрачного) ящика — иногда игнорируют. Как правило, проектировать

классы следует так, чтобы они казались черными ящиками: пользователь класса должен обращаться к нему через интерфейс, не зная о деталях внутреннего устройства класса. Однако при тестировании класса с ним выгодно обращаться как с прозрачным ящиком, анализируя внутренний исходный код, а также входные и выходные данные класса. Зная, что происходит внутри ящика, вы можете протестировать класс более тщательно. Конечно, при тестировании класса вы будете рассматривать его код под тем же углом зрения, что и при написании, поэтому тестирование методом черного ящика также имеет достоинства.

Во время конструирования вы обычно пишете метод или класс, проверяете его в уме, после чего выполняете его обзор или тестирование. Какой бы ни была ваша стратегия интеграционного тестирования или тестирования системы, вы должны тщательно тестировать каждый блок до его объединения с другими блоками. Если вы пишете несколько методов, тестируйте их по одному за раз. На самом деле так их тестировать не легче — зато гораздо легче отлаживать. Если вы объедините несколько непротестированных методов и получите ошибку, вы не сможете определить, какой из методов в ней повинен. Если же вы добавляете в набор протестированных методов по одному методу за раз, при возникновении ошибки вы будете знать, что она содержится в добавленном методе или обусловлена взаимодействием старых методов с новым. Это облегчит отладку.

Методики совместного конструирования имеют много достоинств, не характерных для тестирования. Однако частично это объясняется тем, что тестирование часто выполняют не так хорошо, как было бы можно. Разработчик может выполнить сотни тестов и все же достигнуть лишь частичного покрытия кода тестами. *Чувство* хорошего покрытия кода тестами не означает, что покрытие на самом деле адекватно. Понимание базовых концепций тестирования может повысить его эффективность.

## 22.2. Рекомендуемый подход к тестированию, выполняемому разработчиками

Систематичный подход к тестированию, выполняемому разработчиками, позволяет находить максимальное число дефектов всех типов при минимуме усилий. Поэтому соблюдайте все правила, указанные в следующем списке.

- Тестируйте программу на предмет реализации каждого существенного требования. Планируйте тесты для этого этапа на стадии выработки требований или как можно раньше — лучше всего до написания тестируемого блока. Подумайте о тестах, которые позволили бы выявить распространенные пробелы в требованиях. Уровень безопасности, хранение данных, процедура установки и надежность системы — все эти области тестирования часто упускаются на этапе выработки требований.
- Тестируйте программу на предмет реализации каждого значимого аспекта проектирования. Планируйте тесты для этого этапа на стадии проектирования или как можно раньше — до начала детального кодирования метода или класса, подлежащего тестированию.

- Используя «базисное тестирование», дополните тесты требований и проекта детальными тестами. Разработайте тесты, основанные на потоках данных, а затем создайте остальные тесты, нужные для тщательного тестирования кода. Как минимум вы должны протестировать каждую строку кода. О базисном тестировании и тестировании, основанном на потоках данных, см. ниже.
- Используйте контрольный список ошибок, созданный вами для текущего проекта или в предыдущих проектах.

Проектируйте тесты вместе с системой. Это помогает избегать ошибок в требованиях и проекте, которые обычно дороже ошибок кодирования. Выполняйте тестирование и ищите дефекты как можно раньше, потому что в этом случае исправление дефектов будет дешевле.

## Когда создавать тесты?

Разработчики иногда интересуются, когда лучше создавать тесты: до написания кода или после? (Beck, 2003) Из графика повышения стоимости дефектов (рис. 3-1) следует, что предварительное написание тестов позволяет свести к минимуму интервал времени между моментами внесения дефекта и его обнаружения/устранения. Есть и другие мотивы предварительного написания тестов:

- создание тестов до написания кода требует тех же усилий: вы просто изменяете порядок выполнения этих двух этапов;
- если вы пишете сначала тесты, вы найдете дефекты раньше, да и исправить их легче;
- предварительное написание тестов заставляет хоть немного задумываться о требованиях и проекте до написания кода, что способствует улучшению кода;
- предварительное написание тестов позволяет найти проблемы в требованиях до написания кода, потому что трудно создать тест для плохого требования;
- если вы сохраняете свои тесты, что следует делать всегда, вы можете выполнить тестирование и после написания кода.

По-моему, программирование с изначальными тестами — одна из самых эффективных методик разработки ПО, возникших в последнее десятилетие. Но это не панацея, потому что такой подход тоже страдает от общих ограничений тестирования, выполняемого разработчиками.

## Ограничения тестирования, выполняемого разработчиками

Ниже я описал ряд ограничений этого вида тестирования.

***Разработчики обычно выполняют «чистые тесты»*** Разработчики склонны тестировать код на предмет того, работает ли он (чистые тесты), а не пытаться нарушить его работу всевозможными способами (грязные тесты). В организациях с незрелым процессом тестирования обычно выполняют около пяти чистых тестов на каждый грязный. В организациях со зрелым процессом тестирования на каждый чистый тест обычно приходятся пять грязных. Это отношение изменяется на противоположное не за счет снижения числа чистых тестов, а за счет создания в 25 раз большего числа грязных тестов (данные Бориса Бейзера [Boris Beizer] в Johnson, 1994).

***Разработчики часто имеют слишком оптимистичное представление о покрытии кода тестами*** Как правило, программисты считают, что они достигают 95%-го покрытия кода тестами, но на самом деле они обычно достигают в лучшем случае примерно 80%-го покрытия, в худшем — 30%-го, а в среднем — где-то на 50–60% [данные Бориса Бейзера (Boris Beizer) в Johnson, 1994].

***Разработчики часто упускают из виду более сложные аспекты покрытия кода тестами*** Большинство разработчиков считают тип покрытия кода тестами, известный как «100%-е покрытие операторов», адекватным. Это хорошее начало, но такого покрытия едва ли достаточно. Лучший тип покрытия — так называемое «100%-е покрытие ветвей», требующее, чтобы каждой переменной каждого предиката при тестировании было присвоено хотя бы одно истинное и одно ложное значение (подробнее об этом см. раздел 22.3).

Эти ограничения не уменьшают важность тестирования, выполняемого разработчиками, — они лишь помогают получить о нем объективное представление. Каким бы ценным ни было тестирование, выполняемое разработчиками, его недостаточно для обеспечения адекватного контроля качества, поэтому его следует дополнять другими методиками, в том числе методиками независимого тестирования и совместного конструирования.

## 22.3. Приемы тестирования

Почему невозможно доказать корректность программы, протестировав ее? Чтобы доказать полную работоспособность программы, вы должны были бы протестировать ее со всеми возможными входными значениями и их комбинациями. Даже в случае самой простой программы такое предприятие оказалось бы слишком масштабным. Допустим, ваша программа принимает фамилию, адрес и номер телефона, а затем сохраняет их в файле. Очевидно, что это простая программа — гораздо проще, чем те, о корректности которых приходится беспокоиться в действительности. Предположим далее, что фамилии и адреса могут иметь длину 20 символов, каждый из которых может иметь одно из 26 возможных значений. Число возможных комбинаций входных данных было бы следующим:

| | |
|---|---|
| Фамилия | $26^{20}$ (20 символов; 26 вариантов каждого символа) |
| Адрес | $26^{20}$ (20 символов; 26 вариантов каждого символа) |
| Номер телефона | $10^{10}$ (10 цифр; 10 вариантов каждой цифры) |
| Общее число комбинаций | $= 26^{20} * 26^{20} * 10^{10} » 10^{66}$ |

Даже при таком относительно небольшом объеме входных данных вам пришлось бы выполнить $10^{66}$ тестов. Если бы Ной, высадившись из ковчега, начал тестировать эту программу со скоростью триллион тестов в секунду, на текущий момент он был бы далек от выполнения даже 1% тестов. Очевидно, что при вводе более реалистичного объема данных исчерпывающее тестирование всех комбинаций стало бы еще менее «осуществимым».

## Неполное тестирование

Если уж исчерпывающее тестирование невозможно, на практике искусство тестирования заключается в выборе тестов, способных обеспечить максимальную вероятность обнаружения ошибок. В нашем случае из $10^{66}$ возможных тестов только несколько скорее всего позволили бы найти ошибки, отличающиеся от ошибок, обнаруживаемых другими тестами. Вам нужно выбирать несколько тестов, позволяющих найти разные ошибки, а не использовать множество тестов, раз за разом приводящих к одному результату.

**Перекрестная ссылка** Определить, покрыт ли тестами весь код, позволяет монитор покрытия (см. соответствующий подраздел раздела 22.5).

Планируя тестирование, исключите тесты, которые не могут сообщить ничего нового, например, тесты новых данных, похожих на уже протестированные данные. Существует ряд методов эффективного покрытия базисных элементов; некоторые из них мы и обсудим.

## Структурированное базисное тестирование

Несмотря на пугающее название, в основе структурированного базисного тестирования (structured basis testing) лежит довольно простая идея: вы должны протестировать каждый оператор программы хотя бы раз. Если оператор является логическим, таким как *if* или *while*, вы должны учесть сложность выражения внутри *if* или *while*, чтобы оператор был протестирован полностью. Самый легкий способ покрыть все базисные элементы предполагает подсчет числа возможных путей выполнения программы и создание минимального набора тестов, проверяющих каждый путь.

Возможно, вы слышали о видах тестирования, основанных на «покрытии кода» или «покрытии логики». Эти подходы также требуют тестирования всех путей выполнения программы. В этом смысле они похожи на структурированное базисное тестирование, но они не подразумевают покрытия всех путей *минимальным* набором тестов. Если вы тестируете программу методом покрытия кода или покрытия логики, вы можете создать для покрытия той же логики гораздо больше тестов, чем при использовании структурированного базисного тестирования.

Минимальное число тестов, нужных для базисного тестирования, найти очень просто:

1. начните с 1 для последовательного пути выполнения метода;
2. прибавьте 1 для каждого из ключевых слов *if*, *while*, *repeat*, *for*, *and* и *or* или их аналогов;
3. прибавьте 1 для каждого блока *case*; если отсутствует блок, используемый по умолчанию, прибавьте еще 1.

**Перекрестная ссылка** Эта процедура похожа на методику оценки сложности (см. подраздел «Как измерить сложность» раздела 19.6).

Вот пример:

**Простой пример подсчета числа путей выполнения программы (Java)**

```
Начинаем счет: «1» — сам метод.
Statement1;
Statement2;
```

«2» — оператор *if*.

```
if ( x < 10 ) {
    Statement3;
}
Statement4;
```

В этом случае мы начинаем с 1 и встречаем один оператор *if*, получая в итоге 2. Это значит, что для покрытия всех путей выполнения этого кода вам нужно создать минимум два теста, соответствующих следующим условиям:

- операторы, контролируемые оператором *if*, выполняются (x < 10);
- операторы, контролируемые оператором *if*, не выполняются (x >= 10).

Чтобы у вас сложилось более полное представление об этом виде тестирования, нужно рассмотреть более реалистичный код. В данном случае реализм будет заключаться в наличии дефектов.


Следующий листинг представляет собой более сложный пример. Он используется во многих частях главы и содержит несколько возможных ошибок.

**Пример подсчета числа тестов, нужных для базисного тестирования (Java)**

«1» — сам метод.

```
1  // Вычисление фактической заработной платы.
2  totalWithholdings = 0;
3
```

«2» — цикл *for*.

```
4  for ( id = 0; id < numEmployees; id++ ) {
5
6      // Вычисление суммы взноса в фонд социального страхования.
```

«3» — оператор *if*.

```
7      if ( m_employee[ id ].governmentRetiremenWithheld < MAX_GOVT_RETIREMENT ) {
8          governmentRetirement = ComputeGovernmentRetirement( m_employee[ id ] );
9      }
10
11     // Взнос в пенсионный фонд компании по умолчанию не взимается.
```

```
12    companyRetirement = 0;
13
14    // Вычисление суммы вклада сотрудника в пенсионный фонд компании.
```

«4» — оператор *if*; «5» — операция *&&*.

```
15    if ( m_employee[ id ].WantsRetirement &&
16       EligibleForRetirement( m_employee[ id ] ) ) {
17       companyRetirement = GetRetirement( m_employee[ id ] );
18    }
19
20    grossPay = ComputeGrossPay ( m_employee[ id ] );
21
22    // Вычисление суммы вклада на индивидуальный пенсионный счет сотрудника.
23    personalRetirement = 0;
```

«6» — оператор *if*.

```
24    if ( EligibleForPersonalRetirement( m_employee[ id ] ) ) {
25       personalRetirement = PersonalRetirementContribution( m_employee[ id ],
26          companyRetirement, grossPay );
27    }
28
29    // Вычисление фактической заработной платы сотрудника за неделю.
30    withholding = ComputeWithholding( m_employee[ id ] );
31    netPay = grossPay - withholding - companyRetirement - governmentRetirement -
32       personalRetirement;
33    PayEmployee( m_employee[ id ], netPay );
34
35    // Добавление вычетов из зарплаты сотрудника в соответствующие фонды.
36    totalWithholdings = totalWithholdings + withholding;
37    totalGovernmentRetirement = totalGovernmentRetirement +
         governmentRetirement;
38    totalRetirement = totalRetirement + companyRetirement;
39 }
40
41 SavePayRecords( totalWithholdings, totalGovernmentRetirement, totalRetirement);
```

В этом примере нам нужен один первоначальный тест и один тест для каждого из пяти ключевых слов — всего шесть. Это не значит, что шестью любыми тестами будут покрыты все базисные элементы. Это значит, что нужно минимум шесть тестов. Если тесты не будут разработаны должным образом, они почти наверняка не покроют все базисные элементы. Хитрость в том, что нужно обращать внимание на те же ключевые слова, которые вы использовали при подсчете числа нужных тестов. Все эти ключевые слова представляют нечто, что может быть или истинным, или ложным; убедитесь, что вы разработали хотя бы по одному тесту для каждого истинного и каждого ложного условия.

Вот набор тестов, покрывающий все базисные элементы в этом примере:

| Номер теста | Описание теста | Данные теста |
|---|---|---|
| 1 | Номинальный случай. | Все логические условия истинны |
| 2 | Условие цикла *for* ложно. | *numEmployees < 1* |
| 3 | Условие первого оператора *if* ложно. | *m_employee[ id ].governmentRetirementWithheld >=MAX_GOVT_RETIREMENT* |
| 4 | Условие второго оператора *if* ложно по той причине, что первый операнд операции && ложен. | *not m_employee[ id ].WantsRetirement* |
| 5 | Условие второго оператора *if* ложно по той причине, что второй операнд операции && ложен. | *not EligibleForRetirement( m_employee[id] )* |
| 6 | Условие третьего оператора *if* ложно. | *not EligibleForPersonalRetirement( m_employee[ id ] )* |

**Примечание:** позднее мы дополним эту таблицу другими тестами.

При возрастании сложности метода число тестов, нужных только для покрытия всех путей, быстро увеличивается. Более короткие методы обычно включают меньше путей, которые нужно протестировать. Булевы выражения, не содержащие большого числа операций И и ИЛИ, также имеют меньше вариантов, подлежащих тестированию. Легкость тестирования — еще один хороший повод сокращать методы и упрощать булевы выражения.

Итак, мы разработали для нашего метода шесть тестов, выполнив требования структурированного базисного тестирования. Можно ли считать, что этот метод полностью протестирован? Наверное, нет. Этот тип тестирования гарантирует только выполнение всего кода. Он не учитывает вариации данных.

## Тестирование, основанное на потоках данных

Рассмотрев этот и предыдущий подразделы вместе, вы получите еще одно подтверждение того, что в программировании поток управления и поток данных одинаково важны.

Главная идея тестирования, основанного на потоках данных, в том, что использование данных не менее подвержено ошибкам, чем поток управления. Борис Бейзер утверждает, что по меньшей мере половина всего кода состоит из объявлений данных и операций их инициализации (Beizer, 1990).

Данные могут находиться в одном из трех состояний, которым соответствуют выполняемые над данными действия, указанные ниже.

- **Определение** — данные инициализированы, но еще не использовались.
- **Использование** — данные используются в качестве операндов вычислений, аргументов методов или как-нибудь иначе.
- **Уничтожение** — определенные когда-то данные в результате некоторых операций становятся недействительными. Например, если данными является ука-

затель, соответствующая ему область памяти может быть освобождена. Если это индекс цикла *for*, после выполнения цикла он может остаться за пределами текущей области видимости. Если это указатель на запись в файле, он может стать недействительным в результате закрытия файла.

В дополнение к терминам «определение», «использование» и «уничтожение» удобно иметь термины, описывающие выполнение какого-то действия над переменной сразу после входа в метод или непосредственно перед выходом из него.

- **Вход** — поток управления входит в метод, после чего над переменной сразу выполняется какое-то действие. Пример — инициализация рабочей переменной в начале метода.
- **Выход** — поток управления покидает метод сразу после выполнения операции над переменной. Пример? Присвоение возвращаемого значения переменной статуса в самом конце метода.

### Изменения состояния данных

Нормальный жизненный цикл переменной предполагает, что переменная определяется, используется один или несколько раз и, возможно, уничтожается. Указанные ниже варианты должны вызывать у вас подозрение.

- **Определение — определение** Если вы должны дважды определить переменную, чтобы она получила свое значение, вам следует улучшить не программу, а компьютер! Даже если эта комбинация не является неверной, она впустую тратит ресурсы и подвержена ошибкам.
- **Определение — выход** Если речь идет о локальной переменной, нет смысла ее определять, если вы покидаете метод, не использовав ее. Если это параметр метода или глобальная переменная, возможно, все в порядке.
- **Определение — уничтожение** Определение переменной и ее уничтожение указывает на лишнюю переменную или на отсутствие кода, который должен был ее использовать.
- **Вход — уничтожение** Это проблема, если переменная является локальной. Переменную нельзя уничтожить, если до этого она не была определена или использована. Если, с другой стороны, речь идет о параметре метода или глобальной переменной, эта комбинация нормальна, но только если ранее переменная была определена где-то в другом месте.
- **Вход — использование** Опять-таки, если переменная является локальной, это проблема. Чтобы переменную можно было использовать, ее нужно определить. Если мы имеем дело с параметром метода или глобальной переменной, все нормально, но только если ранее переменная была где-то определена.
- **Уничтожение — уничтожение** Переменные не требуют двойного уничтожения: они не воскресают. Воскресшая переменная — признак небрежного программирования. Кроме того, двойное уничтожение губительно в случае указателей: трудно найти более эффективный способ подвесить компьютер, чем двойное уничтожение (освобождение) указателя.
- **Уничтожение — использование** Вызов переменной после уничтожения — логическая ошибка. Если код все же работает (например, указатель все еще указывает на освобожденную область памяти), это лишь случайность, и, по одно-

му из законов Мерфи, программа потерпит крах тогда, когда это причинит наибольший ущерб.

- **Использование — определение** Определение переменной после использования может быть проблемой, а может и не быть. Это зависит от того, была ли переменная также определена до ее использования. Если вы встречаете такую комбинацию, обязательно проверьте наличие предшествующего определения.

Проверьте код на предмет этих аномальных изменений состояния данных до начала тестирования. После их проверки написание тестов, основанных на потоке данных, сводится к анализу всех возможных комбинаций «определение — использование». Выполнить это можно с различной степенью тщательности.

- Проверка всех определений. Протестируйте каждое определение каждой переменной, т. е. все места, в которых какая-либо переменная получает свое значение. Это слабая стратегия, поскольку, если вы тестируете каждую строку кода, вы и так выполните это.
- Проверка всех комбинаций «определение — использование». Протестируйте все комбинации, включающие определение переменной в одном месте и вызов в другом. Это более грамотная стратегия, так как простое тестирование каждой строки кода далеко не всегда обеспечивает проверку всех комбинаций «определение — использование».

Вот пример:

#### Пример программы, поток данных которой мы протестируем (Java)

```
if ( Condition 1 ) {
   x = a;
}
else {
   x = b;
}
if ( Condition 2 ) {
   y = x + 1;
}
else {
   y = x - 1;
}
```

Для покрытия каждого пути выполнения этой программы нам нужен один тест, при котором условие *Condition 1* истинно, и один — при котором оно ложно, а также аналогичные тесты для условия *Condition 2*. Эти ситуации можно охватить двумя тестами: *(Condition 1=True, Condition 2=True)* и *(Condition 1=False, Condition 2=False)*. Два этих теста — все, что нужно для выполнения требований структурированного базисного тестирования, а также для тестирования всех определений переменных; эти тесты автоматически обеспечивают слабую форму тестирования, основанного на потоках данных.

Однако для покрытия всех комбинаций «определение — использование» этого мало. На текущий момент у нас есть тесты тех случаев, когда условия *Condition 1* и *Condition 2* истинны и когда оба они ложны:

```
x = a
...
y = x + 1
```

и

```
x = b
...
y = x - 1
```

Но у нас есть еще две комбинации «определение — использование»: (1) *x = a*, после чего *y = x - 1* и (2) *x = b*, после чего *y = x + 1*. В данном примере эти комбинации можно покрыть, добавив еще два теста: *(Condition 1=True, Condition 2=False)* и *(Condition 1=False, Condition 2=True)*.

Можно порекомендовать следующую стратегию разработки тестов: начните со структурированного базисного тестирования, которое охватит некоторые, если не все потоки данных «определение — использование». После этого добавьте тесты, нужные для полного охвата комбинаций «определение — использование».

Структурированное базисное тестирование указало нам на шесть тестов метода, для которого мы подсчитывали число тестов. Тестирование каждой пары «определение — использование» требует нескольких дополнительных тестов. Вот пары, которые не охватываются уже созданными тестами:

| Номер теста | Описание теста |
|---|---|
| 7 | Определение переменной *companyRetirement* в строке 12 и использование в строке 26. Этот случай может не покрываться имеющимися тестами. |
| 8 | Определение переменной *companyRetirement* в строке 12 и использование в строке 31. Этот случай может не покрываться имеющимися тестами. |
| 9 | Определение переменной *companyRetirement* в строке 17 и использование в строке 31. Этот случай может не покрываться имеющимися тестами. |

Выполнив тестирование, основанное на потоках данных, несколько раз, вы начнете чувствовать, какие тесты нужны, а какие уже реализованы. В случае сомнений ищите все комбинации «определение — использование». Это может показаться слишком объемной работой, но так вы точно найдете все случаи, не охваченные базисным тестированием.

## Разделение на классы эквивалентности

Хороший тест должен покрывать широкий диапазон входных данных Если два теста приводят к обнаружению одних и тех же ошибок, вам нужен лишь один из них. Понятие «разделения на классы эквивалентности» формализует эту идею и помогает уменьшить число нужных тестов.

**Перекрестная ссылка** Разделение на классы эквивалентности гораздо подробнее обсуждается в книгах, указанных в разделе «Дополнительные ресурсы» в конце этой главы.

Подходящее место для разделения на классы эквивалентности — строка 7 уже известного нам листинга, в которой проверяется условие *m_employee[ ID ].governmentRetirementWithheld < MAX_GOVT_RETIREMENT*. Это условие

делит все значения *m_employee[ ID ].governmentRetirement Withheld* на два класса эквивалентности: значения, которые меньше константы *MAX_GOVT_RETIREMENT*, и значения, которые больше или равны ей. В других частях программы могут использоваться другие классы эквивалентности, что может потребовать тестирования более двух значений *m_employee[ ID ].governmentRetirementWithheld*, но в этой части программы нужно проверить только два значения.

Размышление о разделении на классы эквивалентности не скажет вам много нового о программе, если вы уже покрыли ее базисным тестированием и тестированием, основанным на потоках данных. Однако оно очень полезно, если вы смотрите на программу «извне» (с точки зрения спецификации, а не исходного кода) или если данные сложны, а эта сложность плохо отражена в логике программы.

## Угадывание ошибок

**Перекрестная ссылка** Об эвристических методиках см. раздел 2.2.

Формальные методики тестирования хорошие программисты дополняют менее формальными эвристическими методиками. Одна из них — угадывание ошибок (error guessing). Термин «угадывание ошибок» — довольно примитивное название вполне разумной идеи, подразумевающей создание тестов на основе обдуманных предположений о вероятных источниках ошибок.

Предположения можно выдвигать, опираясь на интуицию или накопленный опыт. Так, в главе 21 в числе прочих достоинств инспекций были указаны создание и обновление списка частых ошибок, используемого при проверке нового кода. Поддерживая списки ранее допущенных ошибок, вы повысите эффективность своих догадок.

Ниже рассматриваются конкретные виды ошибок, угадать которые легче всего.

## Анализ граничных условий

Одной из самых плодотворных областей тестирования являются граничные условия — ошибки занижения или завышения на 1. Действительно, разработчики очень часто используют *num – 1* вместо *num* или >= вместо >.

Идея анализа граничных условий состоит в написании тестов, позволяющих проверить эти условия. Так, при тестировании диапазона значений, которые меньше *max*, возможны три условия:


Как видите, в этой ситуации мы имеем три граничных случая: максимальное значение, которое меньше *max*, само значение *max* и минимальное значение, превышающее *max*. Для исключения распространенных ошибок нужны три теста.

Фрагмент кода, для которого мы подсчитывали число тестов, содержит проверку *m_employee[ ID ].governmentRetirementWithheld < MAX_GOVT_RETIREMENT*. Согласно принципам анализа граничных условий следует изучить три случая:

| Номер теста | Описание теста |
|---|---|
| 1 | Тест требует, чтобы истинное значение выражения *m_employee[ ID ].governmentRetirementWithheld < MAX_GOVT_RETIREMENT* было первым с истинной стороны границы. Иначе говоря, мы должны присвоить элементу *m_employee[ ID ].governmentRetirementWithheld* значение *MAX_GOVT_RETIREMENT - 1*. Для этого годится уже имеющийся тест 1. |
| 3 | Тест требует, чтобы ложное значение выражения *m_employee[ ID ]. governmentRetirementWithheld < MAX_GOVT_RETIREMENT* было первым с ложной стороны границы. Таким образом, элементу *m_employee[ ID ].governmentRetirementWithheld* нужно присвоить значение *MAX_GOVT_RETIREMENT + 1*. Для этого вполне подойдет тест 3. |
| 10 | Дополнительный тест нужен для самой границы, когда *m_employee [ ID ].governmentRetirementWithheld = MAX_GOVT_RETIREMENT*. |

### Сложные граничные условия

Анализ граничных условий можно проводить также в отношении минимального и максимального допустимых значений. В нашем примере ими могли бы быть минимальные или максимальные значения переменных *grossPay*, *companyRetirement* или *personalRetirement*, но из-за того, что эти значения вычисляются вне области видимости метода, их тестирование мы обсуждать не будем.

Более тонкий вид граничного условия имеет место, когда оно зависит от комбинации переменных. Например, что произойдет при умножении двух переменных, если обе являются большими положительными числами? Большими отрицательными числами? Если хотя бы одна из переменных равна 0? Что, если все переданные в метод строки имеют необычно большую длину?

В текущем примере вы могли бы проверить, что происходит с денежными суммами, которые представлены переменными *totalWithholdings, totalGovernmentRetirement* и *totalRetirement*, если каждый член большой группы имеет крупную зарплату — скажем, каждый из программистов зарабатывает по 250 000 долларов (надежда умирает последней!). Для этого нужен еще один тест:

| Номер теста | Описание теста |
|---|---|
| 11 | Большая группа высокооплачиваемых сотрудников (конкретные показатели зависят от конкретной системы — скажем, 1000 сотрудников, каждый из которых зарабатывает по 250 000 долларов в год), не выплачивающих взносы в фонд социального страхования, но отчисляющих средства в пенсионный фонд компании. |

Противоположным тестом из этой же категории было бы вычисление всех показателей для небольшой группы сотрудников, не получающих зарплату:

| Номер теста | Описание теста |
|---|---|
| 12 | Группа из 10 сотрудников, не получающих зарплату. |

## Классы плохих данных

Кроме граничных условий, программу можно тестировать на предмет нескольких других классов плохих данных. Типичными примерами плохих данных можно считать:

- недостаток данных (или их отсутствие);
- избыток данных;
- неверный вид данных (некорректные данные);
- неверный размер данных;
- неинициализированные данные.

Некоторые из этих случаев уже покрыты имеющимися тестами. Так, «недостаток данных» охватывается тестами 2 и 12, а для «неверного размера данных» тесты придумать трудно. И все же рассмотрение классов плохих данных позволяет создать еще несколько тестов:

| Номер теста | Описание теста |
|---|---|
| 13 | Массив из 100 000 000 сотрудников. Тестирование на предмет избытка данных. Конечно, объем данных, который следует считать избыточным, зависит от конкретной системы. |
| 14 | Отрицательная зарплата. Неверный вид данных. |
| 15 | Отрицательное число сотрудников. Неверный вид данных. |

## Классы хороших данных

При поиске ошибок легко упустить из виду тот факт, что номинальный случай также может содержать ошибку. Примерами хороших данных могут служить номинальные случаи, описанные в разделе, посвященном базисному тестированию. Ниже указаны другие виды хороших данных, которые стоит подвергать проверке; проверка каждого из этих видов данных также может приводить к обнаружению ошибок:

- номинальные случаи: средние, ожидаемые значения;
- минимальная нормальная конфигурация;
- максимальная нормальная конфигурация;
- совместимость со старыми данными.

Минимальную нормальную конфигурацию полезно применять для тестирования не только одного элемента, но и группы элементов. Минимальная нормальная конфигурация аналогична граничному условию, составленному из нескольких минимальных значений, но отличается от него тем, что она включает набор минимальных значений из диапазона нормально ожидаемых значений. В качестве примера можно привести сохранение пустой электронной таблицы. При тестировании текстового редактора примером могло бы быть сохранение пустого документа. В нашем примере тестирование минимальной нормальной конфигурации добавило бы в набор следующий тест:

| Номер теста | Описание теста |
|---|---|
| 16 | Группа из 1 сотрудника. Тестирование минимальной нормальной конфигурации. |

Максимальная нормальная конфигурация противоположна минимальной. Она также аналогична граничному условию, но опять-таки включает набор максимальных значений из диапазона ожидаемых значений. Пример — сохранение или

печать электронной таблицы, имеющей «максимальный размер», заявленный в рекламных материалах. В случае текстового процессора — сохранение документа максимального рекомендованного размера. У нас максимальная нормальная конфигурация определяется максимальным нормальным числом сотрудников. Если бы оно равнялось 500, вы добавили бы в набор такой тест:

| Номер теста | Описание теста |
|---|---|
| 17 | Группа из 500 сотрудников. Тестирование максимальной нормальной конфигурации. |

Последний вид тестирования нормальных данных — тестирование совместимости со старыми данными — вступает в игру при замене старого приложения или метода на новый вариант. При вводе старых данных новый метод должен выдавать те же результаты, что и старый метод, за исключением тех ситуаций, в которых старый метод работал ошибочно. Этот вид преемственности версий лежит в основе регрессивного тестирования, призванного гарантировать, что исправления и улучшения поддерживают достигнутый уровень качества, не вызывая проблем. В нашем примере критерий совместимости не добавил бы никаких тестов.

### Используйте тесты, позволяющие легко проверить результаты вручную

Допустим, вы пишете тест для проверки расчета зарплаты; вам нужно ввести зарплату, и один из способов сделать это — ввести числа, которые попадаются под руку. Попробуем:

```
1239078382346
```

Отлично. Это довольно большая зарплата — более триллиона долларов, но если ее «обрезать», можно получить что-то более реалистичное: скажем, 90 783,82 доллара.

Теперь предположим, что этот тест успешен, т. е. указывает на ошибку. Как узнать, что вы обнаружили ошибку? Ну, вы можете вычислить правильный результат вручную и сравнить его с результатом, полученным на самом деле. Однако если в вычислениях фигурируют такие неприятные числа, как 90 783,82 доллара, вероятность допустить ошибку в вычислениях не менее высока, чем вероятность обнаружения ошибки в программе. С другой стороны, удобные круглые числа вроде 20 000 долларов делают вычисления сущим пустяком. Нули легко набирать, а умножение на 2 большинство программистов способны выполнять в уме.

Возможно, вы считаете, что неудобные числа чаще приводят к обнаружению ошибок, но это не так: при использовании любого числа из одного и того же класса эквивалентности вероятность нахождения ошибки одинакова.

## 22.4. Типичные ошибки

Главная идея этого раздела в том, что для достижения максимальной эффективности тестирования мы должны как можно больше знать о нашем враге — ошибках.

## Какие классы содержат наибольшее число ошибок?

Естественно предположить, что дефекты распределяются по коду равномерно. Если код содержит в среднем 10 дефектов на 1000 строк, вы могли бы ожидать, что класс из 100 строк будет иметь один дефект. Это естественное предположение, но оно ошибочно.

Кейперс Джонс сообщает, что в результате принятой в IBM программы повышения качества 31 из 425 классов системы IMS получил статус «подверженный ошибкам». Эти классы были исправлены или полностью разработаны заново, благодаря чему менее чем через год частота обнаружения дефектов в IMS клиентами снизилась в 10 раз. Общие расходы на сопровождение системы снизились примерно на 45%. Мнения клиентов о качестве системы изменились с «неприемлемо» на «хорошо» (Jones, 2000).

Большинство ошибок обычно концентрируется в нескольких особенно дефектных методах. Типичные отношения между ошибками и кодом таковы:

- 80% ошибок содержится в 20% классов или методов проекта (Endres, 1975; Gremillion, 1984; Boehm, 1987b; Shull et al., 2002);
- 50% ошибок содержится в 5% классов проекта (Jones, 2000).

Эти отношения могут казаться не такими уж и важными, пока вы не узнаете несколько следствий. Во-первых, 20% методов проекта обусловливают 80% затрат на разработку (Boehm, 1987b). Это не значит, что 20% самых дорогих методов являются одновременно и самыми дефектными, но такое совпадение настораживает.

Во-вторых, какой бы ни была точная доля расходов, приходящихся на разработку дефектных методов, эти методы очень дороги. В классическом исследовании, проведенном в 1960-х, специалисты IBM проанализировали операционную систему OS/360 и обнаружили, что ошибки не были распределены равномерно между всеми методами, а были сконцентрированы в нескольких методах. Был сделан вывод, что эти методы — «самые дорогие сущности в программировании» (Jones, 1986a). Они содержали целых 50 дефектов на 1000 строк кода, а их исправление часто оказывалось в 10 раз дороже разработки всей системы (затраты включали поддержку пользователей и сопровождение системы на месте).

**Перекрестная ссылка** Другим типом методов, часто содержащих много ошибок, являются слишком сложные методы. Об идентификации таких методов и их упрощении см. подраздел «Общие принципы уменьшения сложности» раздела 19.6.

В-третьих, дорогие методы оказывают очевидное влияние на процесс разработки. Как гласит старая пословица, «время — деньги». Справедливо и обратное: «деньги — время», и, если вы можете исключить почти 80% затрат, избежав проблемных методов, вы можете сэкономить и много времени. Это наглядно иллюстрирует Главный Закон Контроля Качества ПО: повышение качества сокращает сроки и снижает общую стоимость разработки системы.

В-четвертых, проблемные методы оказывают не менее очевидное влияние на сопровождение программы. При сопровождении программистам приходится сосредоточиваться на идентификации методов, подверженных ошибкам, их перепроектировании и переписывании с нуля. В вышеупомянутом проекте IMS после замены дефектных классов производительность труда при разработке новых версий IMS повысилась примерно на 15% (Jones, 2000).

## Классификация ошибок

**Перекрестная ссылка** Список всех контрольных списков приведен после содержания книги.

Некоторые ученые попытались классифицировать ошибки по типу и определить распространенность ошибок каждого типа. У каждого программиста есть свой список наиболее неприятных ошибок: ошибки занижения или завышения на 1, невыполнение повторной инициализации переменной цикла и т. д. В контрольных списках я указал и многие другие типы ошибок.

Борис Бейзер объединил данные нескольких исследований и пришел к исключительно подробной классификации ошибок по распространенности (Beizer, 1990). Вот резюме его результатов:

| | |
|---|---|
| 25,18% | Структурные ошибки |
| 22,44% | Ошибки в данных |
| 16,19% | Ошибки в реализации функциональности |
| 9,88% | Ошибки конструирования |
| 8,98% | Ошибки интеграции |
| 8,12% | Ошибки в функциональных требованиях |
| 2,76% | Ошибки в определении или выполнении тестов |
| 1,74% | Системные ошибки, ошибки в архитектуре ПО |
| 4,71% | Другие ошибки |

Бейзер сообщил свои результаты с точностью до двух разрядов после запятой, но исследования типов ошибок в целом не позволяют сделать окончательный вывод. Разные ученые сообщают о разных ошибках, а результаты исследований похожих типов ошибок иногда различаются на 50%, а не на сотые доли процента.

Из-за таких больших различий выполненное Бейзером объединение результатов ряда исследований, вероятно, не способно дать нам точной информации. Но даже если имеющиеся данные неокончательны, на их основе мы можем сделать несколько общих выводов.

***Большинство ошибок имеет довольно ограниченную область видимости*** Одно исследование показало, что в 85% случаев исправление ошибки требовало изменения только одного метода (Endres, 1975).

***Многие ошибки не связаны с конструированием*** Опрос 97 разработчиков позволил выяснить, что тремя наиболее частыми причинами ошибок были плохое знание прикладной области, изменения или конфликты требований, а также неэффективность общения и плохая координация действий разработчиков (Curtis, Krasner, and Iscoe, 1988).

Если вы видите следы копыт, думайте о лошадях, а не о зебрах. Скорее всего ОС работает нормально. С базой данных тоже, наверное, все в порядке.

*Энди Хант и Дэйв Томас* (*Andy Hunt and Dave Thomas*)

***Как правило, ошибки конструирования лежат на совести программистов*** В двух давнишних исследованиях было установлено, что из всех ошибок 95% были допущены программистами, причиной 2% было системное ПО (компилятор и ОС), еще 2% — другое ПО, а оставшегося 1% — оборудование (Brown and Sampson, 1973; Ostrand and Weyuker, 1984). Системное ПО и инструменты разработки используются сегодня гораздо шире, чем в 1970-х и 1980-х,

поэтому я думаю, что сегодня программисты несут ответственность за еще больший процент ошибок.

***На удивление распространенной причиной проблем являются опечатки*** В одном исследовании было обнаружено, что 36% всех ошибок конструирования были опечатками (Weiss, 1975). Исследование почти 3 000 000 строк приложения для расчета динамики полета, проведенное в 1987 г., показало, что опечатками были 18% всех ошибок (Card, 1987). В другом исследовании было установлено, что 4% всех ошибок были орфографическими ошибками в сообщениях (Endres, 1975). В одной из моих программ коллега обнаружил ряд орфографических ошибок, просто проанализировав все строки исполняемого файла утилитой проверки орфографии. Внимание к деталям на самом деле важно. Если вы сомневаетесь в этом, учтите, что три самых дорогостоящих ошибки всех времен, приведших к убыткам объемом 1,6 миллиарда, 900 миллионов и 245 миллионов долларов, были вызваны изменением *одного символа* в ранее корректных программах (Weinberg, 1983).

***Довольно часто причиной ошибок является неправильное понимание проекта*** Компилятивное исследование Бейзера показало, что 16% ошибок было обусловлено неправильной интерпретацией проекта (Beizer, 1990). В другом исследовании было обнаружено, что неправильным пониманием проекта объяснялось 19% ошибок (Weiss, 1975). Посвящайте анализу проекта столько времени, сколько нужно. Это не обеспечивает немедленную выгоду (кому-то даже может показаться, что вы не работаете!), но в итоге окупается с избытком.

***Большинство ошибок легко исправить*** Примерно в 85% случаев на исправление ошибки требуется менее нескольких часов. Где-то в 15% случаев — от нескольких часов до нескольких дней. И только около 1% ошибок требует большего времени (Weiss, 1975; Ostrand and Weyuker, 1984; Grady, 1992). Это подтверждает и Барри Бом, заметивший, что на исправление около 20% ошибок уходит около 80% ресурсов (Boehm, 1987b). Изо всех сил старайтесь избегать трудных ошибок, выполняя предварительные обзоры требований и проектов. Исправляйте многочисленные мелкие ошибки так эффективно, как только можете.

***Оценивайте опыт борьбы с ошибками в своей организации*** Разнообразие результатов, приведенных в этом подразделе, говорит о том, что каждая организация имеет собственный, уникальный опыт борьбы с ошибками. Это осложняет использование опыта других организаций в ваших условиях. Некоторые результаты противоречат интуиции; возможно, вам следует дополнить свои интуитивные представления другими средствами. Начните оценивать процесс разработки в своей организации, чтобы знать причины проблем.

## Доля ошибок, обусловленных неграмотным конструированием

Как и классификация ошибок, данные, соотносящие ошибки с разными этапами разработки, не окончательны. Но то, что с конструированием всегда связано большое число ошибок, сомнений не вызывает. Иногда программисты утверждают, что ошибки конструирования дешевле исправлять, чем ошибки, допущенные при вы-

работке требований или проектировании. Возможно, исправить отдельную ошибку конструирования и дешевле, но при учете всех ошибок ситуация меняется.

Ниже приведены некоторые мои выводы.

- В небольших проектах дефекты конструирования составляют большинство всех ошибок. В одном исследовании ошибок кодирования, допущенных в небольшом проекте (1000 строк кода), было обнаружено, что 75% дефектов пришлось на этап кодирования, тогда как на этап выработки требований — 10%, а на этап проектирования — 15% (Jones, 1986a). По-видимому, такое распределение ошибок характерно для многих небольших проектов.
- Дефекты конструирования составляют минимум 35% всех дефектов независимо от размера проекта. В крупных проектах доля дефектов конструирования не так велика, как в небольших, но и тогда она равна минимум 35% (Beizer, 1990; Jones, 2000). Некоторые ученые сообщают, что даже в очень крупных проектах дефектами конструирования являются около 75% всех ошибок (Grady, 1987). Обычно чем лучше разработчики знают прикладную область, тем лучше они проектируют общую архитектуру системы. В этом случае на детальное проектирование и кодирование приходится еще больший процент ошибок (Basili and Perricone, 1984).
- Хотя ошибки конструирования и дешевле исправлять, чем ошибки выработки требований и проектирования, это все равно дорого. Исследование двух очень крупных проектов, проведенное в Hewlett-Packard, показало, что стоимость исправления среднего дефекта конструирования составляла 25–50% от стоимости исправления средней ошибки проектирования (Grady, 1987). При учете большего числа дефектов конструирования общая стоимость их исправления могла вдвое превышать стоимость исправления дефектов проектирования.

Вот примерное отношение между объемом проекта и распределением ошибок (рис. 22-2):

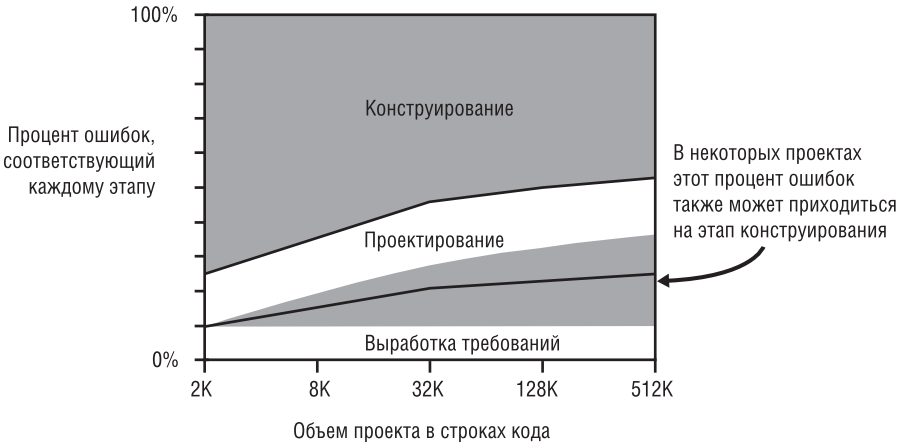



***Рис. 22-2.*** *По мере увеличения объема проекта доля ошибок, допускаемых во время конструирования, снижается. Тем не менее ошибки конструирования составляют 45–75% всех ошибок даже в самых крупных проектах*

## Сколько ошибок можно ожидать?

Ожидаемое число ошибок зависит от качества вашего процесса разработки. Вот некоторые соображения по этому поводу.

- Средний для отрасли показатель таков: примерно 1–25 ошибок на 1000 строк кода в готовом ПО, разрабатывавшегося с использованием разных методик (Boehm, 1981; Gremillion, 1984; Yourdon, 1989a; Jones, 1998; Jones, 2000; Weber, 2003). Проекты, в которых обнаруживается 0,1 от указанного числа ошибок, редки; о случаях, когда ошибок в 10 раз больше, предпочитают не сообщать (вероятно, такие проекты даже не доводят до конца!).
- Отделение прикладного ПО компании Microsoft сообщает о таких показателях, как 10–20 дефектов на 1000 строк кода во время внутреннего тестирования и 0,5 дефекта на 1000 строк кода в готовой продукции (Moore, 1992). Для достижения этого уровня применяется комбинация методик чтения кода, описанных в разделе 21.4, и независимого тестирования.
- Харлан Миллз придумал «разработку методом чистой комнаты» — методику, позволяющую достигнуть всего лишь 3 дефектов на 1000 строк кода во время внутреннего тестирования и 0,1 дефекта на 1000 строк кода в готовой системе (Cobb and Mills, 1990). В нескольких проектах — например, в проекте разработки ПО для космических кораблей — благодаря сочетанию методик формальной разработки, обзоров кода коллегами и статистического тестирования был достигнут такой уровень, как 0 дефектов на 500 000 строк кода (Fishman, 1996).
- Уоттс Хамфри (Watts Humphrey) сообщает, что группы, применяющие методику Team Software Process (TSP), достигают уровня 0,06 дефекта на 1000 строк кода. Главный аспект TSP — обучение разработчиков изначальному предотвращению дефектов (Weber, 2003).

Результаты проектов TSP и проектов «чистой комнаты» подтверждают другую версию Главного Закона Контроля Качества ПО: дешевле сразу создать высококачественную программу, чем создать низкокачественную программу и исправлять ее. Производительность труда в случае одного полностью проверенного проекта из 80 000 строк, разработанного методом «чистой комнаты», оказалась равной 740 строкам кода на человеко-месяц. В то же время средняя для отрасли скорость разработки полностью проверенного кода, учитывающая все затраты, не связанные с кодированием, близка к 250–300 строкам на человеко-месяц (Cusumano et al., 2003). Экономия затрат и повышение производительности труда при использовании методик TSP или «чистой комнаты» объясняются тем, что эти методики почти исключают затраты времени на отладку. Что? Исключают затраты времени на отладку? Это действительно стоящая цель!

## Ошибки самого тестирования

Вероятно, вам знакома следующая ситуация. В программе имеется ошибка. У вас сразу же возникают некоторые предположения о ее причинах, но весь код кажется правильным. Пытаясь изолировать ошибку, вы выполняете еще несколько тестов, но все они дают правильные результаты. Вы проводите несколько часов за чтением кода и пересчетом результатов вручную, но безрезультатно. Еще через несколько часов что-то заставляет вас проверить тесто-

вые данные. Эврика! Ошибка в тестовых данных! Как глупо тратить часы на поиск ошибки, если она кроется в тестовых данных, а не в коде!

Это довольно распространенная ситуация. Зачастую сами тесты содержат не меньше, а то и больше ошибок, чем тестируемый код (Weiland, 1983; Jones, 1986a; Johnson, 1994). Объяснить это легко, особенно если разработчики сами пишут тесты. Тесты обычно создают на ходу, не уделяя должного внимания проектированию и конструированию. Их часто считают одноразовыми и разрабатывают с соответствующей тщательностью.

Ниже дано несколько советов по снижению числа ошибок в тестах.

***Проверяйте свою работу*** Разрабатывайте тесты так же тщательно, как и код, и внимательно проверяйте их. Анализируйте код каждого теста строка за строкой при помощи отладчика, как вы проверяли бы код готового приложения. Проводите анализ и инспекцию тестовых данных.

***Планируйте тестирование программы так же, как и ее разработку*** Планирование тестирования следует начинать на этапе выработки требований или сразу после получения задания. Это поможет избавиться от тестов, основанных на неверных предположениях.

***Храните тесты*** Посвятите немного времени проверке качества тестов. Сохраните их для регрессивного тестирования и для работы над будущими версиями программы. Связанные с этим проблемы легко оправдать, если вы знаете, что собираетесь сохранить тесты, а не выбросить.

***Встраивайте блочные тесты в среду тестирования*** Пишите сначала код блочных тестов, но затем интегрируйте их в системную среду тестирования (такую как JUnit). Использование интегрированной среды тестирования предотвращает только что упомянутую тенденцию выбрасывать тесты.

# 22.5. Инструменты тестирования

В этом разделе рассматриваются типы инструментов тестирования, которые вы можете купить или создать. Конкретные инструменты названы здесь не будут, потому что ко времени издания книги они вполне могут устареть. Для получения свежих сведений обратитесь к своему любимому журналу по программированию.

## Создание лесов для тестирования отдельных классов

Термин «леса» (scaffolding) пришел в программирование из строительства. Строительные леса создаются для того, чтобы рабочие получили доступ к определенным частям здания. В программировании леса создаются исключительно для упрощения тестирования кода.

Одним из типов лесов является объект, используемый другим, тестируемым объектом. Такой объект называется «поддельным объектом» (mock object) или «объектом-заглушкой» (stub object) (Mackinnon, Freemand, and Craig, 2000; Thomas and Hunt, 2002). С аналогичной целью можно создавать и низкоуровневые методы, называемые «заглушками». Поддельные объекты или методы-заглушки можно делать более или

**Дополнительные сведения** Несколько хороших примеров лесов можно найти в эссе Джона Бентли «A Small Matter of Programming» в книге «Programming Pearls, 2d ed.» (Bentley, 2000).

менее реалистичными в зависимости от требуемой степени достоверности. Такие леса могут:

- немедленно возвращать управление, не выполнив никаких действий;
- тестировать полученные данные;
- выводить диагностическое сообщение (например, значения входных параметров) на экран или записывать в файл;
- запрашивать возвращаемые значения у программиста;
- возвращать стандартный ответ независимо от полученных данных;
- впустую тратить ресурсы процессора, выделенные реальному объекту или методу;
- играть роль более медленной, объемной, простой или менее точной версии реального объекта или метода.

Другим типом лесов является поддельный метод, вызывающий реальный тестируемый метод. Такой метод, называемый «драйвером» или иногда «тестовой сбруей» (test harness), может:

- вызывать объект, передавая ему некоторые данные;
- запрашивать информацию у программиста и передавать ее объекту;
- принимать параметры командной строки (если ОС это поддерживает) и передавать их объекту;
- читать аргументы из файла и передавать их объекту;
- вызывать объект несколько раз, передавая ему при каждом вызове новый предопределенный набор входных данных.

**Перекрестная ссылка** Грань между инструментами тестирования и инструментами отладки размыта. Об инструментах отладки см. раздел 23.5.

Последний тип лесов, фиктивный файл (упрощенная версия реального полноразмерного файла), включает такие же элементы. Небольшой фиктивный файл обладает двумя достоинствами. Его небольшой объем позволяет держать в уме все его содержимое и облегчает проверку правильности файла. А так как файл создается специально для тестирования, вы можете спроектировать его так, чтобы любая ошибка в его использовании бросалась в глаза.

http://cc2e.com/2268

Очевидно, что создание лесов требует некоторой работы, но вы сможете повторно использовать их даже при обнаружении ошибки в классе. Кроме того, существуют многие инструменты, упрощающие создание поддельных объектов и других видов лесов. При тестировании класса леса позволяют исключить его взаимодействие с другими классами. Леса особенно полезны, если в программе применяются утонченные алгоритмы. Так, если тестируемый код встроен в другой блок кода, тесты могут выполняться неприемлемо медленно. Леса позволяют тестировать код непосредственно. Несколько минут, потраченных на создание лесов для тестирования кода, скрытого в самых глубинах программы, могут сэкономить много часов на отладке.

Для создания лесов годится любая из многих доступных сред тестирования (JUnit, CppUnit, NUnit и т. д.). Если ваша среда разработки не поддерживается ни одной из существующих сред тестирования, вы можете написать несколько методов класса и включить в файл метод *main()* для их тестирования, пусть даже эти методы не

будут в итоговой программе работать сами по себе. Метод *main()* может читать параметры командной строки и передавать их в тестируемый метод, позволяя проверить его до интеграции с остальной частью программы. Выполняя интеграцию, оставьте методы и предназначенные для их тестирования леса в файле, и заблокируйте леса при помощи директив препроцессора или комментариев. В итоге код лесов будет проигнорирован при компиляции, а если вы разместите его в конце файла, он и на глаза не будет попадаться. Оставленный в файле, он не причинит никакого вреда. Вам не придется тратить время на его удаление и архивирование, а в случае чего он всегда будет под рукой.

## Инструменты сравнения файлов

Регрессивное (или повторное) тестирование значительно облегчают автоматизированные инструменты сравнения действительных выходных данных с ожидаемыми. Данные, выводимые на печать, легко проверить, перенаправив их в файл и сравнив при помощи diff или другой утилиты сравнения файлов с другим файлом, в который ранее были записаны ожидаемые данные. Если файлы различаются, радуйтесь: вы обнаружили регрессивную ошибку.

**Перекрестная ссылка** О регрессивном тестировании см. подраздел «Повторное (регрессивное) тестирование» раздела 22.6.

## Генераторы тестовых данных

Вы также можете написать код для систематичного тестирования выбранных фрагментов программы. Несколько лет назад я разработал собственный алгоритм шифрования и написал на его основе программу шифрования файлов. Программа должна была так кодировать файл, чтобы его можно было декодировать, только введя правильный пароль. При шифровании содержимое файла изменялось самым радикальным образом. Было очень важно, чтобы программа декодировала файл правильно, потому что в противном случае он был бы испорчен.

http://cc2e.com/2275

Я настроил генератор тестовых данных, который полностью тестировал шифровальную и дешифровальную части программы. Он генерировал файлы, состоящие из случайных символов и имеющие случайный объем в пределах от 0 до 500 кб. Он генерировал случайные пароли случайной длины в диапазоне от 1 до 255 символов. Для каждого случая он генерировал две копии случайного файла, шифровал одну копию, заново инициализировался, дешифровал копию и затем сравнивал дешифрованную копию с первоначальной. Если какие-нибудь байты различались, генератор печатал всю информацию, нужную мне для воспроизведения ошибки.

Я настроил генератор так, чтобы средний объем тестируемых файлов равнялся 30 кб. Если бы я этого не сделал, файлы были бы равномерно распределены между 0 кб и 500 кб и имели средний объем 250 кб. Сокращение среднего объема позволило мне быстрее тестировать файлы, пароли, признаки конца файла, необычные объемы файлов и прочие возможные причины ошибок.

Результаты не заставили себя ждать. Выполнив лишь около 100 тестов, я нашел две ошибки. Обе возникали в специфических ситуациях, которые могли ни разу не случиться на практике, однако это все же были ошибки, и я был рад их обнаружить. После их исправления я тестировал программу несколько недель, зашиф-

ровав и дешифровав около 100 000 файлов без каких бы то ни было ошибок. Учитывая то, что я протестировал файлы и пароли самого разного объема и содержания, я мог быть уверен, что с программой все в порядке.

Из этой истории можно извлечь ряд уроков, в том числе следующие.

- Правильно спроектированные генераторы случайных данных способны генерировать комбинации тестовых данных, о которых вы могли бы не подумать.
- Генераторы случайных данных могут протестировать программу тщательнее, чем вы сами.
- Со временем вы можете улучшить случайные тесты, обратив повышенное внимание на реалистичный диапазон входных значений. Это позволяет сконцентрировать тестирование на тех областях, которые будут использоваться чаще всего, и максимально повысить их надежность.
- Модульное проектирование сполна окупается во время тестирования. Я смог отделить код шифрования и дешифрования и задействовать его независимо от кода пользовательского интерфейса, что упростило задачу написания тестового драйвера.
- Вы можете повторно использовать тестовый драйвер даже после изменения тестируемого им кода. Как только я исправил две ошибки, я смог сразу же начать повторное тестирование.

## Мониторы покрытия кода тестами

Карл Вигерс сообщает, что тестирование, выполняемое без измерения покрытия кода тестами, обычно охватывает только 50–60% кода (Wiegers, 2002). Монитор покрытия — это инструмент, который следит за тем, какой код тестировался, а какой нет. Монитор покрытия особенно полезен для систематичного тестирования, потому что он говорит вам, полностью ли код покрывается конкретным набором тестов. Если вы выполнили полный набор тестов, а монитор покрытия сообщает, что какой-то код все еще не протестирован, значит, нужны дополнительные тесты.

## Регистраторы данных

http://cc2e.com/2282

Некоторые инструменты могут следить за программой и собирать информацию о ее состоянии в случае краха подобно «черному ящику», устанавливаемому в самолетах для определения причин крушения. Эффективная регистрация данных облегчает диагностику ошибок и сопровождение программы после ее выпуска.

Вы можете создать собственный регистратор данных, записывающий сведения о важных событиях в файл. Записывайте в файл состояние системы до ошибки и подробные сведения об условиях возникновения ошибки. Можете использовать эту функциональность в предварительных версиях программ и блокировать в окончательных версиях. Если вы используете для хранения зарегистрированных данных самоочищающееся хранилище и тщательно продумали размещение и содержание сообщений об ошибках, можете оставлять функции регистрации данных и в итоговых версиях программ.

## Символические отладчики

Символический отладчик — это технологическое дополнение анализа и инспекций кода. Отладчик может выполнять код строка за строкой, позволяет следить за значениями переменных и всегда интерпретирует код так же, как компьютер. Возможность пошагового выполнения кода и наблюдения за его работой трудно переоценить.

**Перекрестная ссылка** Наличие отладчиков зависит от зрелости технологической среды (см. раздел 4.3).

Анализ кода при помощи отладчика во многом напоминает процесс обзора вашего кода другими программистами. Ни ваши коллеги, ни отладчик не имеют тех же слабых сторон, которые имеете вы. Дополнительное преимущество отладчика в том, что анализ кода с его помощью менее трудоемок, чем групповой обзор. Наблюдение за выполнением кода при разных комбинациях входных данных позволяет убедиться в том, что вы написали то, что хотели.

Кроме того, использование хорошего отладчика является эффективным способом изучения языка, поскольку вы можете получить точную информацию о выполнении кода. Переключаясь между высокоуровневым языком и ассемблером, вы можете изучить соответствие высокоуровневых и низкоуровневых конструкций. Наблюдение за регистрами и стеком позволяет лучше разобраться в передаче аргументов в методы. Вы можете увидеть, как компилятор оптимизирует код. Никакое из этих преимуществ не имеет прямого отношения к главной роли отладчика — диагностике уже обнаруженных ошибок, но при творческом подходе из отладчика можно извлечь гораздо большую выгоду.

## Инструменты возмущения состояния системы

Другой класс инструментов тестирования предназначен для приведения системы в возмущенное состояние. Многие программисты могут рассказать истории о программах, которые работают 99 раз из 100, но при сотом запуске с теми же данными терпят крах. Почти всегда причиной этого является невыполнение инициализации какой-то переменной, и обычно эту ошибку очень трудно воспроизвести, потому что в 99 случаях из 100 неинициализированная переменная получает нулевое значение.

http://cc2e.com/2289

Инструменты тестирования из этого класса могут иметь самые разнообразные возможности.

- **Заполнение памяти** Эта функция помогает находить неинициализированные переменные. Некоторые инструменты перед запуском программы заполняют память произвольными значениями, чтобы неинициализированные переменные не получили случайно значение 0. Иногда память целесообразно заполнять конкретным значением. Например, в случае процессоров с архитектурой x86 значение 0xCC соответствует машинному коду команды прерывания. Если вы заполните память значением 0xCC и программа попробует выполнить что-то, чего выполнять не следует, вы натолкнетесь в отладчике на точку прерывания и обнаружите ошибку.
- **«Встряхивание» памяти** В многозадачных средах некоторые инструменты могут переупорядочивать выделенную программе память, что позволяет гаран-

тировать отсутствие кода, требующего, чтобы данные находились по абсолютным, а не относительным адресам.

- **Селективный сбой памяти** Драйвер памяти позволяет имитировать недостаток или неудачный запрос памяти, может отказывать в запросе памяти после произвольного числа успешных запросов или предоставлять запрошенную память после произвольного числа неудовлетворенных запросов. Это особенно полезно при тестирования сложных программ, выделяющих память динамически.
- **Проверка доступа к памяти (проверка границ)** Инструменты проверки границ следят за операциями над указателями, гарантируя их корректность. Такие инструменты полезны при поиске неинициализированных или «висячих» указателей.

### Базы данных для хранения информации об ошибках

http://cc2e.com/2296

БД, содержащая информацию об обнаруженных ошибках, — тоже мощный инструмент тестирования. Такую базу можно рассматривать и как инструмент управления, и как технический инструмент. Она позволяет узнавать о появлении старых ошибок, следить за частотой обнаружения и исправления новых ошибок, а также за статусом и тяжестью исправленных и неисправленных ошибок. О том, какую информацию об ошибках следует хранить в БД, вы узнаете в разделе 22.7.

## 22.6. Оптимизация процесса тестирования

Способы улучшения тестирования аналогичны способам улучшения любого другого процесса. Опираясь на точные знания о процессе, вы немного изменяете его и смотрите, к каким результатам это приводит. Если результат изменения положителен, значит, процесс стал несколько лучше. В следующих подразделах оптимизация процессов рассматривается в контексте тестирования.

### Планирование тестирования

**Перекрестная ссылка** Одним из элементов планирования тестирования является формализация планов в письменной форме. О документировании тестирования см. работы, указанных в разделе «Дополнительные ресурсы» главы 32.

Эффективность тестирования заметно повышается, если его спланировать в самом начале работы над проектом. Присвоение тестированию той же степени важности, что и проектированию с кодированием, подразумевает, что на тестирование будет выделено время, что оно будет считаться не менее важным и что процесс тестирования будет высококачественным. Планирование тестирования нужно и для обеспечения *повторяемости* процесса тестирования. Если процесс нельзя повторить, его нельзя улучшить.

### Повторное (регрессивное) тестирование

Допустим, вы тщательно протестировали систему и не обнаружили ошибок. Предположим далее, что вы изменяете какой-то фрагмент системы и хотите убедиться в том, что она по-прежнему успешно проходит все тесты, т. е. что изменение

не привело к появлению новых дефектов. Тестирование, призванное гарантировать, что программа не сделала шаг назад, или не «регрессировала», называется «регрессивным».

Почти невозможно создать высококачественную программу, если вы не можете систематично проводить ее повторного тестирования после внесения изменений. Если после каждого изменения выполнять другие тесты, вы не сможете с уверенностью сказать, что в программу не были внесены новые дефекты. Так что при регрессивном тестировании нужно каждый раз выполнять одни и те же тесты. По мере развития системы можно добавлять новые тесты, но сохранять при этом и старые.

## Автоматизированное тестирование

Единственный практичный способ управления регрессивным тестированием — его автоматизация. Многократное выполнение одних и тех же тестов, приводящих к тем же результатам, вводит людей в транс. Они утрачивают концентрацию и начинают упускать ошибки, что сводит на нет эффективность регрессивного тестирования. Гуру тестирования Борис Бейзер сообщает, что уровень ошибок, допускаемых при тестировании вручную, сравним с уровнем ошибок в самом тестируемом коде. По его оценкам, при тестировании, проводимом вручную, должным образом выполняется лишь около половины всех тестов (Johnson, 1994).

Преимущества автоматизированного тестирования таковы.

- При автоматизированном тестировании вероятность допустить ошибку ниже, чем при тестировании вручную.
- Автоматизировав тестирование, вы сможете легко адаптировать его к другим компонентам системы.
- Автоматизация тестирования позволяет выполнять тесты гораздо чаще. Автоматизация тестирования — один из базовых элементов интенсивных методик тестирования, таких как ежедневная сборка программы, дымовое тестирование и экстремальное программирование.
- Автоматизированное тестирование способствует максимально раннему обнаружению проблем, что обычно минимизирует объем работы, нужной для диагностики и исправления проблемы.
- Способствуя более быстрому обнаружению дефектов, внесенных в код при его изменении, автоматизированное тестирование снижает вероятность того, что вам придется позднее заняться крупномасштабным исправлением кода.
- Автоматизированное тестирование особенно полезно в новых изменчивых технологических средах, поскольку оно позволяет быстрее узнавать об изменениях среды.

**Перекрестная ссылка** О связи между зрелостью технологий и методами разработки см. раздел 4.3.

Многие инструменты, используемые для автоматизации тестирования, позволяют создавать тестовые леса, генерировать входные и регистрировать выходные данные и сравнивать действительные выходные данные с ожидаемыми. Для выполнения этих функций можно использовать инструменты, обсуждавшиеся в предыдущем разделе.

## 22.7. Протоколы тестирования

Обеспечить повторяемость процесса тестирования недостаточно — вы должны оценивать и проект, чтобы можно было точно сказать, улучшается он в результате изменений или ухудшается. Вот некоторые категории данных, которые можно собирать с целью оценки проекта:

- административное описание дефекта (дата обнаружения, сотрудник, сообщивший о дефекте, номер сборки программы, дата исправления);
- полное описание проблемы;
- действия, предпринятые для воспроизведения проблемы;
- предложенные способы решения проблемы;
- родственные дефекты;
- тяжесть проблемы (например, критическая проблема, «неприятная» или косметическая);
- источник дефекта: выработка требований, проектирование, кодирование или тестирование;
- вид дефекта кодирования: ошибка занижения или завышения на 1, ошибка присваивания, недопустимый индекс массива, неправильный вызов метода и т. д.;
- классы и методы, измененные при исправлении дефекта;
- число строк, затронутых дефектом;
- время, ушедшее на нахождение дефекта;
- время, ушедшее на исправление дефекта.

Собирая эти данные, вы сможете подсчитывать некоторые показатели, позволяющие сделать вывод об изменении качества проекта:

- число дефектов в каждом классе; все числа целесообразно отсортировать в порядке от худшего класса к лучшему и, возможно, нормализовать по размеру класса;
- число дефектов в каждом методе, все числа целесообразно отсортировать в порядке от худшего метода к лучшему и, возможно, нормализовать по размеру метода;
- среднее время тестирования в расчете на один обнаруженный дефект;
- среднее число обнаруженных дефектов в расчете на один тест;
- среднее время программирования в расчете на один исправленный дефект;
- процент кода, покрытого тестами;
- число дефектов, относящихся к каждой категории тяжести.

### Личные протоколы тестирования

Кроме протоколов тестирования уровня проекта, вы можете хранить и личные протоколы тестирования. Можете включать в них контрольные списки ошибок, которые вы допускаете чаще всего, и указывать время, затрачиваемое вами на написание кода, его тестирование и исправление ошибок.

# Дополнительные ресурсы

http://cc2e.com/2203

Федеральные законы об объективности информации заставляют меня признаться в том, что в нескольких других книгах тестирование рассматривается подробнее, чем в этой главе. Например, в них можно найти материалы о тестировании системы и тестировании методом «черного ящика», которых мы не касались. Кроме того, в этих книгах более глубоко освещаются вопросы тестирования, выполняемого разработчиками. В них обсуждаются формальные подходы к тестированию (такие как создание причинно-следственных диаграмм), а также детали создания независимой организации, занимающейся тестированием.

## Тестирование

Kaner, Cem, Jack Falk, and Hung Q. Nguyen. *Testing Computer Software*, 2d ed. New York, NY: John Wiley & Sons, 1999. Наверное, это лучшая книга по тестированию ПО. Приведенная в ней информация касается в первую очередь тестирования программ, которые будут использоваться большим числом людей (например, крупных Web-сайтов и приложений, продаваемых в магазинах), но полезна и в общем.

Kaner, Cem, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing*. New York, NY: John Wiley & Sons, 2002. Эта книга хорошо дополняет предыдущую. Она разделена на 11 глав, включающих 250 уроков, изученных самими авторами.

Tamre, Louise. *Introducing Software Testing*. Boston, MA: Addison-Wesley, 2002. Эта несложная книга ориентирована на разработчиков, которым нужно понять тестирование. Несмотря на название («Введение в тестирование ПО»), некоторые из приведенных в книге сведений будут полезны и опытным тестировщикам.

Whittaker, James A. *How to Break Software: A Practical Guide to Testing*. Boston, MA: Addison-Wesley, 2002. В этой книге описаны 23 вида атак, которые тестировщики могут использовать для нарушения работы ПО, и приведены примеры каждой атаки с применением популярных программных пакетов. Из-за оригинального подхода эту книгу можно рассматривать и как основной источник информации о тестировании, и как дополнение других книг.

Whittaker, James A. «What Is Software Testing? And Why Is It So Hard?» *IEEE Software*, January 2000, pp. 70–79. В этой статье можно найти хорошее введение в вопросы тестирования ПО и объяснение некоторых проблем, связанных с эффективным тестированием.

Myers, Glenford J. *The Art of Software Testing*. New York, NY: John Wiley, 1979. Эта классическая книга по тестированию ПО издается до сих пор (хотя и стоит немало). Ее содержание довольно простое: тестирование, основанное на самооценке; психология и экономика тестирования программ; анализ и обзоры программ; проектирование тестов; тестирование классов; тестирование более высокого порядка; отладка; инструменты тестирования и другие методики. Книга лаконична (177 страниц) и легко читается. Приведенный в ее начале тест поможет вам начать думать, как тестировщик, и продемонстрирует все разнообразие способов нарушения работы кода.

## Тестовые леса

Bentley, Jon. «A Small Matter of Programming» in *Programming Pearls*, 2d ed. Boston, MA: Addison-Wesley, 2000. В этом эссе приведены хорошие примеры тестовых лесов.

Mackinnon, Tim, Steve Freeman, and Philip Craig. «Endo-Testing: Unit Testing with Mock Objects», *eXtreme Programming and Flexible Processes Software Engineering - XP2000 Conference*, 2000. Первая работа, посвященная использованию поддельных объектов при тестировании, выполняемом разработчиками.

Thomas, Dave and Andy Hunt. «Mock Objects» *IEEE Software*, May/June 2002. Эта статья представляет собой удобочитаемое введение в использование поддельных объектов.

http://cc2e.com/2217

*www.junit.org*. Это сайт поддержки разработчиков, использующих среду JUnit. Аналогичные сайты см. по адресам *cppunit.sourceforge.net* и *nunit.sourceforge.net*.

## Разработка с изначальными тестами

Beck, Kent. *Test-Driven Development: By Example*. Boston, MA: Addison-Wesley, 2003. Бек описывает «разработку через тестирование» — подход, предусматривающий первоначальное создание тестов и последующее написание кода, удовлетворяющего тестам. Хотя местами Бек впадает в евангелический тон, его советы очень полезны, а сама книга лаконична и попадает точно в цель. Стоит отметить, что она включает серьезный пример, для которого приводится реальный код.

## Соответствующие стандарты

*IEEE Std 1008-1987 (R1993)* — стандарт блочного тестирования ПО.

*IEEE Std 829-1998* — стандарт документирования тестов ПО.

*IEEE Std 730-2002* — стандарт планирования контроля качества ПО.

http://cc2e.com/2210

**Контрольный список: тесты**

- ☐ Каждому ли требованию, относящемуся к классу или методу, соответствует отдельный тест?
- ❑ Каждому ли аспекту проектирования, относящемуся к классу или методу, соответствует отдельный тест?
- ❑ Каждая ли строка кода протестирована хотя бы одним тестом? Проверили ли вы это, подсчитав минимальное число тестов, нужное для выполнения каждой строки кода?
- ❑ Каждый ли путь «определение — использование» в потоке данных протестирован хотя бы одним тестом?
- ❑ Проверили ли вы код на наличие в потоке данных путей, которые обычно ошибочны, таких как «определение — определение», «определение — выход» и «определение — уничтожение»?
- ❑ Использовали ли вы список частых ошибок для написания тестов, направленных на обнаружение ошибок, которые часто встречались в прошлом?
- ❑ Протестировали ли вы все простые граничные условия: максимальные, минимальные и граничные условия с завышением или занижением на 1?

- ❑ Протестировали ли вы сложные граничные условия, т. е. комбинации входных данных, которые могут приводить к слишком малому или большому значению вычисляемой переменной?
- ❑ Тестируете ли вы код на предмет неверных видов данных, таких как отрицательное число сотрудников в программе расчета заработной платы?
- ❑ Тестируете ли вы код с использованием типичных средних значений?
- ❑ Тестируете ли вы минимальные нормальные конфигурации?
- ❑ Тестируете ли вы максимальные нормальные конфигурации?
- ❑ Тестируете ли вы совместимость со старыми данными? Тестируете ли вы код, работающий со старым оборудованием и старыми версиями операционной системы, а также интерфейсы со старыми версиями других программ?
- ❑ Позволяют ли тесты легко проверить результаты вручную?

# Ключевые моменты

- Тестирование, выполняемое разработчиками, — один из важнейших элементов полной стратегии тестирования. Независимое тестирование не менее важно, но оно не является предметом этой книги.
- Написание тестов до написания кода требует примерно того же времени и тех же усилий, что и создание тестов после кода, но сокращает циклы регистрации — отладки — исправления дефектов.
- Даже если учесть, что тестирование имеет массу разновидностей, его все равно следует считать лишь одним из элементов хорошей программы контроля качества. Высококачественные методики разработки, позволяющие свести к минимуму число дефектов в требованиях и проектах, играют не менее, а то и более важную роль. Методики совместной разработки характеризуются не меньшей эффективностью обнаружения ошибок, чем тестирование, к тому же они позволяют находить другие ошибки.
- Опираясь на базисное тестирование, анализ потоков данных, анализ граничных условий, классы плохих и классы хороших данных, вы можете создать много тестов детерминированным образом. Методика угадывания ошибок укажет вам на некоторые дополнительные тесты.
- Обычно ошибки концентрируются в нескольких наиболее дефектных классах и методах. Найдите такой код, перепроектируйте его и перепишите.
- Для тестовых данных обычно характерна более высокая плотность ошибок, чем для тестируемого кода. Так как поиск ошибок в тестовых данных требует времени, не приводя к какому бы то ни было улучшению кода, эти ошибки более досадны, чем ошибки программирования. Избегайте их, разрабатывая тесты столь же тщательно, что и код.
- Автоматизация тестирования полезна вообще и практически необходима в случае регрессивного тестирования.
- Чтобы процесс тестирования был максимально эффективным, сделайте его регулярным, выполняйте его оценку и используйте получаемую информацию для его улучшения.

ГЛАВА 23

# Отладка

http://cc2e.com/2361

**Содержание**



**Связанные темы**

- Качество ПО: глава 20
- Тестирование, выполняемое разработчиками: глава 22
- Рефакторинг: глава 24

> Отлаживать код вдвое сложнее, чем писать. Поэтому, если при написании программы вы используете весь свой интеллект, вы по определению недостаточно умны, чтобы ее отладить.
>
> *Брайан Керниган*
> (*Brian W. Kernighan*)

Отладка — это процесс определения и устранения причин ошибок. Этим она отличается от тестирования, направленного на обнаружение ошибок. В некоторых проектах отладка занимает до 50% общего времени разработки. Многие программисты считают отладку самым трудным аспектом программирования.

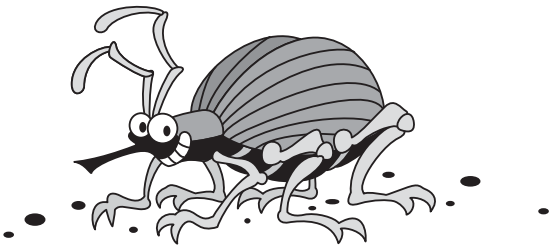
Но отладка не обязана быть таковой. Если вы будете следовать советам, приведенным в этой книге, вам придется отлаживать меньше ошибок. Большинство дефектов будет тривиальными недосмотрами и опечатками, которые вы сможете легко находить, читая исходный код или выполняя его в отладчике. Что до остальных, более сложных ошибок, то, прочитав эту главу, вы сможете сделать их отладку гораздо более легкой.

## 23.1. Общие вопросы отладки

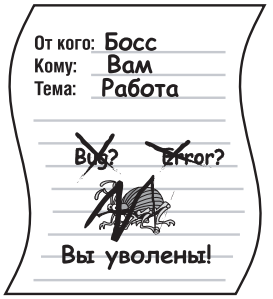
Покойная Грейс Хоппер (Grace Hopper), контр-адмирал ВМС США и один из авторов языка COBOL, утверждала, что слово «bug» появилось в программировании во времена первого крупного цифрового компьютера — Mark I (IEEE, 1992). Разбираясь с причинами неисправности компьютера, программисты нашли в нем крупного мотылька, замкнувшего какую-то цепь, и с тех пор вину за все компью-

терные проблемы стали сваливать на насекомых. Вне программирования слово «bug» уходит корнями по крайней мере во времена Томаса Эдисона, который использовал его уже в 1878 году (Tenner, 1997).

Забавное слово «bug» вызывает в воображении подобные образы:



Однако на самом деле программные дефекты — не организмы, проникающие в код, если его забыли обработать пестицидами. Это ошибки, допущенные программистами, а результаты ошибок обычно больше похожи не на предыдущий рисунок, а на подобную записку:




## Отладка и качество ПО

Как и тестирование, сама по себе отладка не служит для повышения качества ПО — это способ диагностики дефектов. О качестве нужно заботиться с самого начала работы над программой. Лучший способ создания качественной программы заключается в тщательной выработке требований, грамотном проектировании и применении высококачественных методик кодирования. Отладка — это средство для крайних случаев

## Различия в эффективности отладки

Зачем вообще обсуждать отладку? Разве не все программисты умеют отлаживать программы?

Увы, не все. Исследования показали, что опытным программистам для нахождения тех же ошибок требовалось примерно в 20 раз меньше времени, чем неопытным. Более того, некоторые программисты находят больше дефектов и исправляют их грамотнее. Вот результаты классического исследования, в котором профессиональных программистов, обладающих минимум 4-летним опытом, попросили отладить программу с 12 дефектами:

| | Три самых быстрых программиста | Три самых медленных программиста |
|---|---|---|
| Среднее время отладки (в минутах) | 5,0 | 14,1 |
| Среднее число необнаруженных дефектов | 0,7 | 1,7 |
| Среднее число новых дефектов, внесенных в код при исправлении имеющихся дефектов | 3,0 | 7,7 |

**Источник:** «Some Psychological Evidence on How People Debug Computer Programs» (Gould, 1975)

Три самых лучших в отладке программиста смогли найти дефекты примерно в 3 раза быстрее и внесли в код примерно в 2,5 раза меньше дефектов, чем три худших. Самый лучший программист обнаружил все дефекты и не внес во время их исправления новых. Самый худший не нашел 4 из 12 дефектов, а при исправлении 8 обнаруженных дефектов внес в код 11 новых.

Но ставить точку рано. После первого раунда отладки в коде самых быстрых программистов все еще остались 3,7 дефекта, а в коде самых медленных — 9,4 дефекта. Ни одна из групп не довела отладку до конца. У меня возник вопрос, что случится, если экстраполировать данные этого исследования на дополнительные циклы отладки. Мои результаты не являются статистически надежными, но они все же интересны. Применив те же показатели обнаружения и некорректного исправления дефектов к дополнительным циклам, я выяснил, что для снижения показателя необнаруженных дефектов до уровня 0,5 дефекта самой быстрой группе понадобились бы в общей сложности 3 цикла отладки, а самой медленной — 14. Если далее учесть, что время выполнения одного цикла отладки этими группами различается почти в 3 раза, мы получим, что самой медленной группе для полной отладки своих программ потребовалось бы в 13 раз больше времени, чем самой быстрой. Аналогичные крупные различия в эффективности отладки показали и другие исследования (Gilb, 1977; Curtis, 1981).

**Перекрестная ссылка** О связи между качеством ПО и стоимостью его разработки см. раздел 20.5.

Эти данные не только сообщают нам кое-что об отладке, но и подтверждают Главный Закон Контроля Качества ПО: повышение качества программы снижает затраты на ее разработку. Лучшие программисты обнаружили больше дефектов за меньшее время и исправили их более корректно. Нет нужды выбирать между качеством, стоимостью и быстротой — они идут рука об руку.

## Дефекты как возможности обучения

О чем говорит наличие дефекта? Ну, если исходить из того, что наличие дефектов в программе нежелательно, это говорит о том, что вы не полностью понимаете программу. Это тревожит. Все-таки, если вы написали программу, она должна делать то, что вам нужно. Если вы не до конца понимаете свои указания компилятору, еще чуть-чуть, и вы просто начнете пробовать все, что придет в голову, пока что-то не сработает, т. е. начнете программировать методом проб и ошибок. А раз так, дефекты неизбежны. Вам не нужно учиться исправлять дефекты — вам нужно знать, как их избегать.

Однако люди несовершенны, и даже прекрасные программисты иногда допускают промахи. В таких случаях ошибки предоставляют отличные возможности узнать много нового. Например, вы можете сделать все, что описано ниже.

***Изучить программу, над которой работаете*** Наличие дефекта указывает на то, что вы должны лучше изучить программу, потому что, если б вы уже отлично ее знали, в ней не было бы дефектов. Вы уже исправили бы их.

***Изучить собственные ошибки*** Все дефекты вашей программы лежат на вашей совести. Не каждый день у вас появляется такая прекрасная возможность лучше узнать собственные недостатки, и грех ее не использовать. Обнаружив ошибку, спросите себя, как и почему вы допустили ее. Как вы могли найти ее быстрее? Как ее можно было предотвратить? Содержит ли код другие подобные ошибки? Можно ли исправить их до того, как они приведут к проблемам?

**Дополнительные сведения** Методики, помогающие узнать, какие ошибки чаще всего допускаете лично вы, рассматриваются в книге «A Discipline for Software Engineering» (Humphrey, 1995).

***Изучить качество своего кода с точки зрения кого-то, кому придется читать его*** Чтобы найти дефект, вы должны прочитать свой код. Это дает вам прекрасную возможность критически оценить его качество. Легко ли его читать? Как его можно улучшить? Используйте полученные знания для рефакторинга имеющегося кода или для улучшения кода, который вы будете писать позднее.

***Изучить используемые способы решения проблем*** Чувствуете ли вы уверенность, обдумывая свой подход к отладке? Работает ли он? Быстро ли вы находите дефекты? Может, ваш подход к отладке неэффективен? Чувствуете ли вы тоску и огорчение? Не отлаживаете ли вы программу, опираясь на случайные предположения? Нужно ли как-то улучшить процесс отладки? Если учесть объем времени, уходящего на отладку во многих проектах, анализ вашего способа отладки определенно не будет пустой тратой времени. Трата некоторого времени на анализ и изменение процесса отладки может оказаться самым эффективным способом сокращения общего времени разработки программы.

***Изучить используемые способы исправления дефектов*** Кроме способа поиска дефектов, вы можете изучить и свой способ их исправления. Вносите ли вы максимально простые исправления, используя операторы *goto* и частные случаи, устраняющие симптомы, но не проблему? Или вы вносите системные исправления, точно определяя и устраняя причины проблем?

Вышесказанное позволяет сделать вывод, что отладка обеспечивает программистам крайне благоприятные возможности для самосовершенствования. Именно отладка является перекрестком всех дорог конструирования: удобочитаемости, качества проекта и кода и т. д. Именно на этапе отладки окупается создание хорошего кода, особенно если его качество редко заставляет прибегать к отладке.

## Неэффективный подход

К сожалению, в вузах почти никогда не рассматриваются принципы отладки. Возможно, вы прослушали пару специальных лекций, но скорее всего на этом все и кончилось. Я никак не могу пожаловаться на качество полученного образования, но и в моем случае рассмотрение отладки ограничилось советом «для нахождения дефектов включайте в код команды печати». Это неадекватный уровень. Если

образовательный опыт других программистов похож на мой, им приходится изобретать концепции отладки заново. Какая трата времени и сил!

### Руководство Дьявола по отладке

> Программисты не всегда используют доступные данные для ограничения своих рассуждений. Они вносят в код небольшие и иррациональные исправления и часто не отменяют *некорректные* исправления.
>
> *Айрис Весси* (*Iris Vessey*)

В свое время Данте отвел нижний круг ада самому Сатане. Но с тех пор кое-что изменилось, и сейчас Сатана охотно делит нижний круг с программистами, не умеющими эффективно отлаживать программы. Он мучает программистов, заставляя их отлаживать программы с использованием популярных принципов из следующего списка.

***Поиск дефектов, основанный на гадании*** Для нахождения дефекта разбросайте по программе случайным образом команды печати и изучите выходные данные. Если при помощи команд печати найти дефект не получается, попробуйте изменить тот или иной фрагмент — может, что и сработает. Не сохраняйте первоначальный вариант программы и не регистрируйте внесенные изменения. Если вы точно не знаете, что делает программа, программирование становится более увлекательным. Запаситесь колой и леденцами, потому что вам придется провести перед монитором всю ночь.

***Тщательный анализ проблемы — пустая трата времени*** Скорее всего проблема тривиальна, и ее не нужно полностью понимать, чтобы исправить. Достаточно ее просто найти.

***Исправление ошибок самым очевидным образом*** Обычно можно просто устранить специфические симптомы проблемы, а не тратить время на внесение крупного амбициозного исправления, которое может затронуть всю программу. Вот прекрасный пример:

```
x = Compute( y )
if ( y = 17 )
   x = $25.15       - если y = 17, метод Compute() не работает; мы это исправляем
```

Зачем анализировать метод *Compute()* в поисках причин непонятной проблемы со значением *17*, если можно просто написать частный случай в очевидном месте?

### Суеверная отладка

Сатана выделил часть ада программистам, которые отлаживают программы, опираясь на суеверия. В каждой группе есть хоть один программист, бесконечно сражающийся с демоническими компьютерами, таинственными дефектами компилятора, скрытыми дефектами языка, которые проявляются только в полнолуние, плохими данными, утратой важных изменений, заколдованным текстовым редактором, который неправильно сохраняет программы... дополните этот список сами. Это и есть «суеверное программирование».

Если написанная вами программа не работает, ошибка лежит на вашей совести. Компьютер и компилятор ни в чем не виноваты. Программа не делает каждый раз что-то иное. Она не писала сама себя — ее написали вы, так что не уходите от ответственности.

Даже если ошибка поначалу кажется не вашей, в ваших интересах полагать, что справедливо обратное. Это предположение помогает в отладке. Ожидаемый дефект обнаружить в коде трудно; если вы полагаете, что ошибок в вашем коде нет, найти их еще труднее. Допуская вероятность того, что ошибка является вашей, вы будете вызывать большее доверие. Если вы утверждаете, что ошибся кто-то другой, ваши коллеги подумают, что вы тщательно проверили свой код. Предполагая, что виновником проблемы являетесь вы сами, вы сможете избежать неловких ситуаций, связанных с публичным признанием вины, которую вы первоначально отвергали.

# 23.2. Поиск дефекта

Отладка включает поиск дефекта и его исправление. Поиск дефекта и его понимание обычно составляют 90% работы.

К счастью, существуют более эффективный подход к отладке, чем гадание, и, чтобы его применить, совсем не нужно заключать договор с Дьяволом. Отладка программы, основанная на размышлении о проблеме, гораздо эффективнее и интереснее, чем отладка с использованием глаз тритона и помета летучей мыши.

Допустим, вам нужно раскрыть тайну убийства. Что оказалось бы интереснее: обойти дома всех жителей района, проверяя их алиби, или найти несколько улик и, опираясь на них, установить убийцу? Большинство людей предпочло бы второй вариант, поэтому вполне логично, что и у программистов большей популярностью пользуется интеллектуальный подход к отладке. Лучшие программисты, отлаживающие программы в двадцать раз быстрее, чем их менее квалифицированные коллеги, не гадают, как исправить дефекты. Они используют научный метод — основательный процесс анализа и доказательства.

## Научный метод отладки

Классический научный метод включает следующие этапы.

1. Сбор данных при помощи повторяющихся экспериментов.
2. Формулирование гипотезы, объясняющей релевантные данные.
3. Разработка эксперимента, призванного подтвердить или опровергнуть гипотезу.
4. Подтверждение или опровержение гипотезы.
5. Повторение процесса в случае надобности.

Научный метод имеет много аналогов, используемых при отладке. Ниже описан эффективный метод поиска дефекта.

1. Стабилизация ошибки.
2. Определение источника ошибки.
   a. Сбор данных, приводящих к дефекту.
   b. Анализ собранных данных и формулирование гипотезы, объясняющей дефект.
   c. Определение способа подтверждения или опровержения гипотезы, основанного или на тестировании программы, или на изучении кода.

   d. Подтверждение или опровержение гипотезы при помощи процедуры, определенной в п. 2(c).
3. Исправление дефекта.
4. Тестирование исправления.
5. Поиск похожих ошибок.

Первый этап этого процесса аналогичен первому этапу научного метода в том смысле, что оба они основаны на повторяемости. Диагностика дефекта облегчается, если его стабилизировать, т. е. обеспечить его надежное возникновение. Второй этап основан на этапах научного метода. Вы собираете тестовые данные, приводящие к проявлению дефекта, анализируете эти данные и формулируете гипотезу об источнике ошибки. Затем вы проектируете тест или инспекцию и оцениваете гипотезу, после чего празднуете успех (если гипотеза подтверждается) или начинаете поиск источника ошибки сначала. Подтвердив гипотезу, вы исправляете дефект, тестируете исправление и ищете в коде похожие ошибки.

Рассмотрим все эти этапы на примере. Допустим, вы столкнулись с несистематической ошибкой в программе, которая работает с базой данных о сотрудниках. Программа должна печатать список сотрудников в алфавитном порядке и суммы подоходного налога, вычитаемые из зарплаты каждого сотрудника, но при запуске программы выводится:

```
Formatting, Fred Freeform      $5,877
Global, Gary                   $1,666
Modula, Mildred               $10,788
Many-Loop, Mavis               $8,889
Statement, Sue Switch          $4,000
Whileloop, Wendy               $7,860
```

Как видите, список содержит ошибку: записи сотрудников *Many-Loop, Mavis* и *Modula, Mildred* выведены в неверном порядке.

### Стабилизация ошибки

Если дефект проявляется не всегда, его почти невозможно диагностировать. Перевод несистематического дефекта в разряд предсказуемых — один из самых сложных аспектов отладки.

**Перекрестная ссылка** О безопасном использовании указателей см. раздел 13.2.

Непредсказуемые ошибки обычно связаны с инициализацией, расчетом времени или зависшими указателями. Если сумма иногда вычисляется правильно, а иногда неправильно, это, вероятно, объясняется тем, что одна из переменных не инициализируется и просто получает в большинстве случаев нулевое значение. Если вы столкнулись со странной непредсказуемой проблемой при работе с указателями, почти наверняка ее причина — неинициализированный указатель или использование указателя после освобождения соответствующей ему области памяти.

Стабилизация ошибки обычно не ограничивается нахождением теста, приводящего к ошибке. Она предполагает упрощение теста до тех пор, пока не будет получен самый простой тест, все еще приводящий к ошибке. Иначе говоря, тест следует сделать таким простым, чтобы изменение любого его аспекта изменяло

поведение ошибки. Затем, тщательно изменяя тест и наблюдая за поведением программы в контролируемых условиях, вы можете диагностировать проблему. Если в вашей организации есть независимая группа тестирования, упрощение тестов иногда можно поручить ей, но в большинстве случаев это ваша работа.

Для упрощения теста также следует использовать научный метод. Допустим, вы определили 10 факторов, комбинация которых приводит к ошибке. Сформулируйте гипотезу о том, какие факторы нерелевантны для возникновения ошибки. Измените эти факторы и выполните тест еще раз. Если ошибка никуда не исчезает, вы можете упростить тест, исключив эти факторы. Затем вы можете попробовать сделать тест еще проще. Если ошибка больше не возникает, значит, вы опровергли конкретную гипотезу и получили дополнительные сведения о проблеме. Возможно, другие изменения все еще будут приводить к ошибке, но теперь вы хотя бы знаете одно конкретное изменение, после внесения которого ошибка исчезает.

В нашем случае при первом запуске программы запись *Many-Loop, Mavis* выводится в списке после *Modula, Mildred*. Однако при втором запуске записи выводятся в правильном порядке:

```
Formatting, Fred Freeform      $5,877
Global, Gary                   $1,666
Many-Loop, Mavis               $8,889
Modula, Mildred               $10,788
Statement, Sue Switch          $4,000
Whileloop, Wendy               $7,860
```

После ввода записи *Fruit-Loop, Frita*, которая появляется в неверной позиции, вы вспоминаете, что запись *Modula, Mildred* также была введена непосредственно перед отображением некорректного списка. Что необычно в этих случаях, так это то, что записи были введены по отдельности. Как правило, данные вводятся сразу для нескольких сотрудников.

Итак, можно предположить, что проблема как-то связана с вводом одной новой записи. Если это так, то при следующем запуске программы запись *Fruit-Loop, Frita* должна занять правильное место. Ну-ка, проверим:

```
Formatting, Fred Freeform      $5,877
Fruit-Loop, Frita              $5,771
Global, Gary                   $1,666
Many-Loop, Mavis               $8,889
Modula, Mildred               $10,788
Statement, Sue Switch          $4,000
Whileloop, Wendy               $7,860
```

Правильная сортировка списка соответствует выдвинутой гипотезе. И все же для ее окончательного подтверждения следует добавить несколько новых записей по одной за раз и посмотреть, выводятся ли они в неверном порядке при первом запуске программы и изменяется ли их порядок при втором.

### Определение источника ошибки

Определение источника ошибки также призывает к использованию научного метода. Так, вы могли бы заподозрить, что дефект является результатом конкретной проблемы, такой как ошибка занижения или завышения на 1. Для проверки этой гипотезы вы могли бы присвоить параметру, который предположительно вызывает проблему, значение, равное граничному условию, а также два значения, отличающихся от граничного условия на 1.

В нашем примере причиной проблемы может быть занижение или завышение на 1 при добавлении одной новой записи, но не двух или более. Однако изучение кода не приводит к нахождению очевидной ошибки занижения или завышения на 1. Тогда вы обращаетесь к плану Б и добавляете в БД одну новую запись, чтобы увидеть, вызовет ли это проблему. Вы добавляете запись *Hardcase, Henry* и предполагаете, что она займет неправильное место. При запуске программы выводится:

```
Formatting, Fred Freeform      $5,877
Fruit-Loop, Frita              $5,771
Global, Gary                   $1,666
Hardcase, Henry                  $493
Many-Loop, Mavis               $8,889
Modula, Mildred               $10,788
Statement, Sue Switch          $4,000
Whileloop, Wendy               $7,860
```

Список отсортирован правильно, а значит, первая гипотеза ошибочна. Добавление одного нового сотрудника не является причиной проблемы. Проблема имеет или более сложную, или совсем другую природу.

При более внимательном изучении выходных данных теста можно заметить, что только записи *Fruit-Loop, Frita* и *Many-Loop, Mavis* включают дефисы. Сразу же после ввода запись *Fruit-Loop* занимала неверное положение, но с записью *Many-Loop* все было в порядке, ведь так? Хотя распечатки результатов самого первого запуска программы у вас нет, вы помните, что вас смутило место записи *Modula, Mildred*, но она располагалась по соседству с *Many-Loop*. Возможно, ошибочным было положение записи *Many-Loop*, а не *Modula*.

Вы выдвигаете новую гипотезу: причина проблемы связана с фамилиями, содержащими дефисы, а не с фамилиями, которые вводятся по одной.

Но как это объясняет тот факт, что проблема возникает только при вводе записи? Вы изучаете код и замечаете, что в программе используются два разных метода сортировки: один вызывается при вводе записи, а второй — при ее сохранении. Более тщательный анализ первого метода показывает, что он и не должен сортировать список полностью. Вставляя новую запись, он лишь приблизительно определяет ее положение для ускорения сортировки, которую выполняет второй метод. Так что проблема в том, что данные печатаются до их сортировки. Фамилии, содержащие дефисы, располагаются в неправильном порядке из-за того, что метод примерной сортировки не обрабатывает подобные тонкости. Теперь вы можете выдвинуть еще более точную гипотезу: фамилии, содержащие знаки пунктуации, правильно сортируются только во время сохранения.

Далее вы подтверждаете эту гипотезу при помощи дополнительных тестов.

## Советы по поиску причин дефектов

После стабилизации ошибки и уточнения вызывающих ее условий поиск ее может быть как тривиальным, так и трудным: это зависит от того, насколько хорошо написан код. Если найти причину дефекта не удается, возможно, это обусловлено низким качеством кода. Каким бы неприятным ни был такой вывод, ничего не поделаешь: это правда. В подобной ситуации вам помогут следующие советы.

***Формулируя гипотезу, используйте все имеющиеся данные*** Выдвигая гипотезу об источнике дефекта, постарайтесь учесть как можно больше данных. В нашем примере вы могли бы обратить внимание на неверное место записи *Fruit-Loop, Frita* и предположить, что неверно сортируются все фамилии, начинающиеся на букву «F». Гипотеза неудачна: она не объясняет неправильное место записи *Modula, Mildred* и правильную сортировку записей при втором запуске программы. Если данные не соответствуют гипотезе, не игнорируйте их — подумайте, почему они ей не соответствуют, и сформулируйте новую гипотезу.

Вторая наша гипотеза, согласно которой причина проблемы связана с фамилиями, содержащими дефисы, а не с вводом фамилий по одной, также первоначально не объясняла правильную сортировку фамилий при повторном выполнении программы. Однако она привела нас к более точной гипотезе, оказавшейся верной. Если первая гипотеза не объясняет все данные, ничего страшного — улучшайте ее, пока не достигнете этой цели.

***Детализируйте тесты, приводящие к ошибке*** Если вы не можете найти источник ошибки, попробуйте уточнить уже имеющиеся тесты. Возможно, изменение какого-нибудь из параметров в более широком диапазоне или концентрация на одном из параметров позволит сдвинуть отладку с мертвой точки.

***Проверяйте код при помощи блочных тестов*** Как правило, в небольших фрагментах кода дефекты искать легче, чем в крупных интегрированных программах. Используйте для изолированного тестирования фрагментов кода блочные тесты.

**Перекрестная ссылка** Среды блочного тестирования упоминаются в подразделе «Встраивайте блочные тесты в среду тестирования» раздела 22.4.

***Используйте разные инструменты*** На рынке имеются многочисленные инструменты, облегчающие отладку: интерактивные отладчики, строгие компиляторы, инструменты проверки памяти, утилиты проверки синтаксиса и т. д. Правильный инструмент способен сделать трудную работу простой. Мне вспоминается случай, когда одна неуловимая ошибка заставляла одну часть программы перезаписывать память другой части. Традиционные методики отладки не помогли в поисках ошибки: программист не мог определить, какой именно фрагмент перезаписывал память. Тогда он установил точку прерывания на конкретный адрес памяти. Когда программа выполнила запись по этому адресу, отладчик прервал ее выполнение, и виновный код был обнаружен.

Это пример проблемы, которую трудно диагностировать аналитически, но довольно легко с помощью правильного инструмента.

***Воспроизведите ошибку несколькими способами*** Иногда полезную информацию можно получить, выполнив тест, похожий на тесты, приводящие к ошиб-

ке, но не идентичный им. Можете рассматривать этот подход как триангулирование дефекта. Воспроизведя дефект несколькими способами, вы точнее определите его источник.

Воспроизведение ошибки разными способами помогает диагностировать причину ошибки (рис. 23-1). Как только вы решили, что причина дефекта ясна, выполните тест, который сам не должен вызывать ошибку, но напоминает тесты, приводящие к ошибке. Если ошибка при этом все же возникает, значит, вы еще не полностью поняли проблему. Причиной ошибки часто становится комбинация факторов, поэтому попытка диагностировать проблему при помощи только одного теста часто не приводит к обнаружению корня проблемы.

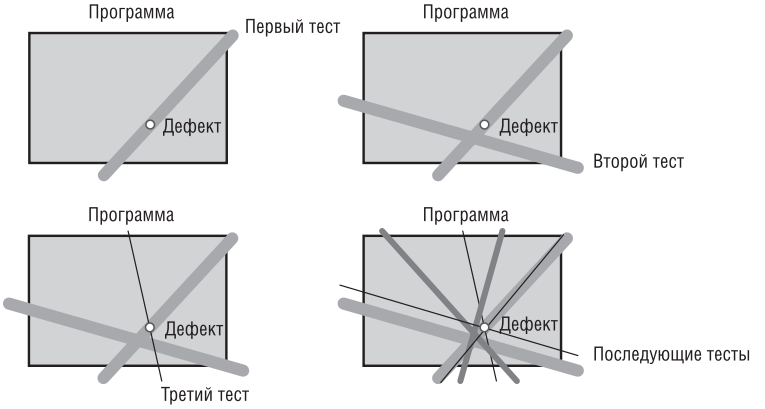



***Рис. 23-1.*** *Чтобы точно определить причину ошибки, попробуйте воспроизвести ошибку разными способами*

***Генерируйте больше данных для формулирования большего числа гипотез*** Выберите тесты, отличающиеся от тестов, результаты которых уже известны. Выполните их, чтобы получить дополнительные данные и использовать их для выдвижения дополнительных гипотез.

***Используйте результаты отрицательных тестов*** Предположим, что вы выдвинули гипотезу и запустили тест с целью ее подтверждения. Допустим далее, что тест опроверг гипотезу, так что причина ошибки все еще неизвестна. И все же вы узнали нечто полезное: одно из ваших предположений о причинах дефекта ошибочно. Это сужает область поиска и уменьшает число оставшихся гипотез.

***Используйте «мозговой штурм» для построения нескольких гипотез*** Не останавливайтесь на первой пришедшей в голову гипотезе, а попробуйте выдвинуть несколько гипотез. Не анализируйте их сразу — просто придумайте за несколько минут максимальное число гипотез. Затем рассмотрите их по очереди и подумайте о тестах, которые могут их доказать или опровергнуть. Это упражнение помогает сдвинуть отладку с мертвой точки, обусловленной слишком сильной концентрацией на одной линии рассуждения.

***Составьте список подходов, которые стоит попробовать*** Иногда программисты не могут найти ошибку по той причине, что слишком долго следовали по пути, ведущему в тупик. Составьте список подходов, которые стоит попробовать, и, если один из них не работает, переходите к следующему.

***Сократите подозрительную область кода*** Вместо тестирования всей программы, всего класса или метода протестируйте сначала меньший фрагмент кода. Используйте для нахождения ошибочного фрагмента команды печати, запись информации в журнал или трассировку.

Есть и более эффективный способ сужения подозрительной области кода: систематически удаляйте части программы и смотрите, возникает ли ошибка. Если ошибка исчезла, ищите ее в удаленной части. Если ошибка по-прежнему возникает, дефектный код все еще присутствует в программе.

Вместо удаления случайных фрагментов руководствуйтесь принципом «разделяй и властвуй». Используйте алгоритм двоичного поиска. Попробуйте удалить в первый раз примерно половину кода. Определите половину, содержащую дефект, и разделите ее. Снова определите дефектную половину и снова разделите ее пополам. Продолжайте, пока дефект не будет найден.

Если программа содержит много небольших методов, можете убирать фрагменты кода, просто комментируя вызовы методов. В противном случае можете блокировать фрагменты кода при помощи комментариев или директив препроцессора.

Работая с отладчиком, удалять фрагменты кода не обязательно. Вместо этого можно задать точки прерывания. Если отладчик позволяет пропускать вызовы методов, попробуйте найти дефектный код, пропуская выполнение определенных методов и наблюдая, исчезает ли после этого ошибка. Этот процесс во многом похож на действительное удаление фрагментов программы.

***С подозрением относитесь к классам и методам, которые содержали дефекты ранее*** Классы, которые были дефектными раньше, более подвержены ошибкам. Новые дефекты чаще обнаруживаются в классах, с которыми и раньше были связаны проблемы, а не в классах, которые были безошибочными. Проанализируйте подверженные ошибкам классы и методы еще раз.

**Перекрестная ссылка** О коде, подверженном ошибкам, см. также подраздел «Выполняйте рефакторинг модулей, подверженных ошибкам» раздела 24.5.

***Проверьте код, который был изменен недавно*** Если у вас появилась новая непростая ошибка, она скорее всего содержится в фрагменте, который был изменен недавно. Ее источником может быть как абсолютно новый код, так и измененный старый. Если вы не можете найти дефект, запустите старую версию программы и проверьте, возникает ли ошибка. Если нет, ошибка содержится в новом коде или объясняется взаимодействием с новым кодом. Изучите различия между старой и новой версиями. Посмотрите в журнале системы управления версиями, какой код был изменен недавно. Если это невозможно, используйте для сравнения старого работоспособного и нового дефектного кода другой инструмент.

***Расширьте подозрительный фрагмент кода*** Сосредоточиться на небольшом фрагменте кода легко, но это принесет пользу, только если дефект *наверняка* содержится в этом фрагменте. Если дефект не удается найти в конкретной области кода, рассмотрите вероятность того, что его в ней нет. Расширьте подозрительную область кода и проанализируйте ее, применив описанную выше методику двоичного поиска.

**Перекрестная ссылка** Об интеграции см. главу 29.

***Выполняйте интеграцию инкрементно*** Отладка будет легкой, если вы будете добавлять элементы в систему по одному за раз. Если после добавления нового элемента возникла новая ошибка, удалите его и протестируйте отдельно.

***Проверяйте наличие распространенных дефектов*** Размышляя о возможных дефектах, используйте контрольные списки качества кода. Следуя методикам инспекций (см. раздел 21.3), вы создадите улучшенные контрольные списки проблем, характерных для вашей среды. Вы также можете использовать контрольные списки, приведенные в этой книге. Все они перечислены после содержания книги.

**Перекрестная ссылка** Обращение за помощью к коллегам иногда помогает рассмотреть проблему под другим углом зрения (см. раздел 21.1).

***Обсудите проблему с кем-то другим*** Некоторые программисты называют это «отладочной исповедью». Довольно часто дефект в своем коде можно найти при объяснении его другому человеку. Так, объясняя проблему с неверной сортировкой фамилий, вы могли бы сказать что-нибудь подобное:

> *Привет, Дженифер, у тебя есть свободная минутка? У меня проблема. Вот этот список сотрудников должен быть отсортирован, но некоторые фамилии выводятся в неверном порядке. Однако во второй раз они сортируются правильно. Чтобы узнать, не связана ли проблема с вводом новых фамилий, я добавил несколько новых записей, но все они были выведены правильно. Я знаю, что список должен быть отсортирован уже при первом запуске программы, потому что фамилии сортируются дважды: при вводе и при сохранении... подожди... нет, при вводе они не сортируются. Все верно. При вводе их место в списке определяется лишь приблизительно. Спасибо, Дженифер, ты мне очень помогла.*

Дженифер не сказала ни слова, но вы нашли решение проблемы!

***Отдохните от проблемы*** Иногда чрезмерная концентрация на проблеме мешает думать. Помните, как вы решили сделать перерыв на чашку кофе и нашли решение проблемы на пути к кофейному автомату? Или во время обеда? Или по пути домой? Или принимая душ следующим утром? Если, попробовав все варианты, вы не получили никаких результатов, отдохните. Прогуляйтесь. Поработайте над чем-то другим. Возьмите выходной. Пусть проблемой займется ваше подсознание.

Дополнительная выгода временного отдыха от проблемы состоит в том, что он снижает связанную с отладкой тревогу. Приступ беспокойства — явный признак того, что пора сделать перерыв.

### Отладка методом грубой силы

При отладке этот способ часто игнорируют. Под «методом грубой силы» я понимаю подход, который может оказаться нудным, трудным и длительным, но *непременно* приведет к решению проблемы. Какие именно подходы непременно приведут к решению проблемы? Это зависит от ситуации, но некоторые типичные варианты назвать можно:

- полный обзор проекта и/или кода дефектного фрагмента;
- выбрасывание фрагмента кода и его повторное проектирование/кодирование с нуля;

- выбрасывание всей программы и ее повторное проектирование/кодирование с нуля;
- компиляция кода с полной отладочной информацией;
- компиляция кода на самом строгом уровне диагностики и исправление всех предупреждений компилятора;
- блочное тестирование нового кода в изоляции от других фрагментов;
- создание и выполнение набора автоматизированных тестов;
- пошаговое выполнение крупного цикла в отладчике вплоть до возникновения ошибки;
- включение в код команд печати, вывода информации на экран или других команд регистрации данных об ошибке;
- компиляция кода с использованием другого компилятора;
- компиляция и выполнение программы в другой среде;
- компоновка кода со специальными библиотеками или выполнение кода в средах, генерирующих предупреждения в подозрительных ситуациях;
- полное воспроизведение конфигурации компьютера конечного пользователя;
- интеграция нового кода в систему небольшими фрагментами с полным тестированием каждого фрагмента.

***Установите лимит времени для быстрой и грязной отладки*** Рассматривая тот или иной метод грубой силы, вы вполне можете решить «Я не могу пойти на это — слишком много работы!» Однако слишком большим можно считать только тот объем работы, который требует больше времени, чем то, что я называю «быстрой и грязной отладкой». Мало кому хочется методично анализировать весь код до тех пор, пока дефекту больше негде будет деться, — всегда есть соблазн быстро угадать причину проблемы. Игрок, живущий в каждом из нас, скорее выбрал бы рискованный подход, позволяющий обнаружить дефект за пять минут, а не надежную методику, непременно приводящую к нахождению дефекта за полчаса. Однако это рискованное дело: если пятиминутный подход не работает, вы начинаете упрямиться. Обнаружение дефекта «легким» способом становится делом принципа, и вы впустую тратите часы, а потом дни, недели, месяцы... Как часто вы тратили два часа на отладку кода, на написание которого уходило только 30 минут? Такое распределение времени трудно признать разумным, и вам следовало бы не отлаживать плохой код, а вообще переписать его.

Если вы все же решаете попробовать быстрый способ отладки, ограничьте его применение определенным интервалом времени. По истечении этого срока смиритесь с тем, что дефект не так прост, как вам казалось сначала, и используйте трудный путь. Так вы сможете быстро избавляться от простых дефектов, а исправление сложных будет чуть более долгим.

***Составьте список методик грубой силы*** До начала отладки сложной ошибки спросите себя: «Есть ли какой-нибудь способ, который *непременно* приведет к решению этой проблемы, если при ее отладке я зайду в тупик?» Если вы определите хотя бы один такой способ (пусть даже переписывание дефектного кода!), вероятность того, что вы впустую потратите лишние часы или дни, уменьшится.

## Синтаксические ошибки

Синтаксические ошибки ждет судьба мамонтов и саблезубых тигров. Диагностические модули компиляторов постоянно улучшаются, и времена, когда поиск неправильно расположенной точки с запятой иногда занимал несколько часов, почти ушли. Соблюдение следующих принципов поможет вам ускорить вымирание подобных ошибок.

***Не полагайтесь на номера строк в сообщениях компилятора*** Если компилятор сообщил о загадочной синтаксической ошибке, изучите фрагменты, расположенные прямо перед ошибкой и сразу после нее: возможно, компилятор неправильно понял проблему или просто включает плохой диагностический модуль. Обнаружив истинный дефект, попробуйте определить, почему компилятор указал не на ту команду. Понимание особенностей компилятора поможет находить дефекты в будущем.

***Не доверяйте сообщениям компилятора*** Компиляторы пытаются сообщить вам точную причину ошибки, но они сами нередко ошибаются, и, чтобы понять смысл их сообщений, иногда приходится читать между строк. Так, при целочисленном делении на 0 компилятор UNIX C может вывести сообщение «floating exception» (исключение при выполнении операции над числом с плавающей точкой). Используя Standard Template Library C++, можно получить пару сообщений об ошибке: первое — о действительной ошибке в использовании STL, а второе — «Error message too long for printer to print; message truncated» («Сообщение об ошибке сокращено, так как оно слишком велико для печати»). Наверное, вы и сами можете привести массу примеров неверных сообщений об ошибках.

***Не доверяйте второму сообщению компилятора*** Одни компиляторы находят множественные ошибки лучше, а другие хуже. Обнаружив первую ошибку, некоторые компиляторы приходят в такое возбуждение, что выдают десятки бессмысленных сообщений о других ошибках. Другим компиляторам, более рассудительным, тоже нравится находить ошибки, но они воздерживаются от вывода неверных сообщений. Если ваш компилятор сгенерировал ряд сообщений об ошибках и вы не можете быстро найти причину второго или третьего сообщения, не волнуйтесь — исправьте первую ошибку и перекомпилируйте программу.

***Разделяй и властвуй*** Разделение программы на части особенно эффективно при поиске синтаксических ошибок. Если вы столкнулись с неуловимой синтаксической ошибкой, удалите часть кода и перекомпилируйте программу. Ошибка или исчезнет (следовательно, она содержится в удаленном коде), или снова появится во всей своей красе (в этом случае удалите другую часть кода). Кроме того, вы можете получить другое сообщение об ошибке (это значит, что вы перехитрили компилятор и заставили его сгенерировать более разумное сообщение).

**Перекрестная ссылка** Наличие редакторов с проверкой синтаксиса зависит от зрелости среды программирования (см. раздел 4.3).

***Грамотно ищите неверно размещенные комментарии и кавычки*** Многие текстовые редакторы для программирования автоматически форматируют комментарии, строковые литералы и другие синтаксические элементы. В более примитивных средах неправильное размещение символов комментария или кавычек может запутать компилятор.

Для нахождения лишних символов комментария или кавычек в коде C, C++ или Java вставьте в него последовательность:

```
/*"/**/
```

Этот фрагмент завершит комментарий или строку, что поможет сузить область, в которой скрываются лишние символы.

## 23.3. Устранение дефекта

Сложной частью отладки является поиск дефекта. Устранить его легко. Однако, как часто бывает, из-за этой самой легкости устранение одних дефектов создает благоприятные условия для внесения других. По крайней мере в одном исследовании было обнаружено, что первые варианты исправления дефектов в половине случаев оказывались некорректными (Yourdon, 1986b). Ниже я привел несколько советов по снижению вероятности ошибок этого рода.

***Прежде чем браться за решение проблемы, поймите ее*** «Руководство Дьявола по отладке» не врет: нет более эффективного способа усложнить себе жизнь и ухудшить качество программы, чем исправление дефектов без их настоящего понимания. Приступайте к устранению проблемы, только разобравшись в ней до конца. Триангулируйте источник ошибки с применением двух видов тестов: тех, что должны привести к ошибке, и тех, которые должны выполниться безошибочно. Выполняйте тесты, пока не поймете проблему достаточно хорошо, чтобы правильно предсказывать появление ошибки в каждом случае.

***Не ограничивайтесь пониманием проблемы — поймите программу*** Если вы понимаете контекст проблемы, у вас больше шансов решить ее полностью, а не частично. Исследование, проведенное с использованием короткой программы, показало, что программисты, стремящиеся полностью понять поведение программы, чаще изменяли ее правильно, чем программисты, концентрировавшиеся на локальном поведении и изучавшие программу только по мере надобности (Littman et al., 1986). Так как программа в этом исследовании была небольшой (280 строк), я не могу утверждать, что вам следует пытаться полностью понять программу из 50 000 строк перед исправлением дефекта. Но вы должны понять хотя бы код, расположенный по соседству с дефектом — под «соседством» я понимаю не *несколько*, а *несколько сотен* строк.

***Подтвердите диагноз проблемы*** Перед исправлением дефекта убедитесь, что вы диагностировали проблему правильно. Выполните тесты, которые доказывают вашу гипотезу и опровергают конкурирующие гипотезы. Если вы установили только то, что ошибка может быть результатом одной из нескольких причин, к устранению проблемы приступать рано — исключите сначала другие причины.

***Расслабьтесь*** Один программист собирался в лыжный поход. Программа была почти готова к выпуску, он опаздывал, и ему оставалось исправить только один дефект. Он изменил исходный файл и зарегистрировал его в системе управления версиями. Он не выполнил перекомпиляцию программы и не проверил правильность изменения.

> Никогда не отлаживайте программу стоя.
>
> *Джеральд Вайнберг (Gerald Weinberg)*

А изменение оказалось неверным, что привело начальника программиста в ярость. Как можно изменять код приложения, готового к выпуску, не проверив его? Что может быть хуже? Разве это не верх некомпетентности?

Если такой поступок и не является вершиной некомпетентности, он очень к ней близок, но это нисколько не сказывается на его распространенности. Решение проблемы второпях — один из самых неэффективных в плане времени подходов. Он подталкивает к необоснованным суждениям, неполной диагностике дефектов и внесению неполных исправлений. Принимая желаемое за действительное, вы можете увидеть решение там, где его нет. Давление — часто самовнушенное — склоняет к принятию случайных и непроверенных решений методом проб и ошибок.

А вот другой подход. В самом конце разработки ОС Microsoft Windows 2000 одному из программистов нужно было исправить последний дефект, после чего уже можно было бы создать коммерческую версию. Он изменил код, проверил исправление и протестировал его на своей локальной сборке. Но сразу регистрировать исправление в системе управления версиями он не стал. Вместо этого он пошел играть в баскетбол, сказав: «Я сейчас чувствую слишком большое напряжение и поэтому не могу быть уверен в том, что рассмотрел все, что следовало. Часок отдохну, приведу мысли в порядок, а потом вернусь и зарегистрирую код, как только увижу, что мое исправление на самом деле корректно».

Отдыхайте, пока правильность решения не станет очевидной. Не поддавайтесь соблазну сэкономить время: обычно это приводит к обратному результату. Следуя этим советам, вы всегда будете вносить правильные исправления, и руководителю не придется вызывать вас из лыжного похода.

**Перекрестная ссылка** Общие вопросы, связанные с изменением кода, подробно обсуждаются в главе 24.

***Сохраняйте первоначальный исходный код*** Перед началом исправления дефекта обязательно заархивируйте имеющийся код, чтобы в случае чего к нему можно было вернуться. Имея дело с несколькими изменениями, вы можете забыть, какое из них важно в текущий момент. Сохранение первоначального исходного кода позволит хотя бы сравнить старый и новый файлы и определить измененные фрагменты.

***Устраняйте проблему, а не ее симптомы*** Конечно, симптомы также нужно устранять, но главной целью должно быть устранение причин проблемы. До конца не разобравшись в проблеме, кода не исправить. Вы устраните симптомы и сделаете код еще хуже. Рассмотрим, например, фрагмент:

**Пример кода, требующего исправления (Java)**

```
for ( claimNumber = 0; claimNumber < numClaims[ client ]; claimNumber++ ) {
   sum[ client ] = sum[ client ] + claimAmount[ claimNumber ];
}
```

Предположим, величина *sum* для клиента *45* отличается от правильного значения на 3,45 доллара. Вот неверный способ решения этой проблемы:

**Пример ухудшения кода в результате его «исправления» (Java)**

```
for ( claimNumber = 0; claimNumber < numClaims[ client ]; claimNumber++ ) {
   sum[ client ] = sum[ client ] + claimAmount[ claimNumber ];
}
```

«Исправление».

```
if ( client == 45 ) {
   sum[ 45 ] = sum[ 45 ] + 3.45;
}
```

Теперь допустим, что при нулевом числе исков (number of claims) со стороны клиента *37* вы не получаете *0*. Плохое решение этой проблемы могло бы быть таким:

**Продолжение примера ухудшения кода в результате его «исправления» (Java)**

```
for ( claimNumber = 0; claimNumber < numClaims[ client ]; claimNumber++ ) {
   sum[ client ] = sum[ client ] + claimAmount[ claimNumber ];
}

if ( client == 45 ) {
   sum[ 45 ] = sum[ 45 ] + 3.45;
}
```

Второе «исправление».

```
else if ( ( client == 37 ) && ( numClaims[ client ] == 0 ) ) {
   sum[ 37 ] = 0.0;
}
```

Если уж и это не заставляет вас содрогнуться от ужаса, вы зря читаете мою книгу: ничто в ней вас не тронет. Перечислить все недостатки этого подхода в книге, объемом лишь около 1000 страниц, невозможно, поэтому ниже я указал только три главных.

- В большинстве случаев эти исправления работать не будут. Проблемы в этом примере очень похожи на дефекты инициализации. Дефекты инициализации по определению непредсказуемы, поэтому тот факт, что сумма для клиента 45 отличается сегодня от верного значения на 3,45 доллара, ничего не говорит о том, что будет завтра. Завтра она может отличаться на 10 000,02 доллара, а может быть верной. Таковы ошибки инициализации.
- Такой код трудно сопровождать. Если для исправления ошибок создаются частные случаи, они становятся самой заметной особенностью кода. Значение 3,45 доллара не всегда будет таким, и позднее возникнет другая ошибка. В код придется включить новый частный случай, а частный случай для 3,45 доллара удален не будет. К коду будут прилипать все новые частные случаи. В конце концов эти «прилипалы» станут слишком тяжелыми для кода, и он пойдет на дно, где ему самое место.

- Неразумно использовать компьютер для выполнения чего-то, что лучше сделать вручную. Компьютеры хороши для выполнения предсказуемых систематичных вычислений, а творческая фальсификация данных лучше дается людям. Вместо подделки данных в коде лучше было бы исправить их в распечатке результатов работы программы.

***Изменяйте код только при наличии веских оснований*** С устранением только симптомов проблемы тесно связана методика случайного изменения кода до тех пор, пока он не покажется верным. Типичный ход мыслей при этом таков: «Похоже, этот цикл содержит дефект. Наверное, это ошибка занижения или завышения на 1, так что я просто отниму 1 и посмотрю, что получится. Ерунда какая-то. Ну-ка, а если прибавить 1? Вроде все работает. Думаю, проблема решена».

Какой бы популярной ни была эта методика, она неэффективна. Внесение случайных изменений в код похоже на накачивание шин автомобиля при неисправности двигателя. Так вы ничего не узнаете — только впустую потратите время. Изменяя программу случайным образом, вы по сути говорите: «Не знаю, в чем тут дело. Будем надеяться, что это изменение сработает». Не делайте так. Это вуду-программирование. Чем сильнее вы измените код, не понимая его, тем более сомнительной станет его корректность.

Не вносите в программу изменение, если не уверены в его правильности. Неверные изменения должны вызывать у вас удивление Они должны вызывать сомнения, пересмотр взглядов и самокритику. Они должны быть редкими.

***Вносите в код по одному изменению за раз*** Одиночные изменения и так довольно коварны. При внесении сразу двух изменений дело только осложняется: они могут привести к тонким ошибкам, похожим на первоначальные. В итоге вы попадаете в затруднительное положение: как узнать, имеете ли вы дело со старой ошибкой, только с новой ошибкой, похожей на старую, или сразу с новой и старой? Не осложняйте себе жизнь и вносите изменения только по одному за раз.

**Перекрестная ссылка** Об автоматизированном регрессивном тестировании см. подраздел «Повторное (регрессивное) тестирование» раздела 22.6.

***Проверяйте исправления*** Проверьте программу сами, попросите сделать это кого-то другого или проанализируйте программу вместе. Выполните те же триангуляционные тесты, что и при диагностике проблемы, и проверьте, все ли аспекты проблемы устранены. Если решена только часть проблемы, вы об этом узнаете.

Проверьте всю программу на предмет того, не привели ли сделанные изменения к побочным эффектам. Самый легкий и эффективный способ обнаружения побочных эффектов — автоматизированное регрессивное тестирование программы в среде JUnit, CppUnit или аналогичной.

***Добавляйте в набор тестов блочные тесты, приводящие к проявлению имеющихся дефектов*** Обнаружив ошибку, на которую не смогли указать имеющиеся тесты, добавьте в набор тестов новый тест, позволяющий предотвратить возвращение этой ошибки.

***Поищите похожие дефекты*** Обнаружив один дефект, поищите аналогичные дефекты. Дефекты часто появляются группами, и, обращая внимание на типы своих дефектов, вы сможете исправлять все дефекты конкретного типа. Поиск похожих

дефектов требует глубокого понимания проблемы. Если вы не можете сообразить, как искать похожие дефекты, значит, вы еще не полностью понимаете проблему.

# 23.4. Психологические аспекты отладки

Отладка предъявляет к интеллекту не меньшие требования, чем любые другие этапы разработки ПО. Самолюбие говорит вам, что ваш код не может содержать дефектов, даже если вы уже встречали их в нем. Выдвигая гипотезы, собирая данные, анализируя гипотезы и методично отказываясь от них, вы должны думать строго, стать немыслимым формалистом. Отлаживая собственный код, вы должны быстро переключаться между гибким творческим мышлением, характерным для проектирования, и жестким критичным мышлением, нужным для отладки. Читая код, стремитесь забыть о том, что он вам известен, и увидеть то, что написано на самом деле, а не то, что вы ожидаете увидеть.

**Дополнительные сведения** Прекрасное обсуждение психологических аспектов отладки и многих других областей разработки ПО вы найдете в книге «The Psychology of Computer Programming» (Weinberg, 1998).

## «Психологическая установка» и слепота при отладке

Встречая в программе элемент *Num*, что вы видите? Неправильно написанное слово «Numb»? Или аббревиатуру слова «Number»? Скорее всего второе. Это одно из проявлений «психологической установки»: вы видите то, что ожидаете увидеть. Что написано тут?


Этот казус уже стал классическим: люди часто замечают только один артикль «the». Они видят то, что ожидают увидеть. Ниже описаны другие похожие факты.

- Студенты, изучающие циклы *while*, часто считают, что условие цикла оценивается непрерывно, т. е. ожидают, что цикл завершится сразу, как только условие станет ложным, а не после проверки условия (Curtis et al., 1986). Они думают, что цикл *while* должен соответствовать слову «пока» в естественном языке.

- Программист, который неумышленно использовал и переменную *SYSTSTS*, и переменную *SYSSTSTS*, думал, что имеет дело с одной переменной. Он не обнаружил проблему, пока программа не была выполнена сотни раз и не была написана книга, содержащая ошибочные результаты (Weinberg, 1998).
- Программисты, сталкивающиеся с кодом:

```
if ( x < y )
   swap = x;
   x = y;
   y = swap;
```

иногда воспринимают его как:

```
if ( x < y ) {
   swap = x;
   x = y;
   y = swap;
}
```

Люди ожидают, что новый феномен будет напоминать аналогичные феномены, виденные ими ранее. Они ожидают, что новая управляющая стурктура будет работать так же, как и старые структуры, что оператор *while* языка программирования будет соответствовать слову «пока», а имена переменных будут такими же, какими были раньше. Вы видите то, что ожидаете увидеть, и поэтому не замечаете отличия, такие как неправильное написание слова «структура» в предыдущем предложении.

Какое отношение психологическая установка имеет к отладке? Во-первых, она подчеркивает важность грамотных методик программирования. Разумное форматирование и комментирование, удачные имена переменных, методов и другие элементы стиля программирования помогают структурировать фоновые условия программирования так, чтобы вероятные дефекты четко выделялись на общем фоне.

Во-вторых, психологическая установка влияет на выбор частей программы, изучаемых при обнаружении ошибки. Исследования показали, что программисты, отлаживающие код максимально эффективно, во время отладки мысленно отделяют нерелевантные фрагменты программы (Basili, Selby, and Hutchens, 1986). Это позволяет им сужать область поиска и находить дефекты быстрее. Однако при этом можно по ошибке «отложить в сторону» часть программы, содержащую дефект. В результате вы будете искать дефект в правильном фрагменте кода, игнорируя фрагмент, который на самом деле содержит дефект. Вы пойдете по неверному пути и должны будете вернуться к перекрестку. Некоторые советы из раздела 23.2 помогут преодолеть эту «слепоту при отладке».

## «Психологическая дистанция»

**Перекрестная ссылка** Советы по выбору ясных имен переменных см. в разделе 11.7.

Психологическую дистанцию можно определить как легкость различения двух элементов. Если дать кому-нибудь длинный список слов и сказать, что все они имеют отношение к уткам, человек может с легкостью перепутать слова «Queck» и «Quack», потому что они кажутся похожими. Психологическая дистанция между ними мала. Гораздо труднее перепутать «Tuack» и «Quack», хотя они тоже различаются только одной буквой. Слово «Tuack» похоже на «Quack» меньше, чем «Queck», потому что первая буква слова сильнее бросается в глаза, чем буква в середине.

Вот примеры психологических дистанций между именами переменных (табл. 23-1):

**Табл. 23-1. Примеры психологических дистанций между именами переменных**

| Первая переменная | Вторая переменная | Психологическая дистанция |
|---|---|---|
| stoppt | stcppt | Почти незаметна |
| shiftrn | shiftrm | Почти отсутствует |
| dcount | bcount | Небольшая |
| claims1 | claims2 | Небольшая |
| product | sum | Большая |

Приступая к отладке, будьте готовы к проблемам, обусловленным недостаточной психологической дистанцией между похожими именами переменных и методов. Выбирайте во время конструирования имена, ясно отличающиеся от других имен, и вы забудете об этих проблемах.

# 23.5. Инструменты отладки — очевидные и не очень

Отладку можно значительно упростить, используя доступные инструменты. Инструментов, способных забить осиновый кол в сердце дефекта-вампира, еще не существует, но с каждым годом они становятся все лучше и лучше.

**Перекрестная ссылка** Граница между инструментами тестирования и отладки размыта. Об инструментах тестирования см. раздел 22.5, а об инструментах разработки ПО — главу 30.

## Утилиты сравнения исходного кода

Утилиты сравнения исходного кода (такие как Diff) полезны при исправлении ошибок. Если после внесения нескольких изменений некоторые из них нужно отменить, но вы плохо помните, что именно вы изменяли, утилита сравнения исходного кода освежит вашу память, указав на различия кода. Если вы обнаружили в новой версии кода дефект, которого не было в предыдущей версии, просто сравните соответствующие файлы.

## Предупреждающие сообщения компилятора

Одним из самых простых и эффективных инструментов отладки является сам компилятор.

***Задайте в компиляторе максимально строгий уровень диагностики и устраняйте все ошибки и предупреждения*** Игнорировать сообщения компилятора об ошибках глупо, но отключать вывод предупреждений еще глупее. Детям иногда кажется, что, если они закроют глаза и перестанут вас видеть, вы пропадете. Отключая вывод предупреждений, вы совершаете такую же ошибку: вы лишь перестаете их видеть, но их причины никуда не исчезают.

Исходите из того, что разработчики компилятора знают язык гораздо лучше вас. Если они о чем-то вас предупреждают, рассматривайте это как возможность узнать о языке что-то новое. Стремитесь понять подлинный смысл каждого предупреждения.

***Рассматривайте предупреждения как ошибки*** Некоторые компиляторы позволяют рассматривать предупреждения как ошибки. Одно из достоинств этой

функции в том, что она повышает важность предупреждений. Как перевод часов на пять минут вперед заставляет думать, что сейчас на пять минут больше, чем на самом деле, так и указание компилятору считать предупреждения ошибками заставляет воспринимать их более серьезно. Кроме того, эта функция часто влияет на процесс компиляции программы. При компиляции и компоновке программы предупреждения в отличие от ошибок обычно не предотвращают компоновку. Если вы хотите проверить предупреждения перед компоновкой, прикажите компилятору рассматривать предупреждения как ошибки.

***Стандартизуйте параметры компиляции в масштабе всего проекта*** Разработайте стандарт, требующий, чтобы все участники проекта компилировали код, используя одинаковые параметры. Иначе при интеграции кода, скомпилированного с использованием разных параметров, вы утонете в сообщениях об ошибках и сами узнаете, что такое интеграционный кошмар. Стандарт легко навязать, создав один общий make-файл или сценарий сборки программы.

## Утилиты расширенной проверки синтаксиса и логики

Существуют инструменты, проверяющие код тщательнее, чем компилятор. Так, программисты на C используют утилиту lint, которая старательно ищет неинициализированные переменные (возникающие, например, при написании = = вместо =) и похожие тонкие проблемы.

## Инструменты профилирования выполнения программы

Вам может показаться, что инструменты профилирования не имеют отношения к отладке, но на самом деле несколько минут, потраченных на изучение профиля программы, могут указать на некоторые неожиданные (и скрытые) дефекты.

Например, при работе над одной из программ у меня возникло подозрение, что метод управления памятью снижает быстродействие программы. Модуль управления памятью изначально был небольшим компонентом, использующим линейно упорядоченный массив указателей на области памяти. Я заменил линейно упорядоченный массив на хэш-таблицу, ожидая, что время выполнения кода сократится минимум вдвое. Однако при профилировании кода я не обнаружил изменения быстродействия. Тщательнее изучив код, я обнаружил дефект в алгоритме выделения памяти, приводивший к огромным тратам времени. Узкое место было обусловлено не линейным поиском, а дефектом. Алгоритм поиска вообще можно было не оптимизировать. Исследуйте результаты работы инструмента профилирования, чтобы гарантировать, что каждая часть программы выполняется за разумный интервал времени.

## Среды тестирования и леса

**Перекрестная ссылка** О лесах см. подраздел «Создание лесов для тестирования отдельных классов» раздела 22.5.

Как уже было сказано в разделе 23.2 в отношении поиска дефектов, выделение проблемного фрагмента кода и его тестирование в изоляции от других фрагментов часто оказывается самым эффективным способом изгнания бесов из дефектной программы.

## Отладчики

Коммерческие отладчики непрерывно совершенствуются и могут значительно изменить ваш способ программирования. Хорошие отладчики позволяют прерывать выполнение программы на конкретной строке, при достижении конкретной строки в *n*-й раз, при изменении глобальной переменной или при присвоении переменной конкретного значения. Они позволяют выполнять код строка за строкой с «перешагиванием» через методы или с их «посещением», возвращаться к началу дефектного фрагмента, а также регистрировать выполнение отдельных операторов в журнале, что похоже на разбрасывание по всей программе команд печати «Я здесь!».

Хорошие отладчики позволяют полностью исследовать данные, в том числе структурированные и динамически выделенные. С их помощью легко просмотреть содержание связного списка указателей или динамически выделенного массива. Они поддерживают типы данных, определенные пользователем. Они позволяют выполнить нерегламентированный запрос данных, присвоить им новые значения и продолжить выполнение программы.

Вы можете работать с высокоуровневым языком или ассемблерным кодом, сгенерированным компилятором. Если вы используете несколько языков, отладчик автоматически выводит любой фрагмент кода на соответствующем языке. Вы можете изучить цепочку вызовов методов и быстро увидеть исходный код любого метода. В среде отладчика вы можете изменять параметры программы.

Лучшие отладчики запоминают параметры отладки (точки прерывания, отслеживаемые переменные и т. д.) каждой отдельной программы, чтобы их не нужно было задавать заново.

Системные отладчики работают на системном, а не прикладном уровне, не влияя на выполнение отлаживаемой программы. Это важно, если вы отлаживаете программу, чувствительную ко времени выполнения или объему доступной памяти.

Если учесть все достоинства отладчиков, может показаться странным, что кто-то их критикует. И все же некоторые из самых уважаемых людей в мире компьютерных наук рекомендуют их не использовать. Они советуют полагаться на свой ум и избегать инструментов отладки вообще. Они утверждают, что инструменты отладки — это костыли и что вы решите проблемы быстрее и лучше, размышляя о них, а не опираясь на инструменты. Они утверждают, что при поиске дефектов программу нужно выполнять в уме, а не в отладчике.

> Интерактивный отладчик — великолепный пример инструмента, который не нужен: он поощряет хакерство методом проб и ошибок, а не систематичное проектирование и позволяет непрофессионалам скрыть свою некомпетентность.
>
> *Харлан Миллз*
> (*Harlan Mills*)

Какими бы ни были эмпирические данные, суть нападок на отладчики некорректна. Если инструмент допускает неграмотное применение, это не значит, что от него надо отказаться. Было бы глупо запретить аспирин из-за возможности передозировки. Риск пораниться не заставил бы вас прекратить подстригать лужайку перед домом. Любой эффективный инструмент можно использовать правильно и неправильно. Отладчик — не исключение.

Отладчик не заменит грамотного рассуждения. Но иногда никакие мысли не заменят хороший отладчик. Наиболее эффективная комбинация — ясный ум и хороший отладчик.

http://cc2e.com/2368

### Контрольный список: отладка

**Методики поиска дефектов**

- ❑ Формулируя гипотезу, используйте все имеющиеся данные.
- ❑ Детализируйте тесты, приводящие к ошибке.
- ❑ Проверяйте код при помощи блочных тестов.
- ❑ Используйте разные доступные инструменты.
- ❑ Воспроизведите ошибку несколькими разными способами.
- ❑ Генерируйте больше данных для формулирования большего числа гипотез.
- ❑ Используйте результаты отрицательных тестов.
- ❑ Используйте «мозговой штурм» для построения нескольких гипотез.
- ❑ Составьте список подходов, которые следует попробовать.
- ❑ Сократите подозрительную область кода.
- ❑ С подозрением относитесь к классам и методам, которые содержали дефекты ранее.
- ❑ Проверьте код, который был изменен недавно.
- ❑ Расширьте подозрительный фрагмент кода.
- ❑ Выполняйте интеграцию инкрементно.
- ❑ Проверяйте наличие распространенных дефектов.
- ❑ Обсудите проблему с кем-то другим.
- ❑ Отдохните от проблемы.
- ❑ Установите сроки быстрой и грязной отладки.
- ❑ Составьте список методик грубой силы и используйте их.

**Методики поиска и исправления синтаксических ошибок**

- ❑ Не полагайтесь на номера строк в сообщениях компилятора.
- ❑ Не доверяйте сообщениям компилятора.
- ❑ Не доверяйте второму сообщению компилятора.
- ❑ Разделяй и властвуй.
- ❑ Используйте редактор с проверкой синтаксиса для поиска неверно размещенных символов комментария и кавычек.

**Методики устранения дефектов**

- ❑ Прежде чем браться за решение проблемы, поймите ее.
- ❑ Не ограничивайтесь пониманием проблемы — поймите программу.
- ❑ Подтвердите диагноз проблемы.
- ❑ Расслабьтесь.
- ❑ Сохраняйте первоначальный исходный код.
- ❑ Устраняйте проблему, а не ее симптомы.
- ❑ Изменяйте код только при наличии веских оснований.
- ❑ Вносите в код по одному изменению за раз.
- ❑ Проверяйте исправления.
- ❑ Добавляйте в набор тестов блочные тесты, приводящие к проявлению имеющихся дефектов.
- ❑ Поищите похожие дефекты.

**Общий подход к отладке**

- ❑ Рассматриваете ли вы отладку как возможность лучше изучить программу, ошибки, качество кода и подход к решению проблем?
- ❑ Избегаете ли вы суеверного подхода к отладке, основанного на методе проб и ошибок?
- ❑ Полагаете ли вы, что ошибки допущены именно вами?
- ❑ Используете ли вы научный метод для стабилизации несистематических ошибок?
- ❑ Используете ли вы научный метод для нахождения дефектов?
- ❑ Используете ли вы несколько методик поиска дефектов?
- ❑ Проверяете ли вы корректность исправлений?
- ❑ Анализируете ли вы предупреждения компилятора? Используете ли вы инструменты профилирования выполнения программы, среды тестирования, леса и интерактивные отладчики?

# Дополнительные ресурсы

Ниже я привел список книг, посвященных отладке.

http://cc2e.com/2375

Agans, David J. *Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems*. Amacom, 2003. В этой книге рассматриваются общие принципы отладки, не зависящие ни от языка, ни от среды.

Myers, Glenford J. *The Art of Software Testing*. New York, NY: John Wiley & Sons, 1979. Седьмая глава этой классической книги посвящена отладке.

Allen, Eric. *Bug Patterns In Java*. Berkeley, CA: Apress, 2002. В данной книге описывается методика отладки программ Java, концептуально очень похожая на описанную в этой главе. Как и здесь, в ней рассмотрен «Научный метод отладки», проведено различие между отладкой и тестированием и определены частые ошибки.

Названия двух следующих книг могут навести на мысль о том, что они адресованы только разработчикам программ для Microsoft Windows и .NET, однако это не так: помимо всего прочего, в них вы найдете обсуждение отладки в целом, советы по использованию утверждений, а также описания методик кодирования, помогающих предотвращать ошибки.

Robbins, John. *Debugging Applications for Microsoft .NET and Microsoft Windows*. Redmond, WA: Microsoft Press, 2003[1].

McKay, Everett N. and Mike Woodring. *Debugging Windows Programs: Strategies, Tools, and Techniques for Visual C++ Programmers*. Boston, MA: Addison-Wesley, 2000.

[1] Роббинс Дж. Отладка приложений для Microsoft .NET и Microsoft Windows. — М.: Русская Редакция, 2004. — *Прим. перев.*

## Ключевые моменты

- Отладка — это тот этап разработки программы, от которого зависит возможность ее выпуска. Конечно, лучше всего вообще избегать ошибок, используя другие методики, описанные в этой книге. Однако потратить время на улучшение навыков отладки все же стоит, потому что эффективность отладки, выполняемой лучшими и худшими программистами, различается минимум в 10 раз.
- Систематичный подход к поиску и исправлению ошибок — непременное условие успешности отладки. Организуйте отладку так, чтобы каждый тест приближал вас к цели. Используйте Научный Метод Отладки.
- Прежде чем приступать к исправлению программы, поймите суть проблемы. Случайные предположения о причинах ошибок и случайные исправления только ухудшат программу.
- Установите в настройках компилятора самый строгий уровень диагностики и устраняйте причины всех ошибок и предупреждений. Как вы исправите неуловимые ошибки, если будете игнорировать явные?
- Инструменты отладки значительно облегчают разработку ПО. Найдите их и используйте, но не забывайте, что у вас есть еще и голова.

ГЛАВА 24

# Рефакторинг

### Содержание



http://cc2e.com/2436

### Связанные темы

- Советы по устранению дефектов: раздел 23.3
- Подход к оптимизации кода: раздел 25.6
- Проектирование при конструировании: глава 5
- Классы: глава 6
- Высококачественные методы: глава 7
- Совместное конструирование: глава 21
- Тестирование, выполняемое разработчиками: глава 22
- Области вероятных изменений: подраздел «Определите области вероятных изменений» раздела 5.3

Все удачные программы изменяются.

*Фред Брукс* (*Fred Brooks*)

Миф: в начале реализации программного проекта проводится методичная выработка требований, и составляется устойчивый список аспектов ответственности программы. Проектирование соответствует требованиям и выполняется со всей тщательностью, благодаря чему кодирование от начала до конца протекает линейно: вы пишете код, тестируете его и оставляете в покое. Согласно этому мифу значительные изменения кода возможны только при сопровождении ПО, т. е. после выпуска первоначальной версии системы.

Реальность: во время разработки первоначальной версии системы в код вносятся значительные изменения. Многие из изменений во время кодирования не менее масштабны, чем изменения, характерные для стадии сопровождения программы. В зависимости от размера проекта кодирование, отладка и блочное тестирование обычно составляют от 30 до 65% общего объема

работы над проектом (см. главу 27). Если бы кодирование и блочное тестирование были линейными однократными процессами, они составляли бы не более 20–30% общего объема работы. Однако даже в хорошо управляемых проектах требования изменяются примерно на 1–4% в месяц (Jones, 2000). Изменения требований неизбежно приводят к изменениям кода, порой весьма существенным.

Другая реальность: современные методики разработки предполагают больший масштаб изменений кода во время конструирования. Целью более старых методик (достигалась она или нет — другой вопрос) было предотвращение изменений кода. Современные подходы снижают предсказуемость кодирования. Они в большей степени ориентированы на код, поэтому вы вполне можете ожидать, что на протяжении жизненного цикла проекта код будет изменяться сильнее, чем когда бы то ни было.

# 24.1. Виды эволюции ПО

Эволюция ПО похожа на биологическую эволюцию тем, что лишь немногие мутации выгодны. При здоровой эволюции кода его развитие напоминает превращение обезьян в неандертальцев, а неандертальцев — в программистов, которые, как известно, являются самыми высокоразвитыми существами на Земле. Однако иногда эволюционные силы проявляют себя иным образом, ввергая программу в спираль деградации.

Главное различие между видами эволюции программы в том, повышается или снижается ее качество в результате изменений. Если при исправлении ошибок вы опираетесь на суеверия и устраняете лишь симптомы проблем, качество снижается. Если же вы рассматриваете изменения как возможности улучшить первоначальный проект программы, — повышается. При снижении качества программа начинает походить на молчащую канарейку в шахте, о которой я уже говорил. Это знак того, что программа развивается в неверном направлении.

Характер эволюции программы зависит и от того, когда в нее вносятся изменения: во время конструирования или во время сопровождения. Изменения во время конструирования обычно вносят первоначальные разработчики, которые еще хорошо помнят код программы. Система еще не используется, поэтому программисты испытывают давление только со стороны графика разработки, а не сотен сердитых пользователей, возмущенных неработоспособностью системы. По той же причине изменения во время конструирования можно вносить более свободно: система находится в более динамичном состоянии, а следствия ошибок менее серьезны. Из этого следует, что во время разработки ПО эволюционирует не так, как при сопровождении.

## Философия эволюции ПО

Слабость многих подходов к эволюции ПО объясняется тем, что она пускается на самотек. Осознав, что эволюция ПО во время его разработки — неизбежный и важный процесс и спланировав его, вы сможете извлечь из него выгоду.

Эволюция заключает в себе и опасность, и возможность приближения к совершенству. При необходимости изменения кода старайтесь улучшить его, чтобы облегчить внесение изменений в будущем. В процессе написания программы вы всегда узнаете о ней что-то новое. Получив возможность изменения программы, используйте то, что вы узнали, для ее улучшения. Пишите первоначальный код и его изменяйте, держа в уме дальнейшие изменения.

> Не бывает кода, настолько громоздкого, изощренного или сложного, чтобы его нельзя было ухудшить при сопровождении.
>
> *Джеральд Вайнберг* (*Gerald Weinberg*)

Главное Правило Эволюции ПО состоит в том, что эволюция должна повышать внутреннее качество программы. О том, как этого добиться, я расскажу в следующих разделах.

## 24.2. Введение в рефакторинг

Важнейшей стратегией достижения цели Главного Правила Эволюции ПО является рефакторинг, который Мартин Фаулер определяет как «изменение внутренней структуры ПО без изменения его наблюдаемого поведения, призванное облегчить его понимание и удешевить модификацию» (Fowler, 1999). Слово «рефакторинг» возникло из слова «факторинг», которое изначально использовал в контексте структурного программирования Ларри Константайн, назвавший так максимально возможную декомпозицию программы на составляющие части (Yourdon and Constantine, 1979).

### Разумные причины выполнения рефакторинга

Иногда код деградирует при сопровождении, а иногда он изначально имеет невысокое качество. В обоих случаях на это — и на необходимость рефакторинга — указывают некоторые предупреждающие знаки, иногда называемые «запахами» (smells) (Fowler, 1999). Они описаны ниже.

***Код повторяется*** Повторение кода почти всегда говорит о неполной факторизации системы на этапе проектирования. Повторение кода заставляет параллельно изменять сразу несколько фрагментов программы и нарушает правило, которое Эндрю Хант и Дэйв Томас назвали «принципом DRY»: Don't Repeat Yourself (не повторяйтесь) (Hunt and Thomas, 2000). Думаю, лучше всех это правило сформулировал Дэвид Парнас: «Копирование и вставка кода — следствие ошибки проектирования» (McConnell, 1998b).

***Метод слишком велик*** В объектно-ориентированном программировании методы, не умещающиеся на экране монитора, требуются редко и обычно свидетельствуют о попытке насильно втиснуть ногу структурного программирования в объектно-ориентированный ботинок.

Одному из моих клиентов поручили разделить самый объемный метод унаследованной системы, включающий более 12 000 строк. Приложив немалые усилия, он смог уменьшить объем этого метода только примерно до 4000 строк.

Одним из способов улучшения системы является повышение ее модульности — увеличение числа хорошо определенных и удачно названных методов, успешно решающих только одну задачу. Если обстоятельства заставляют вас пересмотреть

фрагмент кода, используйте эту возможность для проверки модульности методов, содержащихся в этом фрагменте. Если вам кажется, что после разделения одного метода на несколько код станет яснее, создайте дополнительные методы.

***Цикл слишком велик или слишком глубоко вложен в другие циклы*** Подходящим кандидатом на преобразование в метод часто оказывается тело цикла — это помогает лучше факторизовать код и снизить сложность цикла.

***Класс имеет плохую связность*** Если класс имеет множество никак не связанных аспектов ответственности, разбейте его на несколько классов, так чтобы каждый из них получил связный набор аспектов.

***Интерфейс класса не формирует согласованную абстракцию*** Даже классы, получившие при рождении связный интерфейс, могут терять первоначальную согласованность. В результате необдуманных изменений, повышающих удобство использования класса за счет целостности его интерфейса, интерфейс иногда становится монстром, не поддающимся сопровождению и не улучшающим интеллектуальную управляемость программы.

***Метод принимает слишком много параметров*** Как правило, хорошо факторизованные программы включают много небольших хорошо определенных методов, не нуждающихся в большом числе параметров. Длинный список параметров — свидетельство того, что абстракция, формируемая интерфейсом метода, неудачна.

***Отдельные части класса изменяются независимо от других частей*** Иногда класс имеет две (или более) разных области ответственности. Если это так, вы заметите, что вы изменяете или одну часть класса, или другую, и лишь немногие изменения затрагивают обе части класса. Это признак того, что класс следует разделить на несколько классов в соответствии с отдельными областями ответственности.

***При изменении программы требуется параллельно изменять несколько классов*** Мне известен один проект, в котором был составлен контрольный список где-то из 15 классов, требующих изменения при добавлении нового вида выходных данных. Если вы уже в который раз изменяете один и тот же набор классов, подумайте, можно ли реорганизовать код этих классов так, чтобы изменения затрагивали только один класс. Опыт говорит мне, что этого идеала достичь нелегко, но стремиться к нему нужно.

***Вам приходится параллельно изменять несколько иерархий наследования*** Если при создании каждого нового подкласса одного класса вам приходится создавать подкласс другого класса, вы имеете дело с особым видом параллельного изменения. Решите эту проблему.

***Вам приходится параллельно изменять несколько блоков* case** В самих по себе блоках *case* ничего плохого, но, если вы параллельно изменяете похожие блоки *case* в нескольких частях программы, спросите себя, не лучше ли использовать наследование.

***Родственные элементы данных, используемые вместе, не организованы в классы*** Если вы неоднократно используете один и тот же набор элементов данных, рассмотрите целесообразность объединения этих данных и выполняемых над ними операций в отдельный класс.

***Метод использует больше элементов другого класса, чем своего собственного*** Это значит, что метод нужно переместить в другой класс и вызывать из старого класса.

***Элементарный тип данных перегружен*** Элементарные типы данных могут представлять бесконечное число сущностей реального мира. Если вы собираетесь представить распространенную сущность — скажем, денежную сумму — целочисленным или другим элементарным типом данных, подумайте: не создать ли вместо этого простой класс *Money*, чтобы компилятор мог выполнять контроль типов объектов *Money*, дабы можно было проверять значения, присваиваемые этим объектам, и т. д. Если и *Money*, и *Temperature* будут представлены целыми числами, компилятор не сможет предупредить вас об ошибочных операциях вида *bankBalance = recordLowTemperature*.

***Класс имеет слишком ограниченную функциональность*** Иногда рефакторинг приводит к сокращению функциональности класса. Если класс не соответствует своему званию, спросите себя, не удалить ли его вообще, распределив все аспекты его ответственности между другими классами.

***По цепи методов передаются бродячие данные*** Данные, передаваемые в метод лишь затем, чтобы он передал их другому методу, называются «бродячими» (tramp data) (Page-Jones, 1988). Это не всегда плохо, но в любом случае спросите себя, согласуется ли передача конкретных данных с абстракцией, формируемой интерфейсом каждого из методов. Если с абстракциями интерфейсов порядок, с передачей данных тоже все в норме. Если нет, найдите способ, позволяющий улучшить согласованность интерфейса каждого метода.

***Объект-посредник ничего не делает*** Если роль класса сводится к перенаправлению вызовов методов в другие классы, подумайте, не устранить ли его и вызывать другие классы непосредственно.

***Один класс слишком много знает о другом классе*** Инкапсуляция (сокрытие информации) — наверное, самый эффективный способ улучшения интеллектуальной управляемости программ и минимизации волновых эффектов изменений кода. Увидев, что один класс знает о другом больше, чем следует (это относится и к производным классам, знающим слишком много о своих предках), постарайтесь сделать инкапсуляцию более строгой.

***Метод имеет неудачное имя*** Если методу присвоено плохое имя, измените определение метода, исправьте все его вызовы и перекомпилируйте программу. Какой бы трудной эта задача ни была сейчас, потом она станет еще труднее — поэтому, обнаружив проблему, устраните ее как можно быстрее.

***Данные-члены сделаны открытыми*** Мне кажется, что предоставление открытого доступа к данным-членам не бывает разумным решением. Это стирает грань между интерфейсом и реализацией, неизбежно нарушает инкапсуляцию и ограничивает гибкость программы. Непременно подумайте над сокрытием открытых данных-членов при помощи методов доступа.

***Подкласс использует только малую долю методов своих предков*** Обычно такая ситуация возникает, когда подкласс создается потому, что базовый класс по чистой случайности содержит нужные ему методы, а не потому, что подкласс логически является потомком базового класса. Попробуйте достичь лучшей ин-

капсуляции, изменив характер отношений между подклассом и базовым классом с «является» на «содержит»: преобразуйте базовый класс в данные-члены бывшего подкласса и откройте доступ только к тем методам бывшего подкласса, которые по-настоящему нужны.

***Сложный код объясняется при помощи комментариев*** В важности комментариев трудно усомниться, но их не следует использовать для объяснения плохого кода. Старая мудрость гласит: «Не документируйте плохой код — перепишите его» (Kernighan and Plauger, 1978).

**Перекрестная ссылка** О глобальных переменных см. раздел 13.3. О различиях между глобальными данными и данными класса см. подраздел «Ошибочное представление о данных класса как о глобальных данных» раздела 5.3.

***Код содержит глобальные переменные*** При пересмотре фрагмента, в котором используются глобальные переменные, изучите его получше. Возможно, за время, прошедшее с тех пор, как вы изучали этот фрагмент, вам пришел в голову способ устранения глобальных переменных. За это время вы уже забыли кое-какие аспекты кода — сейчас использование глобальных переменных может показаться вам довольно запутанным, чтобы вы сочли нужным изобретение более ясного подхода. Возможно, сейчас вы лучше представляете, как изолировать глобальные переменные при помощи методов доступа и как пострадает программа, если оставить все как есть. Соберитесь с силами и внесите в код выгодные изменения. Написание первоначального кода ушло в достаточно далекое прошлое, чтобы вы могли объективно отнестись к своей работе, но не настолько далекое, чтобы вы не смогли вспомнить все, что нужно для внесения правильных исправлений. Ранние ревизии кода — прекрасная возможность его улучшить.

***Перед вызовом метода выполняется подготовительный код (после вызова метода выполняется код «уборки»)*** Подобный код должен вызывать у вас подозрение:

**Пример кода подготовки к вызову метода и кода уборки — плохой код (C++)**

Подготовительный код — дурной знак.

```
WithdrawalTransaction  withdrawal;
withdrawal.SetCustomerId( customerId );
withdrawal.SetBalance( balance );
withdrawal.SetWithdrawalAmount( withdrawalAmount );
withdrawal.SetWithdrawalDate( withdrawalDate );

ProcessWithdrawal( withdrawal );
```

Код уборки — еще один дурной знак.

```
customerId = withdrawal.GetCustomerId();
balance = withdrawal.GetBalance();
withdrawalAmount = withdrawal.GetWithdrawalAmount();
withdrawalDate = withdrawal.GetWithdrawalDate();
```

Похожий признак плохого кода — наличие специального конструктора, принимающего подмножество нормальных данных инициализации и нужного для написания чего-нибудь вроде:

**Пример кода подготовки к вызову метода и кода уборки — плохой код (C++)**

```
withdrawal = new WithdrawalTransaction( customerId, balance,
    withdrawalAmount, withdrawalDate );
withdrawal.ProcessWithdrawal();
delete withdrawal;
```

Натолкнувшись на код, подготавливающий программу к вызову метода или восстанавливающий ее боеспособность после вызова, спросите себя, формирует ли интерфейс метода адекватную абстракцию. В последнем примере список параметров метода *ProcessWithdrawal*, возможно, следовало бы изменить так:

**Пример метода, не требующего ни подготовки, ни уборки, — хороший код (C++)**

```
ProcessWithdrawal( customerId, balance, withdrawalAmount, withdrawalDate );
```

Заметьте, что похожая проблема возникает и в обратном случае. Так, если у вас обычно есть объект *WithdrawalTransaction*, но в метод *ProcessWithdrawal* передаются только несколько значений объекта, вам следует подумать о рефакторинге интерфейса метода, чтобы он принимал объект *WithdrawalTransaction*, а не его отдельные поля:

**Пример кода, требующего нескольких вызовов методов (C++)**

```
ProcessWithdrawal( withdrawal.GetCustomerId(), withdrawal.GetBalance(),
    withdrawal.GetWithdrawalAmount(), withdrawal.GetWithdrawalDate() );
```

Каждый из этих подходов может быть как верным, так и неверным: все зависит от того, какую абстракцию формирует интерфейс метода *ProcessWithdrawal()*. Если абстракция подразумевает, что метод ожидает четыре отдельных элемента данных, используйте один подход; если метод ожидает сразу весь объект *WithdrawalTransaction*, выберите другой.

***Программа содержит код, который может когда-нибудь понадобиться***
Программисты очень плохо угадывают, какая функциональность может потребоваться позднее. «Проектирование впрок» связано со многими предсказуемыми проблемами, описанными ниже.

- Требования к коду, «проектируемому впрок», не разрабатываются в полном объеме, поэтому программист скорее всего не угадает эти будущие требования. Код, «написанный в расчете на будущее», в итоге придется выбросить.
- Если догадка программиста по поводу будущих требований близка к истине, он все же не сможет предвосхитить все их тонкости. Эти тонкости опровергнут исходные предпосылки проектирования, и «проект, разработанный в расчете на будущее», также придется выбросить.
- Программисты, использующие код, который в свое время был «спроектирован впрок», не знают об этом или предполагают, что код лучше, чем есть на самом деле. Они думают, что этот код был написан, протестирован и подвергнут обзору с той же тщательностью, что и остальной. Они могут потратить много времени на создание кода, использующего «спроектированное впрок», только для того, чтобы обнаружить в итоге, что «спроектированное впрок» на самом деле не работает.

- Дополнительный код, «проектируемый впрок», создает дополнительную сложность, что приводит к дополнительному тестированию, исправлению дефектов и т. д. Результат — замедление работы над проектом.

Мнение экспертов по этому поводу однозначно: если вы хотите наилучшим образом подготовиться к будущим требованиям, не пишите гипотетически нужный код, а уделите повышенное внимание ясности и понятности кода, который *нужен прямо сейчас*, чтобы будущие программисты знали, что он делает, чего не делает, и могли быстро и правильно изменить его (Fowler, 1999; Beck, 2000).

http://cc2e.com/2443

**Контрольный список: разумные причины выполнения рефакторинга**

- ❑ Код повторяется.
- ❑ Метод слишком велик.
- ❑ Цикл слишком велик или слишком глубоко вложен в другие циклы.
- ❑ Класс имеет плохую связность.
- ❑ Интерфейс класса не формирует согласованную абстракцию.
- ❑ Метод принимает слишком много параметров.
- ❑ Отдельные части класса изменяются независимо от других частей.
- ❑ При изменении программы требуется параллельно изменять несколько классов.
- ❑ Вам приходится параллельно изменять несколько иерархий наследования.
- ❑ Вам приходится параллельно изменять несколько блоков *case*.
- ❑ Родственные элементы данных, используемые вместе, не организованы в классы.
- ❑ Метод использует больше элементов другого класса, чем своего собственного.
- ❑ Элементарный тип данных перегружен.
- ❑ Класс имеет слишком ограниченную функциональность.
- ❑ По цепи методов передаются бродячие данные.
- ❑ Объект-посредник ничего не делает.
- ❑ Один класс слишком много знает о другом классе.
- ❑ Метод имеет неудачное имя.
- ❑ Данные-члены сделаны открытыми.
- ❑ Подкласс использует только малую долю методов своих предков.
- ❑ Сложный код объясняется при помощи комментариев.
- ❑ Код содержит глобальные переменные.
- ❑ Перед вызовом метода выполняется подготовительный код (после вызова метода выполняется код «уборки»).
- ❑ Программа содержит код, который может когда-нибудь понадобиться.

## Когда не следует выполнять рефакторинг?

Значение слова «рефакторинг» довольно размыто: так называют исправление дефектов, реализацию новой функциональности, модификацию проекта — по сути любое изменение кода. Это неуместно. Целенаправленный процесс изменений может быть эффективной стратегией, обеспечивающей постепенное повышение качества программы при ее сопровождении и предотвращающей всем известную смертельную спираль энтропии ПО, но само по себе изменение достоинств не имеет.

# 24.3. Отдельные виды рефакторинга

Ниже вы найдете список видов рефакторинга, многие из которых я сформулировал на основе более подробных описаний, приведенных в книге «Refactoring» (Fowler, 1999). Однако я не пытался сделать этот список исчерпывающим. В некотором смысле каждый пример плохого и аналогичного хорошего кода в этой книге претендует на то, чтобы быть видом рефакторинга. Ради экономии места я привел только те виды рефакторинга, которые счел наиболее полезными.

## Рефакторинг на уровне данных

Следующие виды рефакторинга связаны с использованием переменных и других видов данных.

***Замена магического числа на именованную константу*** Если вы используете численный или строковый литерал, скажем, *3.14*, замените его именованной константой, такой как *PI*.

***Присвоение переменной более ясного или информативного имени*** Если имя переменной неясно, присвойте ей лучшее имя. Конечно, этот же совет относится и к переименованию констант, классов и методов.

***Встраивание выражения в код*** Замените промежуточную переменную, которой присваивается результат вычисления выражения, на само выражение.

***Замена выражения на вызов метода*** Этот вид рефакторинга обычно служит для устранения из кода повторяющихся выражений.

***Введение промежуточной переменной*** Присвойте результат вычисления выражения промежуточной переменной, имя которой резюмирует суть выражения.

***Преобразование многоцелевой переменной в несколько одноцелевых переменных*** Если переменная используется более чем с одной целью (к частым подозреваемым относятся такие переменные, как *i*, *j*, *temp* и *x*), создайте для каждой из целей отдельную переменную, выбрав для нее более определенное имя.

***Использование локальной переменной вместо параметра*** Если исключительно входной параметр метода служит в качестве локальной переменной, подумайте, не лучше ли создать для этого настоящую локальную переменную.

***Преобразование элементарного типа данных в класс*** Если элементарный тип данных нужно расширить дополнительными формами поведения (например, более строгим контролем типа) или дополнительными данными, преобразуйте его в класс и реализуйте нужное поведение. Это относится и к простым численным типам вроде *Money* и *Temperature*, и к перечислениям, таким как *Color*, *Shape*, *Country* или *OutputType*.

***Преобразование набора кодов в класс или перечисление*** В более старых программах часто встречаются фрагменты вида:

```
const int SCREEN = 0;
const int PRINTER = 1;
const int FILE = 2;
```

Вместо определения таких отдельных констант лучше было бы создать класс *OutputType*: это обеспечило бы вам достоинства более строгого контроля типов и по-

зволило бы расширить семантику класса. Создание перечисления — иногда хорошая альтернатива созданию класса.

***Преобразование набора кодов в класс, имеющий производные классы*** Если разные элементы, ассоциированные с разными типами, могут иметь разное поведение, подумайте о создании базового класса для типа и производных классов для каждого кода типа. Так, для базового класса *OutputType* можно было бы создать подклассы *Screen*, *Printer* и *File*.

***Преобразование массива в класс*** Если элементами массива являются разные типы, создайте класс, включающий поля для каждого элемента массива.

***Инкапсуляция набора*** Если класс возвращает набор, наличие нескольких экземпляров набора может привести к проблемам с синхронизацией. Сделайте так, чтобы класс возвращал набор, допускающий только чтение, и создайте методы для добавления элементов в набор и их удаления.

***Замена традиционной записи на класс данных*** Создайте класс, содержащий члены записи (record). Это позволит централизовать проверку ошибок, слежение за персистентностью и выполнение других операций, касающихся записи.

## Рефакторинг на уровне отдельных операторов

***Декомпозиция логического выражения*** Упростите логическое выражение, введя грамотно названные промежуточные переменные, документирующие суть выражения.

***Вынесение сложного логического выражения в грамотно названную булеву функцию*** Если выражение достаточно сложное, этот вид рефакторинга повысит удобочитаемость кода. Если выражение используется более одного раза, он исключит необходимость внесения параллельных изменений и снизит вероятность ошибок.

***Консолидация фрагментов, повторяющихся в разных частях условного оператора*** Если в конце блока *else* содержится такой же фрагмент кода, что и в конце блока *if*, вынесите этот фрагмент за пределы оператора *if-then-else*.

***Использование оператора* break *или* return *вместо управляющей переменной цикла*** Если для управления циклом используется такая переменная, как *done*, удалите ее и выполняйте выход из цикла при помощи оператора *break* или *return*.

***Возврат из метода сразу после получения ответа вместо установки возвращаемого значения внутри вложенных операторов* if-then-else** Выход из метода сразу же после нахождения возвращаемого значения часто помогает облегчить чтение кода и снизить вероятность ошибок. Альтернативный вариант — установка возвращаемого значения и следование к выходу из метода через заторы операторов — может оказаться более сложным.

***Замена условных операторов (особенно многочисленных блоков* case*) на вызов полиморфного метода*** Значительную часть логики, обычно включаемой при структурном программировании в блоки *case*, можно реализовать с помощью наследования в виде полиморфных методов.

***Создание и использование «пустых» объектов вместо того, чтобы проверять, равно ли значение* null** Иногда «пустой» объект должен иметь обоб-

щенное поведение или обобщенные данные, например, определять неизвестного человека как «гражданина». В этом случае подумайте о перемещении ответственности за обработку значений *null* из клиентского кода в класс, т. е. вместо того, чтобы проверять в клиентском коде класса *Customer* (заказчик), известно ли имя заказчика, и подставлять значение «гражданин», если оно неизвестно, определите неизвестного заказчика как «гражданина» прямо в классе *Customer*.

## Рефакторинг на уровне отдельных методов

***Извлечение метода из другого метода*** Превратите фрагмент метода в отдельный метод.

***Встраивание кода метода*** Если метод прост и понятен, можете не вызывать его, а встроить его код в программу.

***Преобразование объемного метода в класс*** Если метод слишком велик, попробуйте превратить его в класс и разбить на несколько методов. Иногда это повышает удобочитаемость кода.

***Замена сложного алгоритма на простой*** Этот вид рефакторинга пояснений не требует.

***Добавление параметра*** Если метод должен получать из вызывающего кода больше информации, передавайте ее в форме дополнительного параметра.

***Удаление параметра*** Если метод не нуждается в каком-то параметре, удалите его.

***Отделение операций запроса данных от операций изменения данных*** Как правило, операции запроса данных не должны изменять состояние объекта. Если операция вроде *GetTotals()* (получение итоговой суммы) изменяет состояние объекта, разделите функциональность запроса и функциональность изменения состояния объекта, создав два отдельных метода.

***Объединение похожих методов путем их параметризации*** Два похожих метода могут различаться только используемой в них константой. Объедините их в один метод и передавайте в него нужное значение как параметр.

***Разделение метода, поведение которого зависит от полученных параметров*** Если метод выполняет разный код в зависимости от значения входного параметра, подумайте о разбиении метода на отдельные методы, которые можно было бы вызывать, не передавая этот входной параметр.

***Передача в метод целого объекта вместо отдельных полей*** Если вы передаете в метод несколько значений одного объекта, подумайте о таком изменении интерфейса метода, чтобы он принимал сразу весь объект.

***Передача в метод отдельных полей вместо целого объекта*** Если вы создаете объект только затем, чтобы передать его в метод, подумайте о таком изменении метода, чтобы он принимал отдельные поля, а не весь объект.

***Инкапсуляция нисходящего приведения типов*** При возвращении объекта из метода обычно следует возвращать максимально определенный тип объекта. Это особенно справедливо для методов, возвращающих объекты-итераторы, наборы, элементы наборов и т. д.

## Рефакторинг реализации классов

***Замена объектов-значений на объекты-ссылки*** Если вы создаете и поддерживаете много копий крупных или сложных объектов, измените подход так, чтобы существовал только один оригинал объекта (объект-значение), а в остальном коде использовались ссылки на этот объект (объекты-ссылки).

***Замена объектов-ссылок на объекты-значения*** Если вам приходится прилагать усилия для обработки ссылок на небольшие или простые объекты, сделайте все объекты-ссылки объектами-значениями.

***Замена виртуальных методов на инициализацию данных*** Если у вас есть набор подклассов, различающихся только возвращаемыми константами, не переопределяйте методы-члены в подклассах, а инициализируйте базовый класс в подклассах соответствующими константами и включите в него обобщенный код, использующий эти константы.

***Изменение положения методов-членов или данных-членов в иерархии наследования*** Подумайте о внесении в иерархию наследования нескольких общих изменений. Следующие изменения обычно выполняются для устранения повторений кода в производных классах:

- перемещение метода в суперкласс;
- перемещение поля в суперкласс;
- перемещение тела конструктора в суперкласс.

Другие изменения обычно вносятся с целью поддержки специализации в производных классах:

- перемещение метода в производные классы;
- перемещение поля в производные классы;
- перемещение тела конструктора в производные классы.

***Перемещение специализированного кода в подкласс*** Если какой-то код класса используется только в подмножестве его экземпляров, переместите этот специализированный код в отдельный подкласс.

***Объединение похожего кода и его перемещение в суперкласс*** Если несколько подклассов включают похожий код, объедините этот код и переместите его в суперкласс.

## Рефакторинг интерфейсов классов

***Перемещение метода в другой класс*** Создайте в целевом классе новый метод и переместите тело метода из исходного класса в целевой класс. После этого вы можете вызывать новый метод из старого.

***Разделение одного класса на несколько*** Если класс имеет более одной области ответственности, разбейте его на несколько классов, имеющих ясно определенные области ответственности.

***Удаление класса*** Если класс почти ничего не делает, переместите его код в другие, более связные классы и удалите его.

***Сокрытие делегата*** Иногда Класс А вызывает и Класс В, и Класс С, тогда как на самом деле А должен вызывать только В, а В — С. Спросите себя, какова пра-

вильная абстракция взаимодействия Класса A с Классом B. Если за вызов C должен отвечать B, внесите нужные изменения.

***Удаление посредника*** Если Класс A вызывает Класс B, а Класс B вызывает Класс C, подумайте, не лучше ли вызывать C непосредственно из A. Целесообразность делегирования полномочий Классу B зависит от того, улучшит ли это целостность его интерфейса или ухудшит.

***Замена наследования на делегирование*** Если из одного класса нужно использовать другой класс, но вы хотите получить больший контроль над интерфейсом второго класса, сделайте суперкласс полем бывшего подкласса и создайте для доступа к нему набор открытых методов, формирующих связную абстракцию.

***Замена делегирования на наследование*** Если класс предоставляет доступ ко всем открытым методам класса-делегата (класса-члена), выполните наследование от класса-делегата, а не просто используйте его.

***Создание внешнего метода*** Если в класс нужно включить дополнительный метод, но изменять класс нельзя, вы можете создать нужный метод в клиентском классе.

***Создание класса-расширения*** Если в класс нужно включить несколько дополнительных методов, но изменять класс нельзя, вы можете создать новый класс, объединяющий функциональность неизменяемого класса с дополнительной функциональностью. Для этого вы можете или выполнить наследование от исходного класса и добавить новые методы в подклассы, или заключить класс в оболочку, предоставив доступ к нужным методам.

***Инкапсуляция открытой переменной-члена*** Если данные-члены открыты, сделайте их закрытыми и реализуйте доступ к ним при помощи методов.

***Удаление методов установки значений неизменяемых полей*** Если поле предполагается устанавливать во время создания объекта и не изменять впоследствии, инициализируйте поле в конструкторе объекта и не создавайте вводящий в заблуждение метод *Set()*.

***Сокрытие методов, которые не следует вызывать извне класса*** Если без метода интерфейс класса будет более согласованным, скройте метод.

***Инкапсуляция неиспользуемых методов*** Если обычно вы используете только часть интерфейса класса, создайте новый интерфейс, предоставляющий доступ только к необходимым методам. Убедитесь в том, что новый интерфейс формирует согласованную абстракцию.

***Объединение суперкласса и подкласса, имеющих очень похожую реализацию*** Если степень специализации подкласса невысока, объедините его с суперклассом.

## Рефакторинг на уровне системы

***Создание эталонного источника данных, которые вы не можете контролировать*** Иногда какие-то данные трудно согласованно использовать из других объектов, которым нужны эти данные. В качестве примера можно привести данные элемента управления с GUI-интерфейсом. В этом случае вы можете создать

класс, воспроизводящий данные элемента управления, и рассматривать этот класс как эталонный источник данных и для элемента управления, и для другого кода.

***Изменение однонаправленной связи между классами на двунаправленную*** Если два класса должны использовать возможности друг друга, но только один класс знает о другом классе, измените классы так, чтобы они оба знали друг о друге.

***Изменение двунаправленной связи между классами на однонаправленную*** Если два класса известны друг другу, но на самом деле только один класс должен знать о другом, измените характер связи между классами.

***Предоставление фабричного метода вместо простого конструктора*** Используйте фабричный метод, если вам нужно создавать объекты на основе кода типа или если вы хотите работать с объектами-ссылками, а не объектами-значениями.

***Замена кодов ошибок на исключения или наоборот*** Убедитесь, что вы используете стандартный подход к обработке ошибок, основанный на той или иной стратегии.

http://cc2e.com/2450

### Контрольный список: виды рефакторинга

**Рефакторинг на уровне данных**

- ❑ Замена магического числа на именованную константу.
- ❑ Присвоение переменной более ясного или информативного имени.
- ❑ Встраивание выражения в код.
- ❑ Замена выражения на вызов метода.
- ❑ Введение промежуточной переменной.
- ❑ Преобразование многоцелевой переменной в несколько одноцелевых переменных.
- ❑ Использование локальной переменной вместо параметра.
- ❑ Преобразование элементарного типа данных в класс.
- ❑ Преобразование набора кодов в класс или перечисление.
- ❑ Преобразование набора кодов в класс, имеющий производные классы.
- ❑ Преобразование массива в класс.
- ❑ Инкапсуляция набора.
- ❑ Замена традиционной записи на класс данных.

**Рефакторинг на уровне отдельных операторов**

- ❑ Декомпозиция логического выражения.
- ❑ Вынесение сложного логического выражения в грамотно названную булеву функцию.
- ❑ Консолидация фрагментов, повторяющихся в разных частях условного оператора.
- ❑ Использование оператора *break* / *return* вместо управляющей переменной цикла.
- ❑ Возврат из метода сразу после получения ответа вместо установки возвращаемого значения внутри вложенных операторов *if-then-else*.
- ❑ Замена условных операторов (особенно многочисленных блоков *case*) на вызов полиморфного метода.
- ❑ Создание и использование «пустых» объектов вместо проверки того, равно ли значение *null*.

**Рефакторинг на уровне отдельных методов**

- ❑ Извлечение метода из другого метода.
- ❑ Встраивание кода метода.
- ❑ Преобразование объемного метода в класс.
- ❑ Замена сложного алгоритма на простой.
- ❑ Добавление параметра.
- ❑ Удаление параметра.
- ❑ Отделение операций запроса данных от операций изменения данных.
- ❑ Объединение похожих методов при помощи их параметризации.
- ❑ Разделение метода, поведение которого зависит от полученных параметров.
- ❑ Передача в метод целого объекта вместо отдельных полей.
- ❑ Передача в метод отдельных полей вместо целого объекта.
- ❑ Инкапсуляция нисходящего приведения типов.

**Рефакторинг реализации классов**

- ❑ Замена объектов-значений на объекты-ссылки.
- ❑ Замена объектов-ссылок на объекты-значения.
- ❑ Замена виртуальных методов на инициализацию данных.
- ❑ Изменение положения методов-членов или данных-членов в иерархии наследования.
- ❑ Перемещение специализированного кода в подкласс.
- ❑ Объединение похожего кода и его перемещение в суперкласс.

**Рефакторинг интерфейсов классов**

- ❑ Перемещение метода в другой класс.
- ❑ Разделение одного класса на несколько.
- ❑ Удаление класса.
- ❑ Сокрытие делегата.
- ❑ Удаление посредника.
- ❑ Замена наследования на делегирование.
- ❑ Замена делегирования на наследование.
- ❑ Создание внешнего метода.
- ❑ Создание класса-расширения.
- ❑ Инкапсуляция открытой переменной-члена.
- ❑ Удаление методов установки значений неизменяемых полей.
- ❑ Сокрытие методов, которые не следует вызывать извне класса.
- ❑ Инкапсуляция неиспользуемых методов.
- ❑ Объединение суперкласса и подкласса, имеющих очень похожую реализацию.

**Рефакторинг на уровне системы**

- ❑ Создание эталонного источника данных, которые вы не можете контролировать.
- ❑ Изменение однонаправленной связи между классами на двунаправленную.
- ❑ Изменение двунаправленной связи между классами на однонаправленную.
- ❑ Предоставление фабричного метода вместо простого конструктора.
- ❑ Замена кодов ошибок на исключения или наоборот.

## 24.4. Безопасный рефакторинг

> Вмешательство в рабочую систему больше похоже на вскрытие головного мозга и замену нерва, чем на замену прокладки в кране. Облегчилось ли бы сопровождение программ, если б оно называлось «нейрохирургией ПО»?
>
> *Джеральд Вайнберг (Gerald Weinberg)*

Рефакторинг — эффективный способ повышения качества кода. Но, как и все эффективные инструменты, при неверном использовании он может причинить вред. Несколько простых советов помогут предотвратить неверное применение рефакторинга.

***Сохраняйте первоначальный код*** Перед началом рефакторинга убедитесь, что вы сможете вернуться к коду, с которого начинаете. Сохраните код в системе управления версиями или скопируйте корректные файлы в резервный каталог.

***Стремитесь ограничить объем отдельных видов рефакторинга*** Некоторые виды рефакторинга масштабнее других, к тому же не всегда можно точно сказать, что именно составляет «один вид рефакторинга». Чтобы четко представлять все следствия вносимых изменений, не раздувайте виды рефакторинга. Примеры соблюдения этого принципа см. в книге «Refactoring» (Fowler, 1999).

***Выполняйте отдельные виды рефакторинга по одному за раз*** Некоторые виды рефакторинга сложнее других. За исключением самых простых случаев выполняйте все виды рефакторинга по одному за раз, компилируя и тестируя программу перед следующим видом рефакторинга.

***Составьте список действий, которые вы собираетесь предпринять*** Естественным расширением Процесса Программирования с Псевдокодом является составление списка видов рефакторинга, которые приведут вас из точки А в точку Б. Составление такого списка поможет поддерживать каждое изменение в соответствующем контексте.

***Составьте и поддерживайте список видов рефакторинга, которые следует выполнить позже*** Выполняя один вид рефакторинга, вы можете счесть необходимым еще один вид. Взявшись за него, вы можете обнаружить в коде недостатки, призывающие к третьему виду. Если изменение не требуется сию минуту, включите его в список изменений, которые следовало бы внести в программу когда-то, но не обязательно вносить прямо сейчас.

***Часто создавайте контрольные точки*** Иногда рефакторинг внезапно уходит в сторону, поэтому вам следует не только сохранять первоначальный код, но и создавать контрольные точки на разных этапах рефакторинга, чтобы можно было вернуться к работоспособной программе, если рефакторинг заведет вас в тупик.

***Используйте предупреждения компилятора*** Компилятор часто не замечает небольших ошибок. Задав компилятору самый строгий уровень диагностики, вы сможете исправлять многие ошибки почти сразу после их внесения.

***Выполняйте регрессивное тестирование*** Дополните обзоры измененного кода регрессивным тестированием. Конечно, это зависит от наличия хорошего набора тестов (см. главу 22).

***Создавайте дополнительные тесты*** Не ограничивайтесь регрессивным тестированием программы с использованием старых тестов — создавайте новые блочные тесты для проверки нового кода. Тесты, устаревшие в результате рефакторинга, удаляйте.

***Выполняйте обзоры изменений*** Если обзоры важны при первоначальном написании кода, то при последующих изменениях их важность лишь повышается. Эд Йордон сообщает, что первая попытка внесения изменения в код более чем в половине случаев оказывается ошибочной (Yourdon, 1986b). Интересно, что, если программисты имеют дело не с несколькими строками кода, а с более объемным фрагментом, вероятность внесения корректного изменения более высока (рис. 24-1). Точнее говоря, по мере увеличения числа изменяемых строк с одной до пяти вероятность внесения неправильного изменения повышается, а после этого снижается.

**Перекрестная ссылка** Об обзорах см. главу 21.

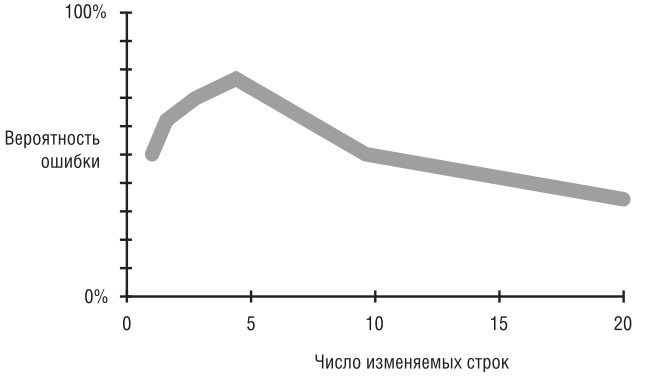



***Рис. 24-1.*** *Небольшие изменения чаще оказываются ошибочными, чем более крупные (Weinberg, 1983)*

Программисты относятся к небольшим изменениям легкомысленно. Они не анализируют их, не просят коллег выполнить их обзор, а иногда даже не запускают программу, чтобы проверить, что исправление корректно.

Мораль проста: рассматривайте простые изменения так, как если бы они были сложными. В одной организации, в которой были введены обзоры изменений одной строки кода, было обнаружено, что доля ошибочных изменений снизилась с 55% до 2% (Freedman and Weinberg, 1982). В другой организации, работающей в сфере телекоммуникаций, введение обзоров изменений позволило повысить их корректность с 86% до 99,6% (Perrott, 2004).

***Изменяйте подход в зависимости от рискованности рефакторинга*** Некоторые виды рефакторинга сопряжены с более высоким риском, чем другие. Так, «замена магического числа на именованную константу» относительно безопасна. Виды рефакторинга, предполагающие изменение интерфейса класса или метода, схемы БД или булевых тестов, обычно более рискованны. Если рефакторинг несложен, вы можете оптимизировать процесс рефакторинга, выполняя более одного вида за раз и ограничиваясь регрессивным тестированием кода без его официального обзора.

В случае более рискованных видов рефакторинга будьте более осторожны. Выполняйте их по одному за раз. Попросите кого-то провести обзор изменения или выполните рефакторинг в паре, дополнив этим обычную проверку кода средствами компилятора и блочное тестирование.

### Плохие причины выполнения рефакторинга

Несмотря на всю свою эффективность, рефакторинг — не панацея и допускает несколько специфических видов злоупотребления.

> Не реализуйте какую-то возможность частично, намереваясь позднее завершить ее в процессе рефакторинга.
>
> *Джон Манзо* (*John Manzo*)

***Не рассматривайте рефакторинг как оправдание написания плохого кода с намерением исправить его позднее*** Самой большой проблемой с рефакторингом является его неверное применение. Иногда программисты говорят, что они выполняют рефакторинг, хотя на самом деле они просто пробуют что попало в надежде хоть как-то привести код в работоспособное состояние. Рефакторинг — это *изменение работоспособного кода*, не влияющее на поведение программы. Программисты, которые возятся с плохим кодом, не выполняют рефакторинг — они занимаются хакерством.

> Крупномасштабный рефакторинг — путь к катастрофе.
>
> *Кент Бек* (*Kent Beck*)

***Не рассматривайте рефакторинг как способ, позволяющий избежать переписывания кода*** Иногда код невозможно улучшить небольшими изменениями — его нужно выбросить и переписать с нуля. Если вы выполняете крупномасштабный рефакторинг, спросите себя, не следует ли вместо этого перепроектировать и переписать фрагмент кода.

## 24.5. Стратегии рефакторинга

Число видов рефакторинга, выгодных для любой конкретной программы, практически бесконечно. Рефакторинг подчиняется тому же закону снижения выгоды, что и другие процессы программирования, и к нему также относится правило 80/20. Тратьте время на 20% видов рефакторинга, обеспечивающих 80% выгоды. При определении наиболее важных видов рефакторинга учитывайте следующие советы.

***Выполняйте рефакторинг при создании новых методов*** Создавая метод, проверьте, хорошо ли организованы связанные с ним методы. При необходимости выполните их рефакторинг.

***Выполняйте рефакторинг при создании новых классов*** Создание нового класса часто подчеркивает недостатки имеющегося кода. Используйте эту возможность для рефакторинга других классов, тесно взаимодействующих с новым классом.

***Выполняйте рефакторинг при исправлении дефектов*** Используйте знания, полученные при исправлении ошибки, для улучшения других фрагментов кода, которые могут быть подвержены похожим ошибкам.

***Выполняйте рефакторинг модулей, подверженных ошибкам*** Некоторые модули более подвержены ошибкам, чем другие. Есть ли в коде фрагмент, внушающий страх вам и всем остальным членам вашей группы? Скорее всего он под-

вержен ошибкам. Большинство людей естественным образом стараются избегать таких проблемных фрагментов кода, однако вы добьетесь большего успеха, если будете рассматривать их как мишени рефакторинга (Jones, 2000).

**Перекрестная ссылка** О коде, подверженном ошибкам, см. подраздел «Какие классы содержат наибольшее число ошибок?» раздела 22.4.

***Выполняйте рефакторинг сложных модулей*** Другим подходом к рефакторингу является концентрация на модулях, имеющих максимальные оценки сложности (см. подраздел «Как измерить сложность» раздела 19.6). Одно классическое исследование показало, что использование этой методики при сопровождении программ приводило к существенному повышению их качества (Henry and Kafura, 1984).

***При сопровождении программы улучшайте фрагменты, к которым прикасаетесь*** Код, который никогда не изменяется, не требует рефакторинга. Но если вы все же изменяете фрагмент кода, убедитесь, что вы оставляете его в лучшем состоянии, чем обнаружили.

***Определите интерфейс между аккуратным и безобразным кодом и переместите безобразный код на другую сторону этого интерфейса*** «Реальность» часто оказывается грязнее, чем нам хотелось бы. Эта грязь может быть обусловлена сложными бизнес-правилами, интерфейсами с оборудованием, программными интерфейсами и т. д. При работе со старыми системами часто приходится иметь дело с плохим кодом, который тем не менее должен постоянно оставаться работоспособным.

Эффективной стратегией омоложения внедренных старых систем является определение фрагментов, относящихся к грязному реальному миру, фрагментов, формирующих идеализированный новый мир, и фрагментов, определяющих интерфейс между двумя мирами (рис. 24-2).

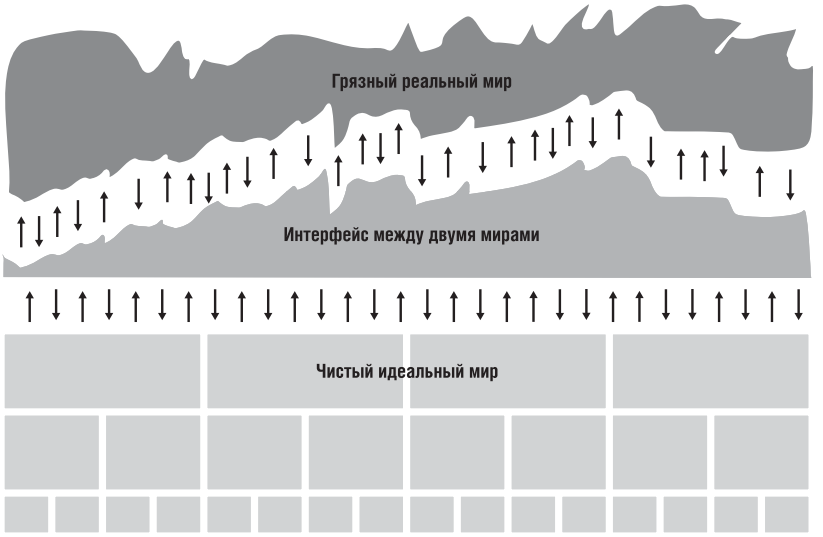



***Рис. 24-2.*** *Ваш код не обязан быть грязным только потому, что таков реальный мир. Рассматривайте систему как комбинацию идеального кода, грязного кода и интерфейса между идеальным и грязным кодом*

Работая с системой, вы можете постепенно перемещать грязный код через «интерфейс между двумя мирами» в более организованный идеальный мир. Если вы только начинаете работать с унаследованной системой, она может почти полностью относиться к грязному миру. Одна эффективная политика подразумевает, что каждый раз, когда вы прикасаетесь к какому-то фрагменту грязного кода, вы должны привести его в соответствие с текущими стандартами кодирования, присвоить переменным ясные имена и т. д. — иначе говоря, переместить его в идеальный мир. Со временем это может обеспечить быстрое улучшение базы кода (рис. 24-3).

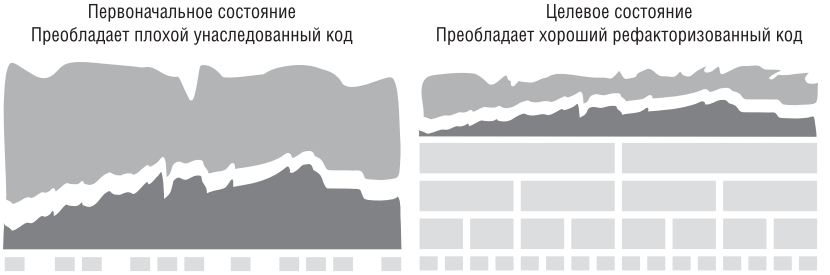



***Рис. 24-3.*** *Одной из стратегий улучшения готовой системы является постепенный рефакторинг плохого унаследованного кода — перемещение его на другую сторону «интерфейса между двумя мирами»*

http://cc2e.com/2457

### Контрольный список: безопасный рефакторинг

- ❑ Является ли каждое изменение частью систематичной стратегии изменений?
- ❑ Сохранили ли вы первоначальный код перед началом рефакторинга?
- ❑ Стараетесь ли вы ограничить объем каждого вида рефакторинга?
- ❑ Выполняете ли вы отдельные виды рефакторинга по одному за раз?
- ❑ Составили ли вы список действий, которые собираетесь предпринять во время рефакторинга?
- ❑ Ведете ли вы список видов рефакторинга, которые следовало бы выполнить позднее?
- ❑ Выполняете ли вы регрессивное тестирование после каждого вида рефакторинга?
- ❑ Выполняете ли вы обзор сложных изменений и изменений, влияющих на критически важный код?
- ❑ Рассматриваете ли вы рискованность отдельных видов рефакторинга и адаптируете ли вы свой подход соответствующим образом?
- ❑ Убеждаетесь ли вы, что изменения улучшают внутреннее качество программы, а не ухудшают его?
- ❑ Не рассматриваете ли вы рефакторинг как оправдание написания плохого кода или как способ избежать переписывания плохого кода?

# Дополнительные ресурсы

http://cc2e.com/2464

Процесс рефакторинга имеет много общего с процессом устранения дефектов (см. раздел 23.3). Факторы риска, связанные с рефакторингом, похожи на факторы риска, касающиеся оптимизации кода. Об управлении факторами риска при оптимизации кода см. раздел 25.6.

Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison Wesley, 1999. Это самое полное и подробное руководство по рефакторингу содержит детальное обсуждение многих конкретных видов рефакторинга, упомянутых в этой главе, а также других видов, которых я не касался. Фаулер привел много примеров пошагового выполнения каждого вида рефакторинга.

# Ключевые моменты

- Изменения программы неизбежны как во время первоначальной разработки, так и после выпуска первой версии.
- Изменения могут приводить как к улучшению, так и к ухудшению ПО. Главное Правило Эволюции ПО заключается в том, что при эволюции кода внутреннее качество программы должно повышаться.
- Одним из условий успешности рефакторинга является пристальное внимание к многочисленным предупреждающим знакам — «запахам», указывающим на необходимость рефакторинга.
- Другое условие — изучение многих конкретных видов рефакторинга.
- Заключительным условием успешности рефакторинга является следование стратегии безопасного рефакторинга. Одни подходы к рефакторингу лучше, а другие хуже.
- Рефакторинг во время разработки — самая благоприятная возможность улучшения программы и внесения в нее всех изменений, которые вам так или иначе захочется внести позднее. Используйте эту возможность!

ГЛАВА 25

# Стратегии оптимизации кода

http://cc2e.com/2578

**Содержание**



**Связанные темы**

- Методики оптимизации кода: глава 26
- Архитектура ПО: раздел 3.5

В этой главе обсуждается исторически противоречивая проблема — повышение производительности кода. В 1960-х годах ресурсы компьютеров были крайне ограниченны, поэтому эффективность их использования была вопросом первостепенной важности. По мере роста производительности компьютеров в 70-х программисты начали понимать, насколько упор на производительность вредит удобочитаемости и легкости сопровождения кода, и оптимизация кода отошла на задний план. Вместе с микрокомпьютерной революцией, свидетелями которой мы стали в 80-х, проблема эффективного использования ресурсов вернулась, но в 90-х ее важность постепенно уменьшилась. В 2000-х мы опять столкнулись с этой проблемой, только теперь она связана с ограниченной памятью мобильных телефонов, карманных ПК и подобных устройств, а также со временем выполнения интерпретируемого кода.

Проблемы с производительностью можно решать на двух уровнях: стратегическом и тактическом. В этой главе рассматриваются стратегические вопросы производительности: мы выясним, что такое производительность, насколько она важна, и обсудим общий подход к ее повышению. Если стратегии достижения нужной производительности вам уже хорошо известны и вы хотите узнать конкретные

методики ее повышения на уровне кода, можете перейти к главе 26. Однако прежде чем приступать к серьезной работе над повышением производительности, хотя бы просмотрите эту главу, чтобы не потратить время на оптимизацию кода тогда, когда следовало бы заняться чем-то другим.

# 25.1. Общее обсуждение производительности ПО

Оптимизация кода — лишь один из способов повышения производительности программы. Часто можно найти другие способы, обеспечивающие большее повышение производительности за меньшее время и с меньшим вредом для кода.

## Характеристики качества и производительность

> Во имя эффективности — причем достигается она далеко не всегда — совершается больше компьютерных грехов, чем по любой другой причине, включая банальную глупость.
>
> *У. А. Вульф (W. A. Wulf)*

Некоторые люди смотрят на мир через розовые очки. Программисты склонны воспринимать мир через кодовые очки. Мы полагаем, что чем лучше будет наш код, тем сильнее наше ПО понравится клиентам.

Эта точка зрения верна лишь отчасти. Пользователей больше интересуют явные характеристики программы, а не качество кода. Производительность обычно привлекает их внимание, только когда она сказывается на их работе. Большее значение для пользователей имеет не грубая производительность приложения, а объем информации, который оно позволяет обработать за конкретный срок, а также такие факторы, как своевременное получение программы, ясный пользовательский интерфейс и предотвращение простоев.

Приведу пример. Я делаю цифровой камерой минимум 50 снимков в неделю. Чтобы скопировать снимки на компьютер при помощи ПО, поставляемого с камерой, я должен выбрать каждый снимок по очереди, причем в окне программы отображаются только 6 снимков сразу. В результате копирование 50 изображений превращается в долгий и нудный процесс, требующий десятков щелчков и массы переключений между окнами. Устав от всего этого, я купил карту памяти, подключаемую прямо к компьютеру и воспринимаемую им как диск. Теперь для копирования изображений на диск компьютера я могу использовать Проводник Windows. Я делаю пару щелчков, нажимаю Ctrl+A и перетаскиваю все файлы в нужное место. Меня не волнует, передает ли карта памяти каждый файл вдвое медленнее или быстрее, чем другое ПО, потому что сейчас я могу обработать больший объем информации за меньшее время. Каким бы быстрым или медленным ни был код драйвера карты памяти, его производительность выше.

Производительность только косвенно связана с быстродействием кода. Чем больше вы работаете над скоростью кода, тем меньше внимания уделяете другим характеристикам его качества. Не приносите их в жертву быстродействию. Стремление к повышению быстродействия может снизить общую производительность программы, а не повысить ее.

## Производительность и оптимизация кода

Решив, что эффективность кода — будь то его быстродействие или объем — должна быть приоритетом, не торопитесь улучшать быстродействие или объем на уровне кода, а рассмотрите несколько вариантов. Подумайте об эффективности в контексте:

- требований к программе;
- проекта программы;
- проектов классов и методов;
- взаимодействия программы с ОС;
- компиляции кода;
- оборудования;
- оптимизации кода.

### Требования к программе

Высокая производительность считается требованием гораздо чаще, чем на самом деле им является. Барри Бом рассказывает, что при разработке одной системы в компании TRW сначала решили, что время реакции системы не должно превышать 1 секунды. Это требование привело к разработке очень сложного проекта, на реализацию которого пришлось бы потратить примерно 100 млн долларов. Дальнейший анализ показал, что в 90% случаев пользователей устроит время реакции, равное 4 секундам. Изменение требования позволило снизить общую стоимость системы примерно на 70 млн долларов (Boehm, 2000b).

Прежде чем тратить время на решение проблемы с производительностью, убедитесь, что она действительно требует решения.

### Проект программы

**Перекрестная ссылка** Проектирование высокопроизводительных программ рассматривается в работах, указанных в разделе «Дополнительные ресурсы» в конце главы.

Проект программы определяет ее основные черты — главным образом, способ ее разделения на классы. Проект может как облегчить, так и затруднить создание высокопроизводительной системы.

Например, при высокоуровневом проектировании одной реальной программы сбора и обработки данных в качестве ключевого атрибута была определена пропускная способность обработки результатов измерений. Каждое измерение включало определение значения электрической величины, калибровку значения, масштабирование значения и преобразование исходных единиц измерения (таких как милливольты) в единицы прикладной области (такие как градусы Цельсия).

Если бы при высокоуровневом проектировании программисты не оценили все факторы риска, им в итоге пришлось бы оптимизировать алгоритмы вычисления многочленов 13 степени, т. е. многочленов, содержащих 14 переменных с максимальной степенью 13. Вместо этого они решили проблему, выбрав другое оборудование и разработав высокоуровневый проект, предусматривающий использование десятков многочленов 3 степени. Оптимизировав код, они вряд ли получили бы нужные результаты. Это пример проблемы, которую нужно было решать на уровне проектирования.

Если объем и быстродействие программы действительно важны, разработайте архитектуру так, чтобы она позволяла добиться намеченных показателей, а затем задайте для отдельных подсистем, функций и классов целевые показатели использования ресурсов. Преимущества такого подхода таковы.

**Перекрестная ссылка** О том, как программисты стремятся к достижению поставленных целей, см. подраздел «Задание целей» раздела 20.2.

- Задание отдельных целевых показателей использования ресурсов делает производительность системы предсказуемой. Если каждая функция соответствует целевым показателям, вся система будет обладать нужной производительностью. Вы можете уже на ранних этапах определить проблемные подсистемы и перепроектировать их или оптимизировать их код.
- Простое задание явных целей повышает вероятность их достижения. Программисты стремятся к достижению целей, если знают, каковы они; чем определеннее цели, тем легче к ним стремиться.

- Вы можете поставить цели, которые непосредственно не направлены на повышение эффективности, но способствуют этому в долгосрочной перспективе. Эффективность часто лучше всего рассматривать в контексте других аспектов. Так, простота внесения изменений может создать лучшие условия для достижения высокой эффективности, чем явное определение эффективности как одной из целей. Если проект отличается высокой степенью модульности и простотой изменения, менее эффективные компоненты можно с легкостью заменить на более эффективные.

### Проекты классов и методов

Проектирование классов и методов предоставляет еще одну возможность повышения производительности ПО. Одним из способов повышения производительности на этом уровне является выбор типа данных или алгоритма, от чего обычно зависит и объем используемой памяти, и быстрота выполнения кода. Именно на этом уровне вы можете выбрать быструю сортировку вместо пузырьковой или двоичный поиск вместо линейного.

**Перекрестная ссылка** О типах данных и алгоритмах см. книги, указанные в разделе «Дополнительные ресурсы» в конце главы.

### Взаимодействие программы с ОС

Если ваша программа работает с внешними файлами, динамической памятью или устройствами вывода, она скорее всего взаимодействует с ОС. Низкая производительность в этом случае может объясняться большим объемом или медленным выполнением методов ОС. Вы можете даже не знать, что программа взаимодействует с ОС: иногда вызовы системных методов генерируются компилятором или содержатся в коде библиотек.

**Перекрестная ссылка** О стратегиях борьбы со слишком медленными или объемными методами ОС на уровне кода см. главу 26.

### Компиляция кода

Хорошие компиляторы преобразуют ясный высокоуровневый код в оптимизированный машинный код. Иногда правильный выбор компилятора позволяет вообще забыть о дальнейшей оптимизации кода.

В главе 26 вы найдете многочисленные примеры ситуаций, когда код, сгенерированный компилятором, оказывается эффективнее кода, оптимизированного вручную.

### Оборудование

Иногда самым выгодным и эффективным способом повышения производительности программы является покупка нового оборудования. Конечно, если программа предназначен для сотнен или тысяч компьютеров по всей стране, этот вариант нереалистичен. Но если вы разрабатываете специализированное ПО для нескольких пользователей, обновление оборудования на самом деле может оказаться самым дешевым вариантом. Это позволит сократить расходы, связанные с улучшением производительности ПО и проблемами, которые могут из-за этого улучшения возникнуть при сопровождении. Кроме того, это повысит производительность всех программ, выполняемых на новых системах.

### Оптимизация кода

Оптимизацией кода (code tuning), которой посвящена оставшаяся часть этой главы, называют изменение корректного кода, направленное на повышение его эффективности. «Оптимизация» подразумевает внесение небольших изменений, затрагивающих один класс, один метод, а чаще всего — несколько строк кода. Крупномасштабные изменения проекта или другие высокоуровневые способы повышения производительности оптимизацией не считаются.

Каждый из уровней от проектирования системы до оптимизации кода допускает существенное повышение производительности ПО. Джон Бентли утверждает, что в некоторых случаях формула общего повышения производительности системы имеет мультипликативный характер (Bentley, 1982). Так как каждый из шести уровней допускает десятикратный рост производительности, значит, что теоретически производительность программы может быть повышена в миллион раз. Хотя для этого необходимо, чтобы результаты, получаемые на каждом из уровней, были независимы от других уровней (что наблюдается редко), этот потенциал вдохновляет.

## 25.2. Введение в оптимизацию кода

Чем соблазняет оптимизация кода? Это не самый эффективный способ повышения производительности: улучшение архитектуры программы, перепроектирование классов и выбор более эффективного алгоритма обычно приводят к более впечатляющим результатам. Кроме того, это не самый легкий способ повысить производительность: легче купить новое оборудование или компилятор с улучшенным модулем оптимизации. Наконец, это не самый дешевый способ повысить производительность: на оптимизацию кода вручную изначально уходит много времени, а потом оптимизированный код труднее сопровождать.

Оптимизация кода привлекательна по ряду причин. Прежде всего она бросает вызов законам природы. Нет ничего радостнее, чем ускорить выполнение метода в 10 раз путем изменения всего лишь нескольких его строк.

Кроме того, овладение мастерством написания эффективного кода — признак превращения в серьезного программиста. В теннисе вы не получаете очков за то, как берете мяч, но все же должны освоить правильный способ делать это. Вы не

можете просто наклониться и взять его рукой. Если вы не хотите показаться новичком, вы ударяете по нему ракеткой, пока он не подпрыгнет до пояса, и тогда вы его ловите. Более трех ударов мяча о землю — серьезная оплошность. Несмотря на кажущуюся незначительность, способ подбора мяча считается в культуре теннисистов отличительным признаком. Компактность вашего кода также обычно не волнует никого, кроме вас и других программистов. Тем не менее в программисткой культуре способность создавать компактный и эффективный код служит подтверждением вашего класса.

Увы, эффективный код не всегда является «лучшим». Этот вопрос мы и обсудим ниже.

## Принцип Парето

Принцип Парето, известный также как «правило 80/20», гласит, что 80% результата можно получить, приложив 20% усилий. Относящийся не только к программированию, этот принцип очень точно характеризует оптимизацию программ.

Барри Бом сообщает, что на 20% методов программы приходятся 80% времени ее выполнения (Boehm, 1987b). В классической работе «An Empirical Study of Fortran Programs» Дональд Кнут указал, что менее 4% кода обычно соответствуют более чем 50% времени выполнения программы (Knuth, 1971).

Кнут обнаружил это неожиданное отношение при помощи инструмента профилирования, поддерживающего подсчет строк. Следствие очевидно: вам нужно найти в коде «горячие точки» и сосредоточиться на оптимизации процентов, используемых более всего. Профилируя свою программу подсчета строк, Кнут обнаружил, что половину времени она проводила в двух циклах. Он изменил несколько строк кода и удвоил скорость профайлера менее чем за час.

Джон Бентли описывает случай, когда программа из 1000 строк проводила 80% времени в 5-строчном методе вычисления квадратного корня. Утроив быстродействие этого метода, он удвоил быстродействие программы (Bentley, 1988). Опираясь на принцип Парето, можно дать еще один совет: напишите большую часть кода на интерпретируемом языке (скажем, на Python), а потом перепишите проблемные фрагменты на более быстром компилируемом языке, таком как C.

Бентли также сообщает о случае, когда группа обнаружила, что ОС половину времени проводит в одном небольшом цикле. Переписав цикл на микрокоде, разработчики ускорили его выполнение в 10 раз, но производительность системы осталась прежней — они переписали цикл бездействия системы!

Разработчики языка ALGOL — прародителя большинства современных языков, сыгравшего одну из самых главных ролей в истории программирования, — руководствовались принципом «Лучшее — враг хорошего». Стремление к совершенству может мешать завершению работы. Доведите работу до конца и только потом совершенствуйтесь. Часть, которую нужно довести до совершенства, обычно невелика.

## Бабушкины сказки

С оптимизацией кода связано множество заблуждений.

***Сокращение числа строк высокоуровневого кода повышает быстродействие или уменьшает объем итогового машинного кода — НЕВЕРНО!*** Многие убеждены в том, что, если сократить какой-то фрагмент до одной или двух строк, он будет максимально эффективным. Рассмотрим код инициализации массива из 10 элементов:

```
for i = 1 to 10
   a[ i ] = i
end for
```

Как вы думаете, он выполнится быстрее или медленнее, чем эти 10 строк, решающих ту же задачу?

```
a[ 1 ] = 1
a[ 2 ] = 2
a[ 3 ] = 3
a[ 4 ] = 4
a[ 5 ] = 5
a[ 6 ] = 6
a[ 7 ] = 7
a[ 8 ] = 8
a[ 9 ] = 9
a[ 10 ] = 10
```

Если вы придерживаетесь старой догмы «меньшее число строк выполняется быстрее», вы скажете, что первый фрагмент быстрее. Однако тесты на Microsoft Visual Basic и Java показали, что второй фрагмент минимум на 60% быстрее первого.

| Язык | Время выполнения цикла *for* | Время выполнения последовательного кода | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| Visual Basic | 8,47 | 3,16 | 63% | 2,5:1 |
| Java | 12,6 | 3,23 | 74% | 4:1 |

**Примечания:** (1) Временные показатели в этой и следующих таблицах данной главы указываются в секундах, а их сравнение имеет смысл только в пределах конкретных строк каждой из таблиц. Действительные показатели будут зависеть от компилятора, параметров компилятора и среды, в которой выполняется тестирование. (2) Большинство результатов сравнительного тестирования основано на выполнении фрагментов кода от нескольких тысяч до многих миллионов раз, что призвано устранить колебания результатов. (3) Конкретные марки и версии компиляторов не указываются. Показатели производительности во многом зависят от марки и версии компилятора. (4) Сравнение результатов тестирования фрагментов, написанных на разных языках, имеет смысл не всегда, так как компиляторы разных языков не всегда позволяют задать одинаковые параметры генерирования кода. (5) Фрагменты, написанные на интерпретируемых языках (PHP и Python), в большинстве случаев тестировались с использованием более чем в 100 раз меньшего числа тестов, чем фрагменты, написанные на других языках. (6) Некоторые из показателей «экономии времени» не совсем точны из-за округления «времени выполнения кода до оптимизации» и «времени выполнения оптимизированного кода».

Разумеется, это не значит, что увеличение числа строк высокоуровневого кода всегда приводит к повышению быстродействия или сокращению объема программы. Это означает, что независимо от эстетической привлекательности компактного кода ничего определенного о связи между числом строк кода на высокоуровневом языке и объемом и быстродействием итоговой программы сказать нельзя.

***Одни операции, вероятно, выполняются быстрее или компактнее других — НЕВЕРНО!*** Если речь идет о производительности, не может быть никаких «вероятно». Без измерения производительности вы никак не сможете точно узнать, помогли ваши изменения программе или навредили. Правила игры изменяются при каждом изменении языка, компилятора, версии компилятора, библиотек, версий библиотек, процессора, объема памяти, цвета рубашки, которую вы надели (ладно, это шутка), и т. д. Результаты, полученные на одном компьютере с одним набором инструментов, вполне могут оказаться противоположными на другом компьютере с другим набором инструментов.

Исходя из этого, можно назвать несколько причин, по которым производительность не следует повышать путем оптимизации кода. Если программа должна быть портируемой, помните: методики, повышающие производительность в одной среде, могут снижать ее в других. Если вы решите изменить или модернизировать компилятор, возможно, новый компилятор будет автоматически выполнять те виды оптимизации, что вы выполнили вручную, и все ваши усилия окажутся бесполезными. Хуже того: оптимизировав код, вы можете помешать компилятору выполнить более эффективные виды оптимизации, ориентированные на простой код.

Оптимизируя код, вы обрекаете себя на перепрофилирование каждого оптимизированного фрагмента при каждом изменении марки компилятора, его версии, версий библиотек и т. д. Если вы не будете перепрофилировать код, оптимизация, бывшая выгодной, после изменения среды сборки программы вполне может стать невыгодной.

***Оптимизацию следует выполнять по мере написания кода — НЕВЕРНО!*** Кое-кто утверждает, что если вы будете стремиться написать самый быстрый и компактный код при работе над каждым методом, то итоговая программа будет быстрой и компактной. Однако на самом деле это мешает увидеть за деревьями лес, и программисты, чрезмерно поглощенные микрооптимизацией, начинают упускать из виду по-настоящему важные глобальные виды оптимизации. Основные недостатки этого подхода рассмотрены ниже.

> Возможности небольшого повышения эффективности следует игнорировать, скажем, в 97% случаев: необдуманная оптимизация — корень всего зла.
>
> *Дональд Кнут (Donald Knuth)*

- До создания полностью работоспособной программы найти узкие места в коде почти невозможно. Программисты очень плохо угадывают, на какие 4% кода приходятся 50% времени выполнения, поэтому, оптимизируя код по мере его написания, они будут тратить примерно 96% времени на оптимизацию кода, который не нуждается в оптимизации. На оптимизацию по-настоящему важных 4% кода времени у них уже не останется.
- В тех редких случаях, когда узкие места определяются правильно, разработчики уделяют им слишком большое внимание, и критически важными становятся уже другие узкие места. Результат очевиден: все то же снижение произ-

водительности. Если оптимизация выполняется после создания полной системы, разработчики могут определить все проблемные области и их относительную важность, что способствует эффективному распределению времени.

- Концентрация на оптимизации во время первоначальной разработки отвлекает от достижения других целей. Разработчики погружаются в анализ алгоритмов и сокровенные дискуссии, от которых пользователям ни тепло, ни холодно. Корректность, сокрытие информации, удобочитаемость и т. д. становятся вторичными целями, хотя потом улучшить их сложнее, чем производительность. Работа над повышением производительности после создания полной программы обычно затрагивает менее 5% кода. Что легче: повысить производительность 5% кода или улучшить удобочитаемость всего кода?

Короче говоря, главный недостаток преждевременной оптимизации — отсутствие перспективы. Это сказывается на быстродействии итогового кода, других, еще более важных атрибутах производительности и качестве программы, ну а расплачиваться за это в итоге приходится пользователям. Если время, сэкономленное благодаря реализации наиболее простой программы, посвятить ее последующей оптимизации, итоговая программа непременно будет работать быстрее, чем программа, разработанная с использованием неорганизованного подхода к оптимизации (Stevens, 1981).

Иногда оптимизация программы после ее написания не позволяет достичь нужных показателей производительности, из-за чего приходится вносить крупные изменения в завершенный код. Можете утешить себя тем, что в этих случаях оптимизация небольших фрагментов все равно не привела бы к нужным результатам. Проблема в таких ситуациях объясняется не низким качеством кода, а неадекватной архитектурой программы.

Если оптимизацию нужно выполнять до создания полной программы, сведите риск к минимуму, интегрировав в процесс оптимизации перспективу. Один из способов сделать это — задать целевые показатели объема и быстродействия отдельных функций и провести оптимизацию кода по мере его написания, направленную на достижение этих показателей. Определив такие цели в спецификации, вы сможете следить сразу и за лесом, и за конкретными деревьями.

**Дополнительные сведения** Описания других занимательных и поучительных случаев можно найти в книге Джеральда Вайнберга «Psychology of Computer Programming» (Weinberg, 1998).

***Быстродействие программы не менее важно, чем ее корректность — НЕВЕРНО!*** Едва ли можно представить ситуацию, когда программу прежде всего нужно сделать быстрой или компактной и только потом корректной. Джеральд Вайнберг рассказывает историю о программисте, которого вызвали в Детройт, чтобы он помог отладить неработоспособную программу. Через несколько дней разработчики пришли к выводу, что ситуация безнадежна.

На пути домой он обдумывал проблему и внезапно понял ее суть. К концу полета у него уже был набросок нового кода. В течение нескольких дней программист тестировал код и уже собирался вернуться в Детройт, но тут получил телеграмму, в которой утверждалось, что работа над проектом прекращена из-за невозможности написания программы. И все же он снова прилетел в Детройт и убедил руководителей в том, что проект можно было довести до конца.

Далее он должен был убедить в этом участников проекта. Они выслушали его, и когда он закончил, создатель старой системы спросил:

— И как быстро выполняется ваша программа?

— Ну, в среднем она обрабатывает каждый набор введенных данных примерно за 10 секунд.

— Ага! Но моей программе для этого требуется только 1 секунда.

Ветеран откинулся назад, удовлетворенный тем, что он приструнил выскочку. Другие программисты, похоже, согласились с ним, но новичок не смутился.

— Да, но ваша программа *не работает*. Если бы моя не обязана была работать, я мог бы сделать так, чтобы она обрабатывала ввод почти мгновенно.

В некоторых проектах быстродействие или компактность кода действительно имеет большое значение. Однако таких проектов немного — гораздо меньше, чем кажется большинству людей, — и их число постоянно сокращается. В этих проектах проблемы с производительностью нужно решать путем предварительного проектирования. В остальных случаях ранняя оптимизация представляет серьезную угрозу для общего качества ПО, *включая производительность*.

## Когда выполнять оптимизацию?

Создайте высококачественный проект. Следите за правильностью программы. Сделайте ее модульной и изменяемой, чтобы позднее над ней было легко работать. Написав корректную программу, оцените ее производительность. Если программа громоздка, сделайте ее быстрой и компактной. Не оптимизируйте ее, пока не убедитесь, что это на самом деле нужно.

> Правила оптимизации Джексона: Правило 1. Не делайте этого. Правило 2 (только для экспертов). Не делайте этого пока — до тех пор, пока вы не получите совершенно ясное неоптимизированное решение.
>
> *М. А. Джексон*
> *(M. A. Jackson)*

Несколько лет назад я работал над программой на C++, которая должна была генерировать графики, помогающие анализировать данные об инвестициях. Написав код расчета первого графика, мы провели тестирование, показавшее, что программа отображает график примерно за 45 минут, что, конечно, было неприемлемо. Чтобы решить, что с этим делать, мы провели собрание группы. На собрании один из разработчиков выкрикнул в сердцах: «Если мы хотим иметь хоть какой-то шанс выпустить приемлемый продукт, мы должны начать переписывать весь код на ассемблере *прямо сейчас*». Я ответил, что мне так не кажется — что около 50% времени выполнения скорее всего приходятся на 4% кода. Было бы лучше исправить эти 4% в конце работы над проектом. После еще некоторых споров наш руководитель поручил мне поработать над производительностью программы (что мне и было нужно: «О, нет! Только не бросай меня в тот терновый куст!»[1]).

Как часто бывает, я очень быстро нашел в коде пару ослепительных узких мест. Внеся несколько изменений, я снизил время рисования с 45 минут до менее чем 30 секунд. Гораздо меньше 1% кода соответствовало 90% времени выполнения.

---

[1] Часто цитируемая фраза из негритянской сказки про Братца Кролика и Братца Волка в изложении У. Фолкнера. — *Прим. перев.*

Ну, а к моменту выпуска ПО нам удалось сократить время рисования почти до 1 секунды.

## Оптимизация кода компилятором

Современные компиляторы могут оптимизировать код куда эффективнее, чем вам кажется. В случае, который я описал выше, мой компилятор выполнил оптимизацию вложенного цикла так эффективно, что я едва ли получил бы лучшие результаты, переписав код. Покупая компилятор, сравните производительность каждого компилятора с использованием своей программы. Каждый компилятор имеет свои плюсы и минусы, и одни компиляторы лучше подходят для вашей программы, чем другие.

Оптимизирующие компиляторы лучше оптимизируют простой код. Если вы жонглируете индексами циклов и делаете другие «хитрые» вещи, компилятору будет труднее выполнить свою работу, от чего пострадает ваша программа. В подразделе «Размещение одного оператора на строке» раздела 31.5 вы найдете пример простого кода, который после оптимизации компилятором оказался на 11% быстрее, чем аналогичный «хитрый» код.

Хороший оптимизирующий компилятор может повысить быстродействие кода на 40 и более процентов, тогда как многие из методик, описанных в следующей главе, — только на 15–30%. Так почему ж просто не написать ясный код и не позволить компилятору выполнить свою работу? Вот результаты нескольких тестов, показывающие, насколько успешно компиляторы оптимизировали метод вставки-сортировки:

| Язык | Время выполнения кода без оптимизации | Время выполнения кода, оптимизированного компилятором | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| Компилятор C++ 1 | 2,21 | 1,05 | 52% | 2:1 |
| Компилятор C++ 2 | 2,78 | 1,15 | 59% | 2,5:1 |
| Компилятор C++ 3 | 2,43 | 1,25 | 49% | 2:1 |
| Компилятор C# | 1,55 | 1,55 | 0% | 1:1 |
| Visual Basic | 1,78 | 1,78 | 0% | 1:1 |
| Java VM 1 | 2,77 | 2,77 | 0% | 1:1 |
| Java VM 2 | 1,39 | 1,38 | <1% | 1:1 |
| Java VM 3 | 2,63 | 2,63 | 0% | 1:1 |

Единственное различие между версиями метода заключалось в том, что при первой компиляции оптимизация была отключена, а при второй включена. Очевидно, что одни компиляторы выполняют оптимизацию лучше, чем другие, а некоторые изначально генерируют максимально эффективный код без его оптимизации. Некоторые виртуальные машины Java (Java Virtual Machine, JVM) также более эффективны, чем другие. Эффективность вашего компилятора или вашей JVM может быть другой; оцените ее сами.

# 25.3. Где искать жир и патоку?

При оптимизации кода вы находите части программы, медленные, как патока зимой, и огромные, как Годзилла, и изменяете их так, чтобы они были быстры, как молния, и могли скрываться в расщелинах между байтами в оперативной памяти. Без профилирования программы вы никогда не сможете с уверенностью сказать, какие фрагменты медленны и огромны, но некоторые операции давно славятся ленью и ожирением, так что вы можете начать исследование именно с них.

## Частые причины снижения эффективности

***Операции ввода/вывода*** Один из самых главных источников неэффективности — ненужные операции ввода/вывода. Если объем используемой памяти не играет особой роли, работайте с данными в памяти, а не обращайтесь к диску, БД или сетевому ресурсу.

Вот результаты сравнения эффективности случайного доступа к элементам 100-элементного массива «в памяти» и записям аналогичного файла, хранящегося на диске:

| Язык | Время обработки внешнего файла | Время обработки данных «в памяти» | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| C++ | 6,04 | 0,000 | 100% | — |
| C# | 12,8 | 0,010 | 100% | 1000:1 |

Судя по этим результатам, доступ к данным «в памяти» выполняется в 1000 раз быстрее, чем доступ к данным, хранящимся во внешнем файле. В случае моего компилятора C++ время доступа к данным «в памяти» не удалось даже измерить.

Результаты аналогичного тестирования последовательного доступа к данным похожи:

| Язык | Время обработки внешнего файла | Время обработки данных «в памяти» | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| C++ | 3,29 | 0,021 | 99% | 150:1 |
| C# | 2,60 | 0,030 | 99% | 85:1 |

**Примечание:** при тестировании последовательного доступа данные были в 13 раз более объемными, чем при тестировании случайного доступа, поэтому результаты двух видов тестов сравнивать нельзя.

Если для доступа к внешним данным используется более медленная среда (например, сетевое соединение), разница только увеличивается. При тестировании случайного доступа к данным по сети результаты выглядят так:

| Язык | Время обработки локального файла | Время обработки файла по сети | Экономия времени |
|---|---|---|---|
| C++ | 6,04 | 6,64 | -10% |
| C# | 12,8 | 14,1 | -10% |

Конечно, эти результаты сильно зависят от скорости сети, объема трафика, расстояния между компьютером и сетевым диском, производительности сетевого и локального дисков, фазы Луны и других факторов.

В целом доступ к данным «в памяти» выполняется гораздо быстрее, так что дважды подумайте, прежде чем включать операции ввода/вывода в фрагменты, к быстродействию которых предъявляются повышенные требования.

***Замещение страниц*** Операция, заставляющая ОС заменять страницы памяти, выполняется гораздо медленнее, чем операция, ограниченная одной страницей памяти. Иногда самое простое изменение может принести огромную пользу. Например, один программист обнаружил, что в системе, использующей страницы объемом по 4 кб, следующий цикл инициализации вызывает массу страничных ошибок:

**Пример цикла инициализации, вызывающего много страничных ошибок (Java)**

```
for ( column = 0; column < MAX_COLUMNS; column++ ) {
   for ( row = 0; row < MAX_ROWS; row++ ) {
      table[ row ][ column ] = BlankTableElement();
   }
}
```

Это хорошо отформатированный цикл с удачными именами переменных, так в чем же проблема? Проблема в том, что каждая строка (row) массива *table* содержит около 4000 байт. Если массив включает слишком много строк, то при каждом обращении к новой строке ОС должна будет заменить страницы памяти. В предыдущем фрагменте изменение номера строки, а значит, и подкачка новой страницы с диска выполняются при каждой итерации внутреннего цикла.

Программист реорганизовал цикл:

**Пример цикла инициализации, вызывающего немного страничных ошибок (Java)**

```
for ( row = 0; row < MAX_ROWS; row++ ) {
   for ( column = 0; column < MAX_COLUMNS; column++ ) {
      table[ row ][ column ] = BlankTableElement();
   }
}
```

Этот код также вызывает страничную ошибку при каждом изменении номера строки, но это происходит только *MAX_ROWS* раз, а не *MAX_ROWS * MAX_COLUMNS* раз.

Степень снижения быстродействия кода из-за замещения страниц во многом зависит от объема памяти. На компьютере с небольшим объемом памяти второй фрагмент кода выполнялся примерно в 1000 раз быстрее, чем первый. При наличии большего объема памяти различие было всего лишь двукратным и было заметно лишь при очень больших значениях *MAX_ROWS* и *MAX_COLUMNS*.

***Системные вызовы*** Вызовы системных методов часто дороги. Они нередко включают переключение контекста — сохранение состояния программы, восстановление состояния ядра ОС и наоборот. В число системных методов входят методы, служащие для работы с диском, клавиатурой, монитором, принтером и другими

устройствами, методы управления памятью и некоторые вспомогательные методы. Если вас беспокоит производительность, узнайте, насколько дороги системные вызовы в вашей системе. Если они дороги, рассмотрите следующие варианты.

- Напишите собственные методы. Иногда функциональность системных методов оказывается избыточной для решения конкретных задач. Заменив низкоуровневые системные методы собственными, вы получите более быстрый и компактный код, лучше соответствующий вашим потребностям.
- Избегайте вызовов системных методов.
- Обратитесь к производителю системы и укажите ему на низкую эффективность тех или иных методов. Обычно производители хотят улучшить свою продукцию и охотно принимают все замечания (поначалу они могут показаться немного недовольными, но они на самом деле в этом заинтересованы).

В программе, про оптимизацию которой я рассказал в подразделе «Когда выполнять оптимизацию?» раздела 25.2, использовался класс *AppTime*, производный от коммерческого класса *BaseTime* (имена изменены). Объекты *AppTime* использовались в программе на каждом шагу и исчислялись десятками тысяч. Через несколько месяцев мы обнаружили, что объекты *BaseTime* инициализировались в конструкторе значением системного времени. В нашей программе системное время не играло никакой роли, а это означало, что мы без надобности генерировали тысячи системных вызовов. Простое переопределение конструктора класса *BaseTime* так, чтобы поле *time* инициализировалось нулем, дало нам такое же повышение производительности, что и все остальные изменения, вместе взятые.

***Интерпретируемые языки*** При выполнении интерпретируемого кода каждая команда должна быть обработана и преобразована в машинный код, поэтому интерпретируемые языки обычно гораздо медленней компилируемых. Вот примерные результаты сравнения разных языков, полученные мной при работе над этой главой и главой 26 (табл. 25-1):

**Табл. 25-1. Относительное быстродействие кода, написанного на разных языках**

| Язык | Тип языка | Время выполнения кода в сравнении с кодом C++ |
|---|---|---|
| C++ | Компилируемый | 1:1 |
| Visual Basic | Компилируемый | 1:1 |
| C# | Компилируемый | 1:1 |
| Java | Байт-код | 1,5:1 |
| PHP | Интерпретируемый | >100:1 |
| Python | Интерпретируемый | >100:1 |

Как видите, в плане быстродействия языки C++, Visual Basic и C# примерно одинаковы. Код на Java выполняется несколько медленнее. PHP и Python — интерпретируемые языки, и код, написанный на них, обычно выполняется в 100 и более раз медленнее, чем написанный на C++, Visual Basic, C# или Java. Однако к общим результатам, указанным в этой таблице, следует относиться с осторожностью. Относительная эффективность C++, Visual Basic, C#, Java и других языков во многом зависит от конкретного кода (читая главу 26, вы сами в этом убедитесь).

***Ошибки*** Наконец, еще одним источником проблем с производительностью являются некоторые виды ошибок. Какие? Вы можете оставить в итоговой версии программы отладочный код (например, записывающий трассировочную информацию в файл), забыть про освобождение памяти, неграмотно спроектировать таблицы БД, опрашивать несуществующие устройства до истечения лимита времени и т. д.

При работе над первой версией одного приложения мы столкнулись с операцией, выполнявшейся гораздо медленнее других похожих операций. Сделав массу попыток объяснить этот факт, мы выпустили версию 1.0, так и не поняв полностью, в чем дело. Однако, работая над версией 1.1, я обнаружил, что таблица БД, используемая в этой операции, не была проиндексирована! Простая индексация таблицы повысила скорость некоторых операций в 30 раз. Определение индекса для часто используемой таблицы нельзя считать оптимизацией — это просто хорошая практика программирования.

## Относительное быстродействие распространенных операций

Хотя нельзя с полной уверенностью утверждать, что одни операции медленнее других, не оценив их, определенные операции все же обычно дороже. Отыскивая патоку в своей программе, используйте табл. 25-2, которая поможет вам выдвинуть первоначальные предположения о том, какие фрагменты кода неэффективны.

**Табл. 25-2. Быстрота выполнения часто используемых операций**

| | | Относительное время выполнения | |
|---|---|---|---|
| **Операция** | **Пример** | **C++** | **Java** |
| *Исходный показатель (целочисленное присваивание)* | *i = j* | 1 | 1 |
| **Вызовы методов** | | | |
| Вызов метода без параметров | *foo()* | 1 | — |
| Вызов закрытого метода без параметров | *this.foo()* | 1 | 0,5 |
| Вызов закрытого метода с одним параметром | *this.foo( i )* | 1,5 | 0,5 |
| Вызов закрытого метода с двумя параметрами | *this.foo( i, j )* | 2 | 0,5 |
| Вызов метода объекта | *bar.foo()* | 2 | 1 |
| Вызов метода производного объекта | *derivedBar.foo()* | 2 | 1 |
| Вызов полиморфного метода | *abstractBar.foo()* | 2,5 | 2 |
| **Обращения к объектам** | | | |
| Обращение к объекту 1-го уровня | *i = obj.num* | 1 | 1 |
| Обращение к объекту 2-го уровня | *i = obj1.obj2. num* | 1 | 1 |
| Стоимость каждого дополнительного уровня | *i = obj1.obj2.obj3…* | неизмеряема | неизмеряема |

**Табл. 25-2.** (*продолжение*)

| | | Относительное время выполнения | |
|---|---|---|---|
| **Операция** | **Пример** | **C++** | **Java** |
| **Операции над целочисленными переменными** | | | |
| Целочисленное присваивание (локальная операция) | *i = j* | 1 | 1 |
| Целочисленное присваивание (унаследованная операция) | *i = j* | 1 | 1 |
| Сложение | *i = j + k* | 1 | 1 |
| Вычитание | *i = j - k* | 1 | 1 |
| Умножение | *i = j * k* | 1 | 1 |
| Деление | *i = j \ k* | 5 | 1,5 |
| **Операции над переменными с плавающей запятой** | | | |
| Присваивание | *x = y* | 1 | 1 |
| Сложение | *x = y + z* | 1 | 1 |
| Вычитание | *x = y - z* | 1 | 1 |
| Умножение | *x = y * z* | 1 | 1 |
| Деление | *x = y / z* | 4 | 1 |
| **Трансцендентные функции** | | | |
| Извлечение квадратного корня из числа с плавающей запятой | *x = sqrt( y )* | 15 | 4 |
| Вычисление синуса числа с плавающей запятой | *x = sin( y )* | 25 | 20 |
| Вычисление логарифма числа с плавающей запятой | *x = log( y )* | 25 | 20 |
| Вычисление экспоненты числа с плавающей запятой | *x = exp( y )* | 50 | 20 |
| **Операции над массивами** | | | |
| Обращение к массиву целых чисел с использованием константы | *i = a[ 5 ]* | 1 | 1 |
| Обращение к массиву целых чисел с использованием переменной | *i = a[ j ]* | 1 | 1 |
| Обращение к двумерному массиву целых чисел с использованием констант | *i = a[ 3, 5 ]* | 1 | 1 |
| Обращение к двумерному массиву целых чисел с использованием переменных | *i = a[ j, k ]* | 1 | 1 |
| Обращение к массиву чисел с плавающей запятой с использованием константы | *x = z[ 5 ]* | 1 | 1 |
| Обращение к массиву чисел с плавающей запятой с использованием целочисленной переменной | *x = z[ j ]* | 1 | 1 |

(*см. след. стр.*)

**Табл. 25-2.** *(окончание)*

| | | Относительное время выполнения | |
|---|---|---|---|
| **Операция** | **Пример** | **C++** | **Java** |
| Обращение к двумерному массиву чисел с плавающей запятой с использованием констант | *x = z[ 3, 5 ]* | 1 | 1 |
| Обращение к двумерному массиву чисел с плавающей запятой с использованием целочисленных переменных | *x = z[ j, k ]* | 1 | 1 |

**Примечание:** показатели, приведенные здесь, сильно зависят от локальной среды, компилятора и выполняемых компилятором видов оптимизации. Результаты, указанные для языков C++ и Java, нельзя сравнивать непосредственно.

С момента выхода первого издания этой книги относительное быстродействие отмеченных операций значительно изменилось, так что, если вы все еще подходите к оптимизации кода, опираясь на идеи 10-летней давности, пересмотрите свои взгляды.

Большинство частых операций — в том числе вызовы методов, присваивание, арифметические операции над целыми числами и числами с плавающей запятой — имеет примерно одинаковую цену. Трансцендентные математические функции очень дороги. Вызовы полиморфных методов чуть дороже вызовов других методов.

Табл. 25-2 или похожая таблица, которую вы можете создать сами, — ключ, открывающий все двери в мир быстрого кода, описанные в главе 26. В каждом случае повышение быстродействия исходит из замены дорогой операции на более дешевую (см. главу 26).

## 25.4. Оценка производительности

На небольшие фрагменты программы обычно приходится непропорционально большая доля времени ее выполнения, поэтому перед оптимизацией кода вам следует оценить его и найти в нем горячие точки. Обнаружив горячие точки и оптимизировав их, снова оцените код, чтобы узнать, насколько вы его улучшили. Многие аспекты производительности противоречат интуиции. Выше я уже привел один пример этого, когда 10 строк кода оказались в итоге значительно быстрее и компактнее, чем одна строка.

Опыт также не особо полезен при оптимизации. Опыт может быть основан на использовании старого компьютера, языка или компилятора, но когда что-либо из этого изменяется, все начинается сначала. Невозможно точно сказать, каковы результаты оптимизации, не оценив их.

Много лет назад я написал программу, суммирующую элементы матрицы. Первоначальный код выглядел примерно так:

### Пример простого кода, суммирующего элементы матрицы (C++)

```
sum = 0;
for ( row = 0; row < rowCount; row++ ) {
   for ( column = 0; column < columnCount; column++ ) {
      sum = sum + matrix[ row ][ column ];
   }
}
```

**Дополнительные сведения** Джон Бентли описывает похожий случай, когда переписывание кода с использованием указателей снизило производительность примерно на 10%. В другой ситуации этот же подход повысил производительность более чем на 50%. См. «Software Exploratorium: Writing Efficient C Programs» (Bentley, 1991).

Как видите, код был прост, но суммирование элементов матрицы должно было выполняться как можно быстрее, а я знал, что все обращения к массиву и проверки условий цикла довольно дороги. Я знал, что при каждом обращении к двумерному массиву выполняются дорогие операции умножения и сложения. Так, обработка матрицы размером 100 на 100 требовала 10 000 умножений и сложений, что дополнялось еще и затратами, связанными с управлением циклами. Использовав указатели, рассудил я, я смогу просто увеличивать указатель, заменив 10 000 дорогих умножений на 10 000 относительно дешевых операций инкремента. Я тщательно преобразовал код и получил:

### Пример попытки оптимизации кода, суммирующего элементы матрицы (C++)

```
sum = 0;
elementPointer = matrix;
lastElementPointer = matrix[ rowCount - 1 ][ columnCount - 1 ] + 1;
while ( elementPointer < lastElementPointer ) {
   sum = sum + *elementPointer++;
}
```

Хотя код стал менее удобочитаемым, особенно для программистов, не являющихся экспертами в C++, я был очень доволен собой. Оно и понятно: все-таки я избавился от 10 000 умножений и многих операций, связанных с управлением циклами! Я был так доволен, что решил подкрепить свои чувства конкретными цифрами и оценить повышение скорости, хотя в то время я выполнял это не всегда.

Ни один программист *никогда* не мог предсказать или обнаружить узкие места, *не обладая данными*. Что бы вы ни думали, реальность окажется совершенно другой.

*Джозеф М. Ньюкамер (Joseph M. Newcomer)*

Знаете, что я обнаружил? Никакого улучшения. Ни для матриц размером 100 на 100. Ни для матриц размером 10 на 10. Ни для каких-либо других матриц. Я был так разочарован, что погрузился в ассемблерный код, сгенерированный компилятором, чтобы понять, почему моя оптимизация не сработала. К моему удивлению, оказалось, что я был не первым, кому понадобилось перебирать элементы массива: компилятор сам преобразовывал обращения к массиву в операции над указателями. Я понял, что единственным результатом оптимизации, в котором можно быть полностью уверенным без измерения производительности, является затруднение чтения кода. Если оценка эффективности не оправдывает себя, не стоит приносить понятность кода в жертву сомнительному повышению производительности.

### Оценка должна быть точной

**Перекрестная ссылка** Об инструментах профилирования см. подраздел «Оптимизация кода» раздела 30.3.

Оценка производительности должна быть точной. Измерение времени выполнения программы с помощью секундомера или путем подсчета «один слон, два слона, три слона» точным не является. Используйте инструменты профилирования или системный таймер и методы, регистрирующие истекшее время выполнения операций.

Используете ли вы инструмент, написанный другим программистом, или пишете для оценки производительности программы собственный код, убедитесь, что измеряете время выполнения только того оптимизируемого кода. Опирайтесь на число тактов процессора, выделенных вашей программе, а не на время суток. Иначе при переключении системы с вашей программы на другую программу один из ваших методов будет оштрафован на время, выделенное другой программе. Кроме того, попытайтесь исключить влияние процесса оценки кода и запуска программы на первоначальный и оптимизированный код.

## 25.5. Итерация

Обнаружив в коде узкое место и попробовав его устранить, вы удивитесь, насколько можно повысить производительность кода путем его оптимизации. Единственная методика редко приводит к десятикратному улучшению, но методики можно эффективно комбинировать, поэтому даже после обнаружения одного удачного вида оптимизации продолжайте пробовать другие виды.

Однажды я написал программную реализацию алгоритма Data Encryption Standard (DES). Ну, на самом деле я писал ее не один раз, а около тридцати. При шифровании по алгоритму DES цифровые данные кодируются так, что их нельзя расшифровать без правильного пароля. Этот алгоритм шифрования так хитер, что иногда кажется, что он сам зашифрован. Моя цель состояла в том, чтобы файл объемом 18 кб шифровался на IBM PC за 37 секунд. Первая реализация алгоритма выполнялась 21 минуту 40 секунд, так что мне предстояла долгая работа.

Хотя большинство отдельных видов оптимизации было незначительно, в сумме они привели к впечатляющим результатам. Никакие три или даже четыре вида оптимизации не позволили бы мне достичь цели, однако итоговая их комбинация оказалась эффективной. Мораль: если копать достаточно глубоко, можно добиться подчас неожиданных результатов.

**Перекрестная ссылка** Методики, указанные в этой таблице, обсуждаются в главе 26.

Оптимизация алгоритма DES — самая агрессивная оптимизация, которую я когда-либо проделывал. В то же время я никогда не создавал более непонятного и трудного в сопровождении кода. Первоначальный алгоритм был сложен. Код, получившийся в результате трансформаций высокоуровневого кода, оказался практически нечитаемым. После преобразования кода на ассемблер я получил один метод из 500 строк, на который боюсь даже смотреть. Это отношение между оптимизацией кода и его качеством справедливо почти всегда. Вот таблица, отражающая историю оптимизации:

| Вид оптимизации | Время выполнения | Улучшение |
|---|---|---|
| Первоначальная реализация | 21:40 | |
| Преобразование битовых полей в массивы | 7:30 | 65% |
| Развертывание самого внутреннего цикла *for* | 6:00 | 20% |
| Удаление перестановок | 5:24 | 10% |
| Объединение двух переменных | 5:06 | 5% |
| Использование логического тождества для объединения первых двух этапов алгоритма DES | 4:30 | 12% |
| Объединение областей памяти, используемых двумя переменными, для сокращения числа операций над данными во внутреннем цикле | 3:36 | 20% |
| Объединение областей памяти, используемых двумя переменными, для сокращения числа операций над данными во внешнем цикле | 3:09 | 13% |
| Развертывание всех циклов и использование литералов для индексации массива | 1:36 | 49% |
| Удаление вызовов методов и встраивание всего кода | 0:45 | 53% |
| Переписывание всего метода на ассемблере | 0:22 | 51% |
| Итог | 0:22 | 98% |

**Примечание:** постепенный процесс оптимизации, описанный в этой таблице, не подразумевает, что все виды оптимизации эффективны. Я мог бы указать массу других видов, приводивших к удвоению времени выполнения. Минимум две трети видов оптимизации, которые я попробовал, оказались неэффективными.

## 25.6. Подход к оптимизации кода: резюме

Рассматривая целесообразность оптимизации кода, придерживайтесь следующего алгоритма:

1. Напишите хороший и понятный код, поддающийся легкому изменению.
2. Если производительность вас не устраивает:
   a. сохраните работоспособную версию кода, чтобы позднее вы могли вернуться к «последнему нормальному состоянию»;
   b. оцените производительность системы с целью нахождения горячих точек;
   c. узнайте, обусловлено ли плохое быстродействие неадекватным проектом, неверными типами данных или неудачными алгоритмами и определите, уместна ли оптимизация кода; если оптимизация кода неуместна, вернитесь к п. 1;
   d. оптимизируйте узкое место, определенное на этапе (c);
   e. оцените каждое улучшение по одному за раз;
   f. если оптимизация не привела к улучшению кода, вернитесь к коду, сохраненному на этапе (a) (как правило, более чем в половине случаев попытки оптимизации будут приводить лишь к незначительному повышению производительности или к ее снижению).
3. Повторите процесс, начиная с п. 2.

# Дополнительные ресурсы

http://cc2e.com/2585

В этом разделе я указал работы, посвященные повышению производительности в общем. Книги, в которых обсуждаются специфические методики оптимизации кода, указаны в разделе «Дополнительные ресурсы» в конце главы 26.

## Производительность

http://cc2e.com/2592

Smith, Connie U. and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Boston, MA: Addison-Wesley, 2002. В этой книге обсуждается создание высокопроизводительного ПО, предусматривающее обеспечение нужной производительности на всех этапах разработки. В ней вы найдете много примеров и конкретных случаев, относящихся к программам нескольких типов, а также конкретные рекомендации по повышению производительности Web-приложений. Особое внимание в книге уделяется масштабируемости программ.

http://cc2e.com/2599

Newcomer, Joseph M. *Optimization: Your Worst Enemy*. May 2000, *www.flounder.com/optimization.htm*. В этой статье, принадлежащей перу опытного системного программиста, описываются разные ловушки, в которые вы можете попасть, используя неэффективные стратегии оптимизации.

## Алгоритмы и типы данных

Knuth, Donald. *The Art of Computer Programming*, vol. 1, *Fundamental Algorithms*, 3d ed. Reading, MA: Addison-Wesley, 1997.

Knuth, Donald. *The Art of Computer Programming*, vol. 2, *Seminumerical Algorithms*, 3d ed. Reading, MA: Addison-Wesley, 1997.

Knuth, Donald. *The Art of Computer Programming*, vol. 3, *Sorting and Searching*, 2d ed. Reading, MA: Addison-Wesley, 1998.

Это три первых тома серии, которая по первоначальному замыслу автора должна включить семь томов. В этих несколько пугающих книгах алгоритмы описываются не только на обычном языке, но и с использованием математической нотации, или MIX — языка ассемблера для воображаемого компьютера MIX. Кнут подробнейшим образом описывает огромное число вопросов, и, если вы испытываете сильный интерес к конкретному алгоритму, лучшего ресурса вам не найти.

Sedgewick, Robert. *Algorithms in Java, Parts 1-4*, 3d ed. Boston, MA: Addison-Wesley, 2002. В четырех частях этой книги исследуются лучшие методы решения широкого диапазона проблем. В число тем книги входят фундаментальные сведения, сортировка, поиск, реализация абстрактных типов данных и более сложные вопросы. В книге Седжвика *Algorithms in Java, Part 5*, 3d ed. (Sedgewick, 2003) обсуждаются алгоритмы, основанные на графах. Книги *Algorithms in C++, Parts 1-4*, 3d ed. (Sedgewick, 1998), *Algorithms in C++, Part 5*, 3d ed. (Sedgewick, 2002), *Algorithms in C, Parts 1-4*, 3d ed. (Sedgewick, 1997) и *Algorithms in C, Part 5*, 3d ed. (Sedgewick, 2001) организованы похожим образом. Седжвик имеет степень доктора философии и в свое время был учеником Кнута.

### Контрольный список: стратегии оптимизации кода

http://cc2e.com/2506

**Производительность программы в общем**

- ❑ Рассмотрели ли вы возможность повышения производительности посредством изменения требований к программе?
- ❑ Рассмотрели ли вы возможность повышения производительности путем изменения проекта программы?
- ❑ Рассмотрели ли вы возможность повышения производительности путем изменения проектов классов?
- ❑ Рассмотрели ли вы возможность повышения производительности путем сокращения объема взаимодействия с ОС?
- ❑ Рассмотрели ли вы возможность повышения производительности путем устранения операций ввода/вывода?
- ❑ Рассмотрели ли вы возможность повышения производительности путем использования компилируемого языка вместо интерпретируемого?
- ❑ Рассмотрели ли вы возможность повышения производительности путем видов оптимизации, поддерживаемых компилятором?
- ❑ Рассмотрели ли вы возможность повышения производительности путем перехода на другое оборудование?
- ❑ Рассматриваете ли вы оптимизацию кода только как последнее средство?

**Подход к оптимизации кода**

- ❑ Убедились ли вы в полной корректности программы перед началом оптимизации кода?
- ❑ Нашли ли вы узкие места в коде перед началом его оптимизации?
- ❑ Оцениваете ли вы результаты выполнения каждого вида оптимизации кода?
- ❑ Отменяете ли вы изменения, которые не привели к ожидаемому улучшению?
- ❑ Пробуете ли вы несколько способов оптимизации каждого узкого места, т. е. используете ли вы итерацию?

# Ключевые моменты

- Производительность — всего лишь один из аспектов общего качества ПО, и, как правило, не самый важный. Оптимизация кода — лишь один из способов повышения производительности ПО, и тоже обычно не самый важный. Быстродействие программы и ее объем обычно в большей степени зависят не от эффективности кода, а от архитектуры программы, детального проектирования выбора структур данных и алгоритмов.
- Важнейшее условие максимизации быстродействия кода — его количественная оценка. Она необходима для обнаружения областей, производительность которых действительно нуждается в повышении, а также для проверки того, что в результате оптимизации производительность повысилась, а не понизилась.

- Как правило, основная часть времени выполнения программы приходится на небольшую часть кода. Не выполнив оценку, вы не найдете этот код.
- Достижение желаемого повышения производительности кода при помощи его оптимизации обычно требует нескольких итераций.
- Во время первоначального кодирования нет лучше способа подготовки к повышению производительности программы, чем написание ясного и понятного кода, поддающегося легкому изменению.

ГЛАВА 26

# Методики оптимизации кода

**Содержание**



http://cc2e.com/2665

**Связанные темы**

- Стратегии оптимизации кода: глава 25
- Рефакторинг: глава 24

Оптимизация кода уже давно привлекает пристальное внимание программистов. Если вы решили повысить производительность и хотите сделать это на уровне кода (с учетом предупреждений, описанных в главе 25), то можете использовать целый ряд методик.

Эта глава посвящена в первую очередь повышению быстродействия, но включает и несколько советов по сокращению объема кода. Производительность обычно охватывает и быстродействие, и объем кода, но при необходимости сокращения объема кода обычно лучше прибегнуть к перепроектированию классов и данных, а не к оптимизации кода. Последняя подразумевает небольшие изменения, а не изменения более крупномасштабных аспектов проектирования.

В этой главе вы почти не найдете методик, настолько общих, чтобы код примеров можно было копировать прямо в другие программы. Я просто хочу проиллюстрировать ряд видов оптимизации кода, которые вы сможете приспособить к своей ситуации.

Виды оптимизации, описываемые в этой главе, могут показаться похожими на виды рефакторинга из главы 24, однако помните, что рефакторинг направлен на улуч-

шение внутренней структуры программы (Fowler, 1999). То, о чем мы будем говорить в этой главе, возможно, лучше называть «антирефакторингом». Эти изменения ухудшают внутреннюю структуру программы ради повышения ее производительности. Это верно по определению. Если бы изменения не ухудшали внутреннюю структуру, мы не считали бы их видами оптимизации — мы использовали бы их по умолчанию и считали стандартными методиками кодирования.

**Перекрестная ссылка** Оптимизация кода основана на эвристике (см. раздел 5.3).

Некоторые авторы характеризуют методики оптимизации кода как «практические правила» или приводят данные, говорящие о том, что определенный вид оптимизации непременно обеспечит желательный результат. Однако, как вы скоро увидите, концепция «практических правил» плохо описывает оптимизацию кода. Единственным надежным практическим правилом является оценка результатов каждого вида оптимизации в конкретной среде. Так что в этой главе представлен каталог «вещей, которые стоит попробовать»: многие из них в вашей среде пользы не принесут, но некоторые на самом деле окажутся очень эффективными.

# 26.1. Логика

**Перекрестная ссылка** О других аспектах использования операторов, определяющих логику программы, см. главы 14–19.

Многие задачи программирования связаны с манипулированием логикой программы. В этом разделе мы рассмотрим эффективное использование логических выражений.

## Прекращение проверки сразу же после получения ответа

Допустим, у вас есть выражение:

```
if ( 5 < x ) and ( x < 10 ) then ...
```

Как только вы определили, что *x* больше 5, вторую часть проверки выполнять не нужно.

**Перекрестная ссылка** О сокращенной оценке логических выражений см. также подраздел «Понимание правил вычисления логических выражений» раздела 19.1.

Некоторые языки поддерживают так называемую «сокращенную оценку выражений», при которой компилятор генерирует код, автоматически прекращающий проверку после получения ответа. Сокращенная оценка выполняется, например, для стандартных операторов C++ и «условных» операторов Java.

Если ваш язык не поддерживает сокращенную оценку, избегайте операторов *and* и *or*, используя вместо них дополнительную логику. Для сокращенной оценки наш код следовало бы изменить так:

```
if ( 5 < x ) then
   if ( x < 10 ) then ...
```

Принцип прекращения проверки сразу по получении ответа уместен и в других случаях. В качестве примера можно привести цикл поиска. Если вы сканируете массив введенных чисел и вам нужно только узнать, присутствует ли в массиве отрицательное значение, вы могли бы проверять значения по очереди, устанав-

ливая при обнаружении отрицательного числа флаг *negativeFound*. Вот как выглядел бы такой цикл поиска:

**Пример, в котором цикл продолжает выполняться даже после получения ответа (C++)**

```
negativeInputFound = false;
for ( i = 0; i < count; i++ ) {
   if ( input[ i ] < 0 ) {
      negativeInputFound = true;
   }
}
```

Лучше было бы прекращать просмотр массива сразу по обнаружении отрицательного значения. Любой из следующих советов привел бы к решению проблемы.

- Включите в код оператор *break* после строки *negativeInputFound = true*.
- Если язык не поддерживает оператор *break*, имитируйте его при помощи оператора *goto*, передающего управление первой команде, расположенной после цикла.
- Измените цикл *for* на цикл *while* и проверяйте значение *negativeInputFound* вместе с проверкой того, не превысил ли счетчик цикла значение *count*.
- Измените цикл *for* на цикл *while*, поместите сигнальное значение в первый элемент массива, расположенный после последнего исходного значения, а в условии цикла *while* просто проверяйте, не отрицательно ли значение. По завершении цикла узнайте, относится ли индекс обнаруженного отрицательного значения к исходному массиву или превышает на 1 индекс верхней границы массива. Подробнее о сигнальных значениях см. ниже.

Вот результаты использования ключевого слова *break* в коде C++ и Java:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени |
|---|---|---|---|
| C++ | 4,27 | 3,68 | 14% |
| Java | 4,85 | 3,46 | 29% |

**Примечания:** (1) Временные показатели в этой и следующих таблицах данной главы указываются в секундах, а их сравнение имеет смысл только в пределах конкретных строк каждой из таблиц. Действительные показатели будут зависеть от компилятора, параметров компилятора и среды, в которой выполняется тестирование. (2) Большинство результатов сравнительного тестирования основано на выполнении фрагментов кода от нескольких тысяч до многих миллионов раз, что призвано устранить колебания результатов. (3) Конкретные марки и версии компиляторов не указываются. Показатели производительности во многом зависят от марки и версии компилятора. (4) Сравнение результатов тестирования фрагментов, написанных на разных языках, имеет смысл не всегда, поскольку компиляторы разных языков не всегда позволяют задать одинаковые параметры генерирования кода. (5) Фрагменты, написанные на интерпретируемых языках (PHP и Python), в большинстве случаев тестировались с использованием более чем в 100 раз меньшего числа тестов, чем фрагменты, написанные на других языках. (6) Некоторые из показателей «экономии времени» не совсем точны из-за округления «времени выполнения кода до оптимизации» и «времени выполнения оптимизированного кода».

Результаты этого вида оптимизации во многом зависят от числа проверяемых значений и вероятности обнаружения отрицательного значения. В данном тесте число значений в среднем было равным 100, а отрицательные значения составляли половину всех значений.

## Упорядочение тестов по частоте

Упорядочивайте тесты так, чтобы самый быстрый и чаще всего оказывающийся истинным тест выполнялся первым. Нормальные случаи следует обрабатывать первыми, а вероятность выполнения неэффективного кода должна быть низкой. Этот принцип относится к блокам *case* и цепочкам операторов *if-then-else*.

Рассмотрим, например, оператор *Select-Case*, обрабатывающий символы, вводимые с клавиатуры:

**Пример плохо упорядоченного логического теста (Visual Basic)**

```
Select inputCharacter
   Case "+", "="
      ProcessMathSymbol( inputCharacter )
   Case "0" To "9"
      ProcessDigit( inputCharacter )
   Case ",", ".", ":", ";", "!", "?"
      ProcessPunctuation( inputCharacter )
   Case " "
      ProcessSpace( inputCharacter )
   Case "A" To "Z", "a" To "z"
      ProcessAlpha( inputCharacter )
   Case Else
      ProcessError( inputCharacter )
End Select
```

Порядок обработки символов в этом фрагменте близок к порядку сортировки ASCII. Однако блоки *case* во многом похожи на большой набор операторов *if-then-else*, так что если первым введенным символом будет «*a*», данный фрагмент проверит, является ли символ математическим символом, числом, знаком пунктуации или пробелом, и только потом определит, что это алфавитно-цифровой символ. Зная примерную вероятность ввода тех или иных символов, вы можете разместить самые вероятные случаи первыми. Вот переупорядоченные блоки *case*:

**Пример хорошо упорядоченного логического теста (Visual Basic)**

```
Select inputCharacter
   Case "A" To "Z", "a" To "z"
      ProcessAlpha( inputCharacter )
   Case " "
      ProcessSpace( inputCharacter )
   Case ",", ".", ":", ";", "!", "?"
      ProcessPunctuation( inputCharacter )
   Case "0" To "9"
      ProcessDigit( inputCharacter )
   Case "+", "="
```

```
      ProcessMathSymbol( inputCharacter )
   Case Else
      ProcessError( inputCharacter )
End Select
```

Теперь наиболее вероятные символы обрабатываются первыми, что снижает общее число выполняемых тестов. При типичной смеси вводимых символов результаты этого вида оптимизации таковы:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени |
|---|---|---|---|
| C# | 0,220 | 0,260 | -18% |
| Java | 2,56 | 2,56 | 0% |
| Visual Basic | 0,280 | 0,260 | 7% |

**Примечание:** тестирование выполнено для ввода, включавшего 78% алфавитых символов, 17% пробелов и 5% знаков пунктуации.

С Visual Basic все ясно, а вот результаты тестирования кода Java и C# довольно неожиданны. Очевидно, это объясняется способом структурирования операторов *switch-case* в языках C# и Java: из-за необходимости перечисления всех значений по отдельности, а не в форме диапазонов, код C# и Java не выигрывает от этого вида оптимизации в отличие от кода Visual Basic. Это доказывает, что никакой из видов оптимизации не следует применять слепо: результаты будут во многом зависеть от реализации конкретных компиляторов.

Вы могли бы предположить, что для аналогичного набора операторов *if-then-else* компилятор Visual Basic сгенерирует похожий код. Взгляните на результаты:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени |
|---|---|---|---|
| C# | 0,630 | 0,330 | 48% |
| Java | 0,922 | 0,460 | 50% |
| Visual Basic | 1,36 | 1,00 | 26% |

Совершенно иная картина. Те же тесты на Visual Basic теперь выполняются медленнее в пять раз без оптимизации и в четыре — в случае оптимизированного кода. Это говорит о том, что для блоков *case* и операторов *if-then-else* компилятор генерирует разный код.

Результаты оптимизации операторов *if-then-else* более согласованны, но общая ситуация от этого не проясняется. Обе версии кода C# и Visual Basic, основанного на блоках *case*, выполняются быстрее, чем обе версии кода, написанного на основе *if-then-else*, тогда как в случае Java все наоборот. Это различие результатов наводит на мысль о третьем виде оптимизации, описанном чуть ниже.

## Сравнение быстродействия похожих структур логики

Описанное выше тестирование можно выполнить и для блоков *case*, и для операторов *if-then-else*. В зависимости от среды любой из подходов может оказаться более

выгодным. Ниже данные из двух предыдущих таблиц представлены в форме, облегчающей сравнение быстродействия оптимизированного кода, написанного с применением обоих подходов:

| Язык | *case* | *if-then-else* | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| C# | 0,260 | 0,330 | –27% | 1:1 |
| Java | 2,56 | 0,460 | 82% | 6:1 |
| Visual Basic | 0,260 | 1,00 | 258% | 1:4 |

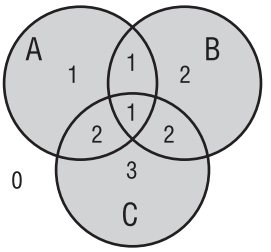
Эти результаты не имеют логического объяснения. В одном из языков *case* гораздо лучше, чем *if-then-else*, а в другом наоборот. В третьем языке различие относительно невелико. Можно было бы предположить, что из-за похожего синтаксиса *case* в C# и Java результаты тестирования этих языков также будут похожими, но на самом деле имеет место обратное.

Этот пример ясно показывает, что оптимизация кода не подчиняется ни «практическим правилам», ни «логике». Так что без *оценки* результатов вам не обойтись.

## Замена сложных выражений на обращение к таблице

**Перекрестная ссылка** Об использовании таблиц вместо сложной логики см. главу 18.

Иногда просмотр таблицы может оказаться более быстрым, чем выполнение сложной логической цепи. Суть сложной цепи обычно сводится к категоризации чего-то и выполнении того или иного действия, основанного на конкретной категории. Допустим, вы хотите присвоить чему-то номер категории на основе принадлежности этого чего-то к группам *A*, *B* и *C*:




Вот как эта задача решается при помощи сложной логической цепи:

**Пример сложной логической цепи (C++)**

```
if ( ( a && !c ) || ( a && b && c ) ) {
   category = 1;
}
else if ( ( b && !a ) || ( a && c && !b ) ) {
   category = 2;
}
else if ( c && !a && !b ) {
   category = 3;
}
else {
```

```
   category = 0;
}
```

Вместо этого теста вы можете использовать более модифицируемый и быстродействующий подход, основанный на просмотре табличных данных:

**Пример использования таблицы вместо сложной логики (C++)**

```
// Определение таблицы categoryTable.

Определение таблицы понять нелегко, поэтому используйте любые комментарии, способные помочь.

static int categoryTable[ 2 ][ 2 ][ 2 ] = {
   // !b!c  !bc  b!c  bc
       0,    3,    2,    2,    //   !a
       1,    2,    1,    1    //    a
};
...
category = categoryTable[ a ][ b ][ c ];
```

Определение этой таблицы кажется запутанным, но, если она хорошо документирована, читать ее будет не труднее, чем код сложной логической цепи. Кроме того, изменить таблицу будет гораздо легче, чем более раннюю логику. Вот результаты сравнения быстродействия обоих подходов:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| C++ | 5,04 | 3,39 | 33% | 1,5:1 |
| Visual Basic | 5,21 | 2,60 | 50% | 2:1 |

## Отложенные вычисления

Один из моих бывших соседей любил все откладывать на потом. В оправдание своей лени он говорил, что многое из того, что люди порываются сделать, делать просто не нужно. Если подождать достаточно долго, утверждал он, неважные дела канут в Лету, и он не будет тратить на них свое время.

Методика отложенных вычислений основана на принципе моего соседа: программа делает что-то, только когда это действительно нужно. Отложенное вычисление похоже на стратегию решения задач «по требованию», при которой работа выполняется максимально близко к тому месту, где нужны ее результаты.

Допустим, ваша программа работает с таблицей из 5000 значений, полностью генерируя ее при запуске и затем обращайтесь к ней по мере выполнения. Если программа использует только небольшую часть элементов таблицы, возможно, есть смысл вычислять их по мере надобности, а не все сразу. После вычисления элемента его можно сохранить на будущее (это называется «кэшированием»).

# 26.2. Циклы

**Перекрестная ссылка** О циклах см. также главу 16.

Так как циклы выполняются многократно, горячие точки часто следует искать именно внутри циклов. Методики, описываемые в этом разделе, помогают ускорить выполнение циклов.

## Размыкание цикла

Замыканием (switching) цикла называют принятие решения внутри цикла при каждой его итерации. Если во время выполнения цикла решение не изменяется, вы можете разомкнуть (unswitch) цикл, приняв решение вне цикла. Обычно для этого нужно вывернуть цикл наизнанку, т. е. поместить циклы в условный оператор, а не условный оператор внутрь цикла. Вот пример цикла до размыкания:

**Пример замкнутого цикла (C++)**

```
for ( i = 0; i < count; i++ ) {
   if ( sumType == SUMTYPE_NET ) {
      netSum = netSum + amount[ i ];
   }
   else {
      grossSum = grossSum + amount[ i ];
   }
}
```

В этом фрагменте проверка *if ( sumType == SUMTYPE_NET )* выполняется при каждой итерации, хотя ее результат остается постоянным. Вы можете ускорить выполнение этого кода, переписав его так:

**Пример разомкнутого цикла (C++)**

```
if ( sumType == SUMTYPE_NET ) {
   for ( i = 0; i < count; i++ ) {
      netSum = netSum + amount[ i ];
   }
}
else {
   for ( i = 0; i < count; i++ ) {
      grossSum = grossSum + amount[ i ];
   }
}
```

**Примечание:** Этот фрагмент нарушает несколько правил хорошего программирования. Удобочитаемость и удобство сопровождения кода обычно важнее его быстродействия или размера, но темой этой главы является производительность, а для ее повышения часто нужно поступиться другими целями. Как и в предыдущей главе, здесь вы найдете примеры методик кодирования, которые в других частях этой книги не рекомендуются.

Размыкание этого цикла позволяет ускорить его выполнение примерно на 20%:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени |
|---|---|---|---|
| C++ | 2,81 | 2,27 | 19% |
| Java | 3,97 | 3,12 | 21% |
| Visual Basic | 2,78 | 2,77 | <1% |
| Python | 8,14 | 5,87 | 28% |

К сожалению, после размыкания цикла вам придется сопровождать оба цикла параллельно. Так, если переменную *count* потребуется заменить на *clientCount*, нужно будет изменить два фрагмента, что будет раздражать и вас, и всех других программистов, которым придется работать с вашим кодом.

Этот пример также иллюстрирует главную проблему оптимизации кода: результат любого отдельного вида оптимизации непредсказуем. Размыкание цикла оказалось выгодным для трех языков из четырех, но не для Visual Basic. В случае этой конкретной версии Visual Basic размыкание цикла только затруднило сопровождение кода, ничего не дав взамен. Урок очевиден: чтобы с уверенностью говорить о результатах любого вида оптимизации, вы должны их оценить. Никаких исключений.

## Объединение циклов

Если два цикла работают с одним набором элементов, можно выполнить их объединение (jamming). Выгода здесь объясняется устранением затрат, связанных с выполнением дополнительного цикла. Например, на объединение претендуют следующие циклы:

**Пример отдельных циклов, которые можно объединить (Visual Basic)**

```
For i = 0 to employeeCount - 1
   employeeName( i ) = ""
Next
...
For i = 0 to employeeCount - 1
   employeeEarnings( i ) = 0
Next
```

Объединение циклов обычно требует, чтобы условия циклов были одинаковы. В нашем примере оба цикла выполняются от *0* до *employeeCount - 1*, поэтому мы можем их объединить:

**Пример объединенного цикла (Visual Basic)**

```
For i = 0 to employeeCount - 1
   employeeName( i ) = ""
   employeeEarnings( i ) = 0
Next
```

Результаты объединения циклов таковы:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени |
|---|---|---|---|
| C++ | 3,68 | 2,65 | 28% |
| PHP | 3,97 | 2,42 | 32% |
| Visual Basic | 3,75 | 3,56 | 4% |

**Примечание:** тестирование выполнено для случая *employeeCount* = 100.

Как и прежде, все зависит от конкретного языка.

С объединением циклов связаны два главных фактора риска. Во-первых, индексы двух объединенных циклов могут позднее измениться, утратив совместимость. Во-вторых, объединить циклы иногда трудно. Прежде чем объединять циклы, убедитесь, что это не нарушит работу остальных частей кода.

## Развертывание цикла

Целью развертывания (unrolling) цикла является сокращение затрат, связанных с его выполнением. Если помните, после полного развертывания цикла из трех строк в главе 25 оказалось, что 10 отдельных обращений к массиву выполняются быстрее.

Полное развертывание цикла — быстрое решение, эффективное при малом числе элементов, но оно непрактично, если элементов много или вы не знаете заранее, с каким числом элементов вы будете иметь дело. Вот пример обычного цикла:

**Пример цикла, допускающего развертывание (Java)**

Для решения подобной задачи вы, вероятно, использовали бы цикл *for*, но перед оптимизацией вы должны были бы преобразовать его в цикл *while*. Ради ясности здесь показан цикл *while*.

```
i = 0;
while ( i < count ) {
   a[ i ] = i;
   i = i + 1;
}
```

После частичного развертывания цикла при каждой его итерации обрабатывается не один случай, а два или более. Это ухудшает удобочитаемость, но не нарушает общность цикла. Вот цикл, развернутый один раз:

**Пример однократного развертывания цикла (Java)**

```
i = 0;
while ( i < count - 1 ) {
   a[ i ] = i;
   a[ i + 1 ] = i + 1;
   i = i + 2;
}
```

```
Эти строки обрабатывают случай, который может быть упущен из-за увеличении счетчика цикла
на 2, а не на 1.
if ( i == count ) {
   a[ count - 1 ] = count - 1;
}
```

Как видите, мы заменили первоначальную строку *a[ i ] = i* на две строки и увеличиваем счетчик цикла на *2*, а не на *1*. Дополнительный код после цикла *while* нужен на случай нечетных значений переменной *count*, при которых цикл завершается, так и не обработав один элемент массива.

Конечно, девять строк хитрого кода труднее читать и сопровождать, чем пять строк простого. Что греха таить: после развертывания цикла качество кода ухудшилось. Однако любой подход к проектированию предполагает поиск компромиссных решений, и, даже если конкретная методика обычно плоха, в определенных обстоятельствах она может стать оптимальной.

Вот результаты развертывания цикла:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени |
|---|---|---|---|
| C++ | 1,75 | 1,15 | 34% |
| Java | 1,01 | 0,581 | 43% |
| PHP | 5,33 | 4,49 | 16% |
| Python | 2,51 | 3,21 | -27% |

**Примечание:** тестирование выполнено для случая *count* = 100.

Возможность ускорения кода на 16–43% заслуживает внимания, хотя, как показывает тест кода, написанного на Python, тут тоже не все однозначно. Главная опасность при развертывании цикла — ошибка завышения или занижения на единицу в коде, обрабатывающем последнюю итерацию.

Что, если мы продолжим развертывание цикла? Принесет ли дополнительную выгоду двойное развертывание?

### Пример двукратного развертывания цикла (Java)

```
i = 0;
while ( i < count - 2 ) {
   a[ i ] = i;
   a[ i + 1 ] = i+1;
   a[ i + 2 ] = i+2;
   i = i + 3;
}
if ( i <= count - 1 ) {
    a[ count - 1 ] = count - 1;
}
if ( i == count - 2 ) {
    a[ count -2 ] = count - 2;
}
```

Развертывание цикла во второй раз привело к таким результатам:

| Язык | Время выполнения кода до оптимизации | Время выполнения кода после второго развертывания цикла | Экономия времени |
|---|---|---|---|
| C++ | 1,75 | 1,01 | 42% |
| Java | 1,01 | 0,581 | 43% |
| PHP | 5,33 | 3,70 | 31% |
| Python | 2,51 | 2,79 | -12% |

**Примечание:** тестирование выполнено для случая *count* = 100.

Итак, дальнейшее развертывание цикла может принести дополнительную пользу, а может и не принести, как показывает случай языка Java. В то же время сложность кода быстро возрастает. Если предыдущий фрагмент не кажется вам таким уж сложным, вспомните, что в самом начале он был циклом из пяти строк, и сами оцените компромисс между производительностью и удобочитаемостью.

## Минимизация объема работы, выполняемой внутри циклов

Одной из методик повышения эффективности циклов является минимизация объема работы, выполняемой внутри цикла. Если вы можете вычислить выражение или его часть вне цикла и использовать внутри цикла результат вычисления, сделайте это. Это хорошая методика программирования, которая иногда улучшает удобочитаемость кода.

Допустим, у вас есть цикл, включающий сложное выражение с указателями:

**Пример цикла, включающего сложное выражение с указателями (C++)**

```
for ( i = 0; i < rateCount; i++ ) {
   netRate[ i ] = baseRate[ i ] * rates->discounts->factors->net;
}
```

Присвоив результат выражения удачно названной переменной, вы улучшите удобочитаемость кода, а может, и ускорите его выполнение:

**Пример упрощения сложного выражения с указателями (C++)**

```
quantityDiscount = rates->discounts->factors->net;
for ( i = 0; i < rateCount; i++ ) {
   netRate[ i ] = baseRate[ i ] * quantityDiscount;
}
```

Дополнительная переменная *quantityDiscount* (оптовая скидка) ясно показывает, что элементы массива *baseRate* умножаются на показатель скидки. В первом фрагменте это совсем не было очевидно. Кроме того, вынесение сложного выражения за пределы цикла устраняет три разыменования указателей при каждой итерации, что приводит к таким результатам:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени |
|---|---|---|---|
| C++ | 3,69 | 2,97 | 19% |
| C# | 2,27 | 1,97 | 13% |
| Java | 4,13 | 2,35 | 43% |

**Примечание:** тестирование выполнено для случая *rateCount* = 100.

За исключением компилятора Java экономия времени не так уж и велика, поэтому при первоначальном кодировании вы можете применить любую методику, улучшающую удобочитаемость кода, и отложить работу над быстродействием на потом.

## Сигнальные значения

Если цикл включает проверку сложного условия, время его выполнения часто можно сократить, упростив проверку. В случае циклов поиска это можно сделать, использовав сигнальное значение (sentinel value) — значение, которое располагается сразу после окончания диапазона поиска и непременно завершает поиск.

Классический пример сложной проверки, которую можно упростить с помощью сигнального значения, — условие цикла поиска, включающее проверки обнаружения нужной переменной и выхода за пределы диапазона поиска. Вот код такого цикла:

**Пример проверки сложного условия цикла (C#)**

```
found = FALSE;
i = 0;

Проверка сложного условия.
while ( ( !found ) && ( i < count ) ) {
   if ( item[ i ] == testValue ) {
      found = TRUE;
   }
   else {
      i++;
   }
}

if ( found ) {
   ...
```

При каждой итерации этого цикла проверяются два условия: *!found* и *i* < *count*. Проверка *!found* служит для определения того, найден ли нужный элемент. Проверка *i* < *count* нужна для предотвращения выхода за пределы массива. Кроме того, внутри цикла проверяются отдельные значения массива *item[]*, так что на самом деле при каждой итерации цикла выполняются три проверки.

Этот вид циклов поиска позволяет объединить три проверки и выполнять при каждой итерации только одну проверку: для этого нужно поместить в конце диапазона поиска «сигнальное значение», завершающее цикл. В нашем случае можно просто присвоить искомое значение элементу, располагающемуся сразу после окончания диапазона поиска (объявляя массив, не забудьте выделить место для этого элемента). Далее вы проверяете по очереди каждый элемент: если вы достигаете сигнального значения, значит, нужного вам значения в массиве нет. Вот соответствующий код:

**Пример использования сигнального значения для ускорения цикла (C#)**

```
// Установка сигнального значения с сохранением начальных значений.
initialValue = item[ count ];

Не забудьте зарезервировать в конце массива место для сигнального значения.
item[ count ] = testValue;

i = 0;
while ( item[ i ] != testValue ) {
   i++;
}

// Обнаружено ли значение?
if ( i < count ) {
   ...
```

Если *item* содержит целые числа, выгода может быть весьма существенной:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| C# | 0,771 | 0,590 | 23% | 1,3:1 |
| Java | 1,63 | 0,912 | 44% | 2:1 |
| Visual Basic | 1,34 | 0,470 | 65% | 3:1 |

**Примечание:** поиск выполнялся в массиве из 100 целых чисел.

Результаты, полученные для Visual Basic, особенно впечатляют, но и остальные тоже очень неплохи. Однако при изменении типа массива результаты также изменяются. Если *item* включает числа с плавающей запятой, результаты таковы:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени |
|---|---|---|---|
| C# | 1,351 | 1,021 | 24% |
| Java | 1,923 | 1,282 | 33% |
| Visual Basic | 1,752 | 1,011 | 42% |

**Примечание:** поиск выполнялся в массиве из 100 четырехбайтовых чисел с плавающей запятой.

Как обычно, многое зависит от языка.

Сигнальное значение можно использовать почти в любой ситуации, требующей выполнения линейного поиска, причем не только в массивах, но и в связных списках. Вы только должны тщательно выбирать сигнальные значения и с осторожностью включать их в структуры данных.

## Вложение более ресурсоемкого цикла в менее ресурсоемкий

Если вы имеете дело с вложенными циклами, подумайте, какой из них должен быть внешним, а какой внутренним. Вот пример вложенного цикла, который можно улучшить:

**Пример вложенного цикла, который можно улучшить (Java)**

```
for ( column = 0; column < 100; column++ ) {
   for ( row = 0; row < 5; row++ ) {
      sum = sum + table[ row ][ column ];
   }
}
```

Ключ к улучшению цикла в том, что внешний цикл состоит из гораздо большего числа итераций, чем внутренний. С выполнением любого цикла связаны накладные расходы: в начале цикла индекс должен быть инициализирован, а при каждой итерации — увеличен и проверен. Общее число итераций равно 100 для внешнего цикла и 100 * 5 = 500 для внутреннего цикла, что дает в сумме 600 итераций. Просто поменяв местами внешний и внутренний циклы, вы можете снизить число итераций внешнего цикла до 5, тогда как число итераций внутреннего цикла останется тем же. В итоге вместо 600 итераций будут выполнены только 505. Можно ожидать, что перемена циклов местами приведет примерно к 16%-ому улучшению: (600 – 505) / 600 = 16%. На самом деле результаты таковы:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени |
|---|---|---|---|
| C++ | 4,75 | 3,19 | 33% |
| Java | 5,39 | 3,56 | 34% |
| PHP | 4,16 | 3,65 | 12% |
| Python | 3,48 | 3,33 | 4% |

Значительные различия в очередной раз доказывают, что по поводу следствий оптимизации нельзя сказать ничего определенного, не оценив их в конкретной среде.

## Снижение стоимости операций

Под снижением стоимости (strength reduction) понимают замену дорогой операции на более дешевую, например, умножения на сложение. Иногда внутри цикла выполняется умножение индекса на какие-то другие значения. Сложение обычно выполняется быстрее, чем умножение, и, если вы можете вычислить то же число,

заменив умножение на прибавление значения при каждой итерации цикла, это скорее всего приведет к ускорению выполнения кода. Вот пример кода, основанного на умножении:

### Пример умножения с использованием индекса цикла (Visual Basic)

```
For i = 0 to saleCount - 1
   commission( i ) = (i + 1) * revenue * baseCommission * discount
Next
```

Этот код прост, но дорог. В то же время цикл можно переписать так, чтобы при каждой итерации выполнялось более дешевое сложение:

### Пример замены умножения на сложение (Visual Basic)

```
incrementalCommission = revenue * baseCommission * discount
cumulativeCommission = incrementalCommission
For i = 0 to saleCount - 1
   commission( i ) = cumulativeCommission
   cumulativeCommission = cumulativeCommission + incrementalCommission
Next
```

Этот вид изменения похож на купон, предоставляющий скидку со стоимости цикла. В первоначальном коде при каждой итерации выполнялось умножение выражения *revenue * baseCommission * discount* на счетчик цикла, увеличенный на единицу: сначала на 1, затем на 2, затем на 3 и т. д. В оптимизированном коде значение выражения *revenue * baseCommission * discount* присваивается переменной *incrementalCommission*. После этого при каждой итерации цикла значение *incrementalCommission* прибавляется к *cumulativeCommission*. При первой итерации оно прибавляется один раз, при второй — два, при третьей — три и т. д. Эффект тот же, что и при умножении *incrementalCommission* на 1, на 2, на 3 и т. д., но оптимизированный вариант дешевле.

Чтобы этот вид оптимизации оказался возможным, первоначальное умножение должно зависеть от индекса цикла. В данном примере индекс цикла был единственной изменяющейся частью выражения, поэтому мы и смогли сделать выражение более эффективным. Вот к чему это привело:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени |
|---|---|---|---|
| C++ | 4,33 | 3,80 | 12% |
| Visual Basic | 3,54 | 1,80 | 49% |

**Примечание:** тестирование выполнено для *saleCount* = 20. Все используемые в вычислении переменные были переменными с плавающей запятой.

# 26.3. Изменения типов данных

Изменение типов данных может быть эффективным способом сокращения кода и повышения его быстродействия. Проектирование структур данных в этой книге не рассматривается, но умеренные изменения реализации отдельных типов данных также могут повышать производительность. Ниже описано несколько способов оптимизации типов данных.

## Использование целых чисел вместо чисел с плавающей запятой

Сложение и умножение целых чисел, как правило, выполняются быстрее, чем аналогичные операции над числами с плавающей запятой. Например, циклы выполняются быстрее, если индекс имеет целочисленный тип.

**Перекрестная ссылка** Об использовании целых чисел и чисел с плавающей запятой см. главу 12.

**Пример неэффективного цикла с индексом с плавающей запятой (Visual Basic)**

```
Dim x As Single
For x = 0 to 99
   a( x ) = 0
Next
```

Сравните этот код с аналогичным циклом, в котором явно используется целочисленный индекс:

**Пример эффективного цикла с целочисленным индексом (Visual Basic)**

```
Dim i As Integer
For i = 0 to 99
   a( i ) = 0
Next
```

Насколько выгоден этот вид оптимизации? Вот результаты выполнения указанных фрагментов кода Visual Basic и аналогичных циклов, написанных на C++ и PHP:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного | Экономия времени | Соотношение быстродействия кода |
|---|---|---|---|---|
| C++ | 2,80 | 0,801 | 71% | 3,5:1 |
| PHP | 5,01 | 4,65 | 7% | 1:1 |
| Visual Basic | 6,84 | 0,280 | 96% | 25:1 |

## Использование массивов с минимальным числом измерений

Использовать массивы, имеющие несколько измерений, накладно. Если вы сможете структурировать данные так, чтобы их можно было хранить в одномерном, а не двумер-

**Перекрестная ссылка** О массивах см. раздел 12.8.

ном или трехмерном массиве, вы скорее всего ускорите выполнение программы. Допустим, у вас есть подобный код инициализации массива:

**Пример стандартной инициализации двумерного массива (Java)**

```
for ( row = 0; row < numRows; row++ ) {
   for ( column = 0; column < numColumns; column++ ) {
      matrix[ row ][ column ] = 0;
   }
}
```

При инициализации массива из 50 строк и 20 столбцов этот код выполняется вдвое дольше, чем код инициализации аналогичного одномерного массива, сгенерированный тем же компилятором Java. Вот как выглядел бы исправленный код:

**Пример одномерного представления массива (Java)**

```
for ( entry = 0; entry < numRows * numColumns; entry++ ) {
   matrix[ entry ] = 0;
}
```

А вот результаты тестирования этого кода и похожего кода, написанного на нескольких других языках:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| C++ | 8,75 | 7,82 | 11% | 1:1 |
| C# | 3,28 | 2,99 | 9% | 1:1 |
| Java | 7,78 | 4,14 | 47% | 2:1 |
| PHP | 6,24 | 4,10 | 34% | 1,5:1 |
| Python | 3,31 | 2,23 | 32% | 1,5:1 |
| Visual Basic | 9,43 | 3,22 | 66% | 3:1 |

**Примечание:** временные показатели, указанные для Python и PHP, получены в результате более чем в 100 раз меньшего числа итераций, чем показатели, приведенные для других языков, поэтому их непосредственное сравнение недопустимо.

Результаты этого вида оптимизации прекрасны для Visual Basic и Java, хороши для PHP и Python, но довольно заурядны для C++ и C#. Правда, время выполнения неоптимизированного кода C# было лучшим, так что на это едва ли можно жаловаться.

Широкий разброс результатов лишь подтверждает недальновидность слепого следования любым советам по оптимизации. Не испытав методику в конкретных обстоятельствах, ни в чем нельзя быть уверенным.

## Минимизация числа обращений к массивам

Кроме минимизации числа обращений к двумерным или трехмерным массивам часто выгодно минимизировать число обращений к массивам вообще. Подходящий кандидат для применения этой методики — цикл, в котором повторно ис-

пользуется один и тот же элемент массива. Вот пример необязательного обращения к массиву:

#### Пример необязательного обращения к массиву внутри цикла (C++)

```
for ( discountType = 0; discountType < typeCount; discountType++ ) {
   for ( discountLevel = 0; discountLevel < levelCount; discountLevel++ ) {
      rate[ discountLevel ] = rate[ discountLevel ] * discount[ discountType ];
   }
}
```

При изменении индекса *discountLevel* по мере выполнения внутреннего цикла обращение к массиву *discount[ discountType ]* остается все тем же. Вы можете вынести его за пределы внутреннего цикла, и тогда у вас будет одно обращение к массиву на одну итерацию внешнего, а не внутреннего цикла. Вот оптимизированный код:

#### Пример вынесения обращения к массиву за пределы цикла (C++)

```
for ( discountType = 0; discountType < typeCount; discountType++ ) {
   thisDiscount = discount[ discountType ];
   for ( discountLevel = 0; discountLevel < levelCount; discountLevel++ ) {
      rate[ discountLevel ] = rate[ discountLevel ] * thisDiscount;
   }
}
```

Результаты:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени |
|---|---|---|---|
| C++ | 32,1 | 34,5 | -7% |
| C# | 18,3 | 17,0 | 7% |
| Visual Basic | 23,2 | 18,4 | 20% |

**Примечание:** тестирование выполнено для *typeCount* = 10 и *levelCount* = 100.

Как обычно, результаты зависят от конкретного компилятора.

## Использование дополнительных индексов

Использование дополнительного индекса предполагает добавление данных, связанных с основным типом данных и повышающих эффективность обращений к нему. Связанные данные можно добавить к основному типу или хранить в параллельной структуре.

### Индекс длины строки

Примером использования дополнительного индекса может служить одна из форм представления строк. В языке C строки заканчиваются нулевым байтом. Что касается строк Visual Basic, то их длина хранится в начальном байте. Чтобы определить длину строки C, нужно начать с начала строки и продвигаться по ней, подсчитывая байты, до достижения нулевого байта. Для определения длины строки Visual Basic, нужно просто прочитать байт длины. Байт длины строки Visual Basic

— наглядный пример дополнения типа данных индексом, ускоряющим выполнение определенных операций, таких как вычисление длины строки.

Идею индексации длины можно приспособить к любому типу данных переменной длины. Слежение за длиной структуры часто — более эффективный подход, чем вычисление длины каждый раз, когда она требуется.

### Независимая параллельная структура индексации

Иногда выгоднее работать с индексом типа данных, а не с самим типом данных. Если элементы типа данных велики или их накладно перемещать (скажем, на диск), сортировка и поиск по индексам будут выполняться быстрее, чем непосредственные операции над данными. Если каждый элемент данных велик, вы можете создать вспомогательную структуру, состоящую из ключевых значений и указателей на подробную информацию. Если различие размеров элемента структуры данных и элемента вспомогательной структуры достаточно велико, элемент-ключ можно хранить в памяти, а сами данные — на внешнем носителе. Все операции поиска и сортировки будут выполняться в памяти, а к диску можно будет обращаться только после определения точного расположения нужного вам элемента.

## Кэширование

Кэширование — это такой способ хранения нескольких значений, при котором значения, используемые чаще всего, получить легче, чем значения, используемые реже. Так, если программа случайным образом читает записи с диска, метод может хранить в кэше записи, считываемые наиболее часто. Получив запрос записи, метод проверяет, имеется ли запись в кэше. Если да, запись возвращается непосредственно из памяти, а не считывается с диска.

Кэширование можно применять и в других областях. В программе обработки шрифтов Microsoft Windows узким местом было получение ширины символа при его отображении на экране. Кэширование ширины символа, использованного последним, позволило примерно вдвое ускорить отображение.

Вы можете кэшировать и результаты ресурсоемких вычислений, особенно если их параметры просты. Пусть, например, вам нужно найти длину гипотенузы прямоугольного треугольника по длинам двух катетов. Простая реализация этого метода была бы примерно такой:

#### Пример метода, напрашивающегося на кэширование (Java)

```
double Hypotenuse(
   double sideA,
   double sideB
   ) {
   return Math.sqrt( ( sideA * sideA ) + ( sideB * sideB ) );
}
```

Если вы знаете, что те же значения скорее всего будут переданы в метод повторно, их можно кэшировать:

### Пример кэширования для предотвращения дорогих вычислений (Java)

```
private double cachedHypotenuse = 0;
private double cachedSideA = 0;
private double cachedSideB = 0;

public double Hypotenuse(
   double sideA,
   double sideB
   ) {

   // Присутствуют ли параметры треугольника в кэше?
   if ( ( sideA == cachedSideA ) && ( sideB == cachedSideB ) ) {
      return cachedHypotenuse;
   }

   // Вычисление новой гипотенузы и ее кэширование.
   cachedHypotenuse = Math.sqrt( ( sideA * sideA ) + ( sideB * sideB ) );
   cachedSideA = sideA;
   cachedSideB = sideB;

   return cachedHypotenuse;
}
```

Вторая версия метода сложнее и объемнее первой, поэтому она должна обосновать свое право на жизнь быстродействием. Многие схемы кэширования предполагают кэширование более одного элемента, и с ними связаны еще большие затраты. Вот быстродействие двух версий метода:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| C++ | 4,06 | 1,05 | 74% | 4:1 |
| Java | 2,54 | 1,40 | 45% | 2:1 |
| Python | 8,16 | 4,17 | 49% | 2:1 |
| Visual Basic | 24,0 | 12,9 | 47% | 2:1 |

**Примечание:** эти результаты предполагают, что на каждый промах кэша приходятся два попадания.

Польза кэширования зависит от относительной стоимости обращения к кэшированному элементу, создания некэшированного элемента и сохранения нового элемента в кэше. Она также зависит от числа запросов кэшированной информации. Иногда она может зависеть и от аппаратного кэширования. В целом чем дороже генерирование нового элемента и чем чаще запрашивается та же самая информация, тем выгоднее кэширование. Чем дешевле обращение к кэшированному элементу и сохранение новых элементов в кэше, тем выгоднее кэширование. Как и другие методики оптимизации, кэширование усложняет код и часто оказывается источником ошибок.

# 26.4. Выражения

**Перекрестная ссылка** О выражениях см. раздел 19.1.

Многие задачи программирования требуют применения математических и логических выражений. Сложные выражения обычно дороги, и в этом разделе мы рассмотрим способы их удешевления.

## Алгебраические тождества

Алгебраические тождества иногда позволяют заменить дорогие операции на более дешевые. Так, следующие выражения логически эквивалентны:

```
not a and not b
not (a or b)
```

Выбрав второе выражение вместо первого, вы сэкономите одну операцию *not*.

Устранение одной операции *not*, вероятно, не приведет к заметным результатам, однако в целом этот принцип очень полезен. Так, Джон Бентли пишет, что в одной программе проверялось условие *sqrt(x) < sqrt(y)* (Bentley, 1982). Так как *sqrt(x)* меньше *sqrt(y)*, только когда *x* меньше, чем *y*, исходную проверку можно заменить на *x < y*. Если учесть дороговизну метода *sqrt()*, можно ожидать, что это приведет к огромной экономии. Так и есть:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| C++ | 7,43 | 0,010 | 99,9% | 750:1 |
| Visual Basic | 4,59 | 0,220 | 95% | 20:1 |
| Python | 4,21 | 0,401 | 90% | 10:1 |

## Снижение стоимости операций

Как уже было сказано, снижение стоимости операций подразумевает замену дорогой операции более дешевой. Вот некоторые возможные варианты:

- замена умножения сложением;
- замена возведения в степень умножением;
- замена тригонометрических функций их эквивалентами;
- замена типа *longlong* на *long* или *int* (следите при этом за аспектами производительности, связанными с применением целых чисел естественной и неестественной длины);
- замена чисел с плавающей запятой числами с фиксированной точкой или целые числа;
- замена чисел с плавающей запятой с удвоенной точностью числами с одинарной точностью;
- замена умножения и деления целых чисел на два операциями сдвига.

Допустим, вам нужно вычислить многочлен. Если вы забыли, что такое многочлены, напомню, что это выражения вида *Ax2 + Bx + C*. Буквы *A*, *B* и *C* — это коэффи-

циенты, а $x$ — переменная. Обычный код вычисления значения многочлена $n$-ной степени выглядит так:

**Пример вычисления многочлена (Visual Basic)**

```
value = coefficient( 0 )
For power = 1 To order
   value = value + coefficient( power ) * x^power
Next
```

Если вы подумаете о снижении стоимости операций, то поймете, что оператор возведения в степень — не самое эффективное решение в этом случае. Возведение в степень можно заменить на умножение, выполняемое при каждой итерации цикла, что во многом похоже на снижение стоимости, выполненное нами ранее, когда умножение было заменено на сложение. Вот как выглядел бы код, снижающий стоимость вычисления многочлена:

**Пример снижения стоимости вычисления многочлена (Visual Basic)**

```
value = coefficient( 0 )
powerOfX = x
For power = 1 to order
   value = value + coefficient( power ) * powerOfX
   powerOfX = powerOfX * x
Next
```

Если вы имеете дело с многочленами второй или более высокой степени, выгода может быть очень приличной:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| Python | 3,24 | 2,60 | 20% | 1:1 |
| Visual Basic | 6,26 | 0,160 | 97% | 40:1 |

Если вы действительно серьезно отнесетесь к снижению стоимости операций, то позаботитесь и о двух умножениях чисел с плавающей запятой. Стоимость операций, выполняемых в цикле, можно сделать еще меньшей, если постепенно возводить в нужную степень сразу несколько компонентов выражения, а не находить нужную степень при каждой итерации путем умножения:

**Пример дальнейшего снижения стоимости вычисления многочлена (Visual Basic)**

```
value = 0
For power = order to 1 Step -1
   value = ( value + coefficient( power ) ) * x
Next
value = value + coefficient( 0 )
```

В этой версии метода отсутствует переменная *powerOfX*, а вместо двух умножений при каждой итерации выполняется одно. Результаты таковы:

| Язык | Время кода выполнения до оптимизации | Время выполнения после первой оптимизации | Время выполнения после второй оптимизации | Экономия времени за счет второй оптимизации |
|---|---|---|---|---|
| Python | 3,24 | 2,60 | 2,53 | 3% |
| Visual Basic | 6,26 | 0,16 | 0,31 | -94% |

Это хороший пример расхождения теории и практики. Код, имеющий сниженную стоимость, казалось бы, должен работать быстрее, но на деле это не так. Возможно, в Visual Basic снижение производительности объясняется декрементом счетчика цикла на *1* вместо инкремента, но чтобы говорить об этом с уверенностью, эту гипотезу нужно оценить.

## Инициализация во время компиляции

Если вы вызываете метод, передавая ему в качестве единственного аргумента именованную константу или магическое число, попробуйте предварительно вычислить нужное значение, присвоить его константе и избежать вызова метода. Это же справедливо для умножения, деления, сложения и других операций.

Однажды мне понадобилось вычислять значение двоичного логарифма целого числа, округленное до ближайшего целого числа. Система не предоставляла метод вычисления двоичного логарифма, поэтому я написал собственный. Быстрый и легкий подход был основан на формуле:

```
log(x)base = log(x) / log(base)
```

**Перекрестная ссылка** О связывании переменных со значениями см. раздел 10.6.

Опираясь на это тождество, я написал такой метод:

**Пример метода, вычисляющего двоичный логарифм с использованием системных методов (C++)**

```
unsigned int Log2( unsigned int x ) {
   return (unsigned int) ( log( x ) / log( 2 ) );
}
```

Этот метод был очень медленным, а так как значение *log(2)* измениться не может, я заменил вызов метода *log(2)* на действительное значение, равное *0.69314718*:

**Пример метода, вычисляющего двоичный логарифм с использованием системного метода и константы (C++)**

```
const double LOG2 = 0.69314718;
...
unsigned int Log2( unsigned int x ) {
   return (unsigned int) ( log( x ) / LOG2 );
}
```

Вызов метода *log()* довольно дорог — гораздо дороже преобразования типа или деления, и поэтому резонно предположить, что уменьшение числа вызовов метода *log()* вдвое должно примерно в два раза ускорить выполнение метода. Вот результаты измерений:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени |
|---|---|---|---|
| C++ | 9,66 | 5,97 | 38% |
| Java | 17,0 | 12,3 | 28% |
| PHP | 2,45 | 1,50 | 39% |

Обоснованное предположение оказалось довольно близким к действительности. Если учесть невысокую предсказуемость результатов, приводимых в этой главе, точность моего прогноза в этом случае доказывает только то, что даже слепые белки иногда наталкиваются на орехи.

## Недостатки системных методов

Системные методы дороги и часто обеспечивают избыточную точность. Например, большинство системных математических методов написаны с тем расчетом, чтобы космонавт, отправившийся на Луну, прилунился с точностью ±2 фута. Если вам не нужна такая точность, нет смысла тратить на нее время.

В предыдущем примере метод *Log2()* возвращал целое число, но использовал для его вычисления метод *log()*, возвращающий число с плавающей запятой. Я не нуждался в такой точности, так что после своей первой попытки я написал ряд целочисленных проверок, которые прекрасно вычисляли целое значение двоичного логарифма:

**Пример метода, возвращающего примерное значение двоичного логарифма (C++)**

```
unsigned int Log2( unsigned int x ) {
   if ( x < 2 ) return 0 ;
   if ( x < 4 ) return 1 ;
   if ( x < 8 ) return 2 ;
   if ( x < 16 ) return 3 ;
   if ( x < 32 ) return 4 ;
   if ( x < 64 ) return 5 ;
   if ( x < 128 ) return 6 ;
   if ( x < 256 ) return 7 ;
   if ( x < 512 ) return 8 ;
   if ( x < 1024 ) return 9 ;
   ...
   if ( x < 2147483648 ) return 30;
   return 31 ;
}
```

Этот метод использует целочисленные операции, никогда не преобразовывает целые числа в числа с плавающей запятой и значительно превосходит по быстродействию оба метода, работающих с числами с плавающей запятой:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| C++ | 9,66 | 0,662 | 93% | 15:1 |
| Java | 17,0 | 0,882 | 95% | 20:1 |
| PHP | 2,45 | 3,45 | -41% | 2:3 |

Большинство так называемых «трансцендентных» функций разработано для наихудшего случая, т. е. внутри себя они даже целочисленный аргумент преобразуют в число с плавающей запятой с удвоенной точностью. Обнаружив вызов такой функции в проблемном фрагменте кода, уделите ей самое пристальное внимание, если, конечно, вам не нужна подобная точность.

Но вернемся к нашему методу. Еще один вариант его оптимизации основан на том факте, что деление на 2 аналогично операции сдвига вправо. Двоичный логарифм числа равен количеству операций деления на 2, которое можно выполнить над этим числом до получения нулевого значения. Вот как выглядит соответствующий код:

**Пример альтернативного метода, определяющего примерное значение двоичного логарифма с использованием оператора сдвига вправо (C++)**

```
unsigned int Log2( unsigned int x ) {
   unsigned int i = 0;
   while ( ( x = ( x >> 1 ) ) != 0 ) {
      i++;
   }
   return i ;
}
```

Читать этот код трудно, особенно программистам, не работавшим с C++. Сложное выражение в условии цикла *while* — прекрасный пример того, что следует использовать только в крайних случаях.

Этот метод выполняется примерно на 350% дольше, чем более длинная предыдущая версия (2,4 и 0,66 секунды соответственно), но он быстрее, чем первый оптимизированный метод, и легко адаптируется к 32-, 64-разрядным и другим средам.

Этот пример ясно показывает, насколько полезно продолжать поиск после нахождения первого успешного вида оптимизации. Первый вид оптимизации привел к приличному повышению быстродействия на 30–40%, но это не идет ни в какое сравнение с результатами второго и третьего видов оптимизации.

## Использование констант корректного типа

Используйте именованные константы и литералы, имеющие тот же тип, что и переменные, которым они присваиваются. Если константа и соответствующая ей переменная имеют разные типы, перед присвоением константы переменной компилятор должен будет выполнить преобразование типа. Хорошие компиляторы преобразуют типы во время компиляции, чтобы не снижалась производительность в период выполнения программы.

Однако менее эффективные компиляторы или интерпретаторы генерируют код, преобразующий типы в период выполнения. Чуть ниже указаны различия во времени инициализации переменной с плавающей точкой *x* и целочисленной переменной *i* в двух случаях. В первом случае инициализация выглядит так:

```
x = 5
i = 3.14
```

и требует преобразований типов. Во втором случае преобразования типов не нужны:

```
x = 3.14
i = 5
```

Результаты в очередной раз указывают на большие различия между компиляторами:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| C++ | 1,11 | 0,000 | 100% | Не поддается измерению |
| C# | 1,49 | 1,48 | <1% | 1:1 |
| Java | 1,66 | 1,11 | 33% | 1,5:1 |
| Visual Basic | 0,721 | 0,000 | 100% | Не поддается измерению |
| PHP | 0,872 | 0,847 | 3% | 1:1 |

## Предварительное вычисление результатов

При низкоуровневом проектировании часто приходится решать, вычислять ли результаты по ходу дела или лучше вычислить их один раз, сохранить и извлекать по мере надобности. Если результаты используются много раз, второй вариант часто предпочтительнее.

Этот выбор проявляется несколькими способами. На самом простом уровне вы можете вычислить часть выражения вне цикла вместо того, чтобы вычислять его внутри. Пример этого я приводил выше. На более сложном уровне вы можете вычислить табличные данные один раз при запуске программы и использовать их позднее; вы также можете сохранить результаты в файле данных или встроить их в программу.

Например, работая над игрой «звездные войны», программисты сначала вычисляли коэффициенты гравитации для разных расстояний от Солнца. Эти вычисления были ресурсоемкими и снижали производительность программы. Однако число разных расстояний, используемых в игре, было небольшим, поэтому разработчики в итоге вычислили коэффициенты гравитации предварительно и сохранили их в массиве из 10 элементов. Получение коэффициентов из массива оказалось гораздо более быстрым, чем их вычисление.

**Перекрестная ссылка** Об использовании табличных данных вместо сложной логики см. главу 18.

Допустим, у вас есть метод, вычисляющий сумму, которую нужно уплатить при погашении ссуды на покупку автомобиля. Код подобного метода мог бы выглядеть так:

**Пример сложного выражения, которое можно вычислить предварительно (Java)**

```
double ComputePayment(
   long loanAmount,
   int months,
```

```
   double interestRate
   ) {
   return loanAmount /
      (
      ( 1.0 - Math.pow( ( 1.0 + ( interestRate / 12.0 ) ), -months ) ) /
      ( interestRate / 12.0 )
      );
}
```

Формула вычисления платежей по ссуде сложна и довольно дорога. Помещение информации в таблицу, наверное, сделало бы вычисление более дешевым.

Насколько крупной была бы эта таблица? Переменной с наибольшим диапазоном является переменная *loanAmount*. Переменная *interestRate* может принимать значения от 5% до 20% с шагом 0,25%, что дает нам 61 значение. Переменная *months* может иметь значение от 12 до 72, или 61 значение. Значение переменной *loanAmount*, вероятно, может находиться в пределах от 1000 до 100 000 долларов, и вам едва ли удастся сохранить все эти значения в таблице.

Однако большая часть выражения не зависит от *loanAmount*, поэтому параметры действительно отвратительной части (знаменатель более крупного выражения) можно сохранить в таблице, индексируемой значениями *interestRate* и *months*. Значение *loanAmount* нужно будет вычислять:

**Пример предварительного вычисления сложного выражения (Java)**

```
double ComputePayment(
   long loanAmount,
   int months,
   double interestRate
   ) {

Новая переменная interestIndex используется как индекс массива loanDivisor.

   int interestIndex =
      Math.round( ( interestRate - LOWEST_RATE ) * GRANULARITY * 100.00 );
   return loanAmount / loanDivisor[ interestIndex ][ months ];
}
```

Итак, сложное вычисление мы заменили вычислением индекса массива и одним обращением к массиву. К чему же это привело?

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| Java | 2,97 | 0,251 | 92% | 10:1 |
| Python | 3,86 | 4,63 | –20% | 1:1 |

В зависимости от обстоятельств вы должны были бы предварительно вычислять массив *loanDivisor* при инициализации программы или читать его из файла. Вы также могли бы инициализировать массив нулями, вычислять каждый элемент при его первом запросе, сохранять его, а впоследствии просто извлекать из массива. Это было бы одной из форм кэширования, которое обсуждалось выше.

Предварительное вычисление выражения не обязывает вас создавать таблицу — возможен и иной вариант. Допустим, у вас есть код, вычисляющий взносы, которые нужно уплатить при погашении ссуд на разную сумму:

**Пример второго сложного выражения, которое можно вычислить предварительно (Java)**

```
double  ComputePayments(
   int months,
   double interestRate
   ) {
   for ( long loanAmount = MIN_LOAN_AMOUNT; loanAmount < MAX_LOAN_AMOUNT;
      loanAmount++ ) {
      payment = loanAmount / (
         ( 1.0 - Math.pow( 1.0+(interestRate/12.0), - months ) ) /
         ( interestRate/12.0 )
        );

Следующий код делал бы что-то с переменной payment; что именно — в данном примере не важно.

      ...
   }
}
```

Даже без таблицы, вы можете предварительно вычислить сложную часть выражения вне цикла и использовать найденное значение внутри цикла:

**Пример предварительного вычисления второго сложного выражения (Java)**

```
double  ComputePayments(
   int months,
   double interestRate
   ) {
   long loanAmount;

Предварительное вычисление части выражения.

    double divisor = ( 1.0 - Math.pow( 1.0+(interestRate/12.0), - months ) ) /
       ( interestRate/12.0 );
   for ( long loanAmount = MIN_LOAN_AMOUNT; loanAmount <= MAX_LOAN_AMOUNT;
      loanAmount++ ) {
      payment = loanAmount / divisor;
      ...
   }
}
```

Этот вид оптимизации похож на уже рассмотренные нами методики, предполагающие вынесение обращений к массиву и разыменований указателей за пределы цикла. Результаты оптимизации кода Java в данном случае сравнимы с результатами предыдущего вида оптимизации, основанного на предварительном вычислении табличных данных:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени | Соотношение быстродействия |
|---|---|---|---|---|
| Java | 7,43 | 0,24 | 97% | 30:1 |
| Python | 5,00 | 1,69 | 66% | 3:1 |

В отличие от первой попытки оптимизации быстродействие кода Python сейчас также выросло. Это происходит довольно часто: если один вид оптимизации не приводит к желаемым результатам, другой, казалось бы, аналогичный вид оказывается очень эффективным.

Оптимизация программы путем предварительного вычисления выражений имеет несколько вариантов:

- вычисление результатов до выполнения программы и связывание их с константами во время компиляции;
- вычисление результатов до выполнения программы и присвоение их переменным, используемым в период выполнения;
- вычисление результатов до выполнения программы и сохранение их в файле, загружаемом в период выполнения;
- однократное вычисление результатов при запуске программы и их использование во всех последующих случаях;
- вычисление как можно большего числа значений до начала цикла, позволяющее свести к минимуму объем работы, выполняемой внутри цикла;
- вычисление результатов при возникновении первой потребности в них и их сохранение, позволяющее получить результаты, когда они понадобятся снова.

## Устранение часто используемых подвыражений

Если какое-то выражение повторяется в коде несколько раз, присвойте его значение переменной и используйте переменную вместо вычисления выражения в нескольких местах. Подвыражение, которое можно устранить, уже встречалось нам в предыдущем подразделе:

### Пример часто используемого подвыражения (Java)

```
payment = loanAmount / (
      ( 1.0 - Math.pow( 1.0 + ( interestRate / 12.0 ), - months ) ) /
      ( interestRate / 12.0 )
   );
```

Вместо двукратного вычисления выражения *interestRate/12.0* вы можете присвоить его переменной и обратиться к ней два раза. Если вы присвоите переменной удачное имя, этот вид оптимизации не только повысит производительность кода, но и улучшит его удобочитаемость. Вот оптимизированный код:

### Пример устранения часто используемого подвыражения (Java)

```
monthlyInterest = interestRate / 12.0;
payment = loanAmount / (
```

```
    ( 1.0 - Math.pow( 1.0 + monthlyInterest, - months ) ) /
    monthlyInterest
);
```

Результаты не впечатляют:

| Язык | Время выполнения кода до оптимизации | Время выполнения оптимизированного кода | Экономия времени |
|---|---|---|---|
| Java | 2,94 | 2,83 | 4% |
| Python | 3,91 | 3,94 | –1% |

По-видимому, метод *Math.pow()* настолько дорог, что он перекрывает выгоду устранения подвыражения. Возможно также, что подвыражение уже было устранено компилятором. Если бы подвыражение составляло не такую малую часть стоимости всего выражения или если бы оптимизатор компилятора был менее эффективен, этот вид оптимизации, наверное, оказал бы большее влияние на производительность.

# 26.5. Методы

Одним из самых эффективных способов оптимизации кода является грамотная декомпозиция программы на методы. Небольшие, хорошо определенные методы делают программу компактнее, устраняя повторяющиеся фрагменты кода. Они упрощают оптимизацию, потому что рефакторинг одного метода улучшает все методы, которые его вызывают. Небольшие методы относительно легко переписать на низкоуровневом языке. Объемные хитроумные методы понять сложно, а после переписывания их на низкоуровневом языке вроде ассемблера это вообще невыполнимо.

**Перекрестная ссылка** Об использовании методов см. главу 7.

## Встраивание методов

На заре программирования вызывать методы на некоторых компьютерах было крайне дорого. Вызов метода означал, что ОС должна выгрузить программу, загрузить каталог методов, загрузить конкретный метод, выполнить метод, выгрузить метод и снова загрузить вызвавший метод. Все это потребляло много ресурсов и замедляло программу.

Современные компьютеры облагают вызовы методов гораздо меньшей пошлиной. Например, встраивание метода копирования строк приводит к таким результатам:

| Язык | Время выполнения метода | Время выполнения встроенного кода | Экономия времени |
|---|---|---|---|
| C++ | 0,471 | 0,431 | 8% |
| Java | 13,1 | 14,4 | –10% |

В некоторых случаях вы можете сэкономить несколько наносекунд, встроив код метода в программу, используя ключевое слово *inline* языка C++ или аналогичную возможность. Если ваш язык не поддерживает *inline* непосредственно, но имеет препроцессор макросов, вы можете встраивать код при помощи макроса, включая и выключая встраивание по требованию. Однако современные компьютеры

(под «современными» я понимаю любые машины, с которыми вы можете столкнуться в своей работе), при вызове метода почти не тратят дополнительных ресурсов. Как показывает пример, встроив код, вы можете как улучшить производительность, так и ухудшить ее.

## 26.6. Переписывание кода на низкоуровневом языке

Давняя мудрость, которую не стоит оставлять без внимания, гласит, что при низком быстродействии код следует переписать на языке низкого уровня. Если вы пишете на C++, языком низкого уровня может быть ассемблер, если на Python — C. Переписывание кода на низкоуровневом языке обычно положительно влияет и на быстродействие кода, и на его объем. Типичный подход к оптимизации при помощи низкоуровневого языка таков.

1. Напишите все приложение на высокоуровневом языке.
2. Выполните полное тестирование приложения и проверьте его корректность.
3. Если производительность недостаточна, выполните профилирование приложения с целью выявления горячих точек. Так как около 50% времени выполнения программы обычно приходится примерно на 5% кода, горячими точками обычно будут небольшие фрагменты программы.
4. Перепишите несколько небольших фрагментов на низкоуровневом языке для повышения общей производительности программы.

**Перекрестная ссылка** Подробнее о том, что основная часть времени выполнения программы приходится на малый процент кода, см. подраздел «Принцип Парето» раздела 25.2.

Последуете ли вы по этому проторенному пути, зависит от того, насколько хорошо вы программируете на низкоуровневых языках, насколько хорошо проблема подходит для решения на низкоуровневом языке, а также от степени вашего отчаяния. Сам я впервые применил эту методику при реализации алгоритма Data Encryption Standard, о чем я упоминал в главе 25. Я перепробовал все виды оптимизации, о которых когда-либо слышал, но программа все равно работала вдвое медленнее, чем нужно. Мне ничего не оставалось делать, кроме как переписать часть программы на ассемблере. Не имея особого опыта программирования на ассемблере, я по сути ограничился простым переводом кода с высокоуровневого языка на ассемблер, но даже этот примитивный подход ускорил выполнение программы на 50%.

Рассмотрим переписывание на ассемблере метода, преобразующего двоичные данные в символы ASCII верхнего регистра. В следующем примере показан соответствующий код Delphi:

**Пример кода на Delphi, который лучше переписать на ассемблере**

```
procedure HexExpand(
   var source: ByteArray;
   var target: WordArray;
   byteCount: word
);
```

```
var
   index: integer;
   lowerByte: byte;
   upperByte: byte;
   targetIndex: integer;
begin
   targetIndex := 1;
   for index := 1 to byteCount do begin
      target[ targetIndex ] := ( (source[ index ] and $F0) shr 4 ) + $41;
      target[ targetIndex+1 ] := (source[ index ] and $0f) + $41;
      targetIndex := targetIndex + 2;
   end;
end;
```

Трудно увидеть жир в этом коде, однако он содержит много манипуляций с битами, что не является сильной стороной Delphi. А вот ассемблер подходит для этого как нельзя лучше, так что этот код является хорошим кандидатом на переписывание. Вот что получается в итоге:

### Пример метода, переписанного на ассемблере

```
procedure HexExpand(
   var source;
   var target;
   byteCount : Integer
);
    label
    EXPAND;

    asm
          MOV   ECX, byteCount      // Загрузка числа расширяемых байт.
          MOV   ESI, source         // Смещение источника.
          MOV   EDI, target         // Смещение приемника.
          XOR   EAX, EAX            // Обнуление индекса смещения в массиве.

    EXPAND:
          MOV   EBX, EAX            // Смещение в массиве.
          MOV   DL, [ESI+EBX]       // Получение байта источника.
          MOV   DH, DL              // Копирование байта источника.

          AND   DH, $F              // Получение старших разрядов.
          ADD   DH, $41             // Преобразование символа в верхний регистр.

          SHR   DL, 4               // Сдвиг разрядов на нужную позицию.
          AND   DL, $F              // Получение младших разрядов.
          ADD   DL, $41             // Преобразование символа в верхний регистр.

          SHL   BX, 1               // Удвоение смещения в массиве-приемнике.
          MOV   [EDI+EBX], DX       // Копирование слова в приемник.
```

```
        INC   EAX                // Увеличение индекса смещения в массиве.
        LOOP  EXPAND             // Повторение цикла.
    end;
```

Переписывание этого кода на ассемблере привело к уменьшению времени выполнения на 41%. Логично предположить, что код, написанный на языке, более подходящем для операций над битами, (скажем, на C++), менее восприимчив к этому виду оптимизации, чем код Delphi. Проверим:

| Язык | Время выполнения высокоуровневого кода | Время выполнения ассемблерного кода | Экономия времени |
|---|---|---|---|
| C++ | 4,25 | 3,02 | 29% |
| Delphi | 5,18 | 3,04 | 41% |

Второй столбец этой таблицы отражает эффективность двух языков в отношении операций над битами. После оптимизации время выполнения стало почти одинаковым — перевод кода на ассемблер свел к минимуму первоначальные различия между Delphi и C++.

Полученный нами метод показывает, что результатом переписывания кода на ассемблере не всегда является огромный и уродливый метод. Итоговый код часто оказывается довольно компактным. Иногда ассемблерный код почти так же компактен, как его высокоуровневый эквивалент.

Существует одна относительно легкая и эффективная стратегия переписывания кода на ассемблере. В самом начале воспользуйтесь компилятором, генерирующем ассемблерные листинги во время компиляции. Извлеките ассемблерный код метода, который вам нужно оптимизировать, и сохраните его в отдельном исходном файле. Используя этот код как основу, выполните оптимизацию вручную, проверяя корректность кода и оценивая улучшения на каждом этапе. Некоторые компиляторы вставляют в ассемблерный код высокоуровневые операторы в форме комментариев. Если ваш компилятор на это способен, можете сохранить первоначальные операторы в ассемблерном коде в качестве документации.

http://cc2e.com/2672

### Контрольный список: методики оптимизации кода

**Улучшение и быстродействия, и объема**

- ❑ Замените сложную логику на обращения к таблице.
- ❑ Объедините циклы.
- ❑ Используйте целочисленные переменные вместо переменных с плавающей запятой.
- ❑ Инициализируйте данные во время компиляции.
- ❑ Используйте константы корректного типа.
- ❑ Вычислите результаты предварительно.
- ❑ Устраните часто используемые подвыражения.
- ❑ Перепишите ключевые методы на низкоуровневом языке.

**Улучшение только быстродействия**

- ❑ Прекращайте проверку сразу же после получения ответа.
- ❑ Упорядочивайте тесты в блоках case и цепочках операторов *if-then-else* по частоте.

- ❑ Сравните быстродействие похожих структур логики.
- ❑ Используйте методику отложенных вычислений.
- ❑ Разомкните цикл, содержащий оператор *if*.
- ❑ Выполните развертывание цикла.
- ❑ Минимизируйте объем работы, выполняемой внутри цикла.
- ❑ Используйте сигнальные значения в циклах поиска.
- ❑ Вложите более ресурсоемкий цикл в менее ресурсоемкий.
- ❑ Снизьте стоимость операций, выполняемых внутри цикла.
- ❑ Замените многомерные массивы на одномерные.
- ❑ Минимизируйте число обращений к массивам.
- ❑ Дополните типы данных индексами.
- ❑ Кэшируйте часто используемые значения.
- ❑ Проанализируйте алгебраические тождества.
- ❑ Снизьте стоимость логических и математических выражений.
- ❑ Остерегайтесь системных методов.
- ❑ Встройте методы.

## 26.7. Если что-то одно изменяется, что-то другое всегда остается постоянным

Трудно не согласиться с тем, что за 10 лет, прошедших со времени первого издания этой книги, некоторые параметры производительности систем изменились. Компьютеры стали гораздо быстрее, а объем памяти вырос во много раз. Работая над первым изданием, для получения выразительных измеримых результатов я выполнял большинство тестов этой главы от 10 000 до 50 000 раз. При подготовке этого издания мне пришлось выполнять большинство тестов от 1 до 100 млн раз. Если для получения измеримых результатов тест нужно выполнить 100 млн раз, стоит задуматься над тем, заметит ли хоть кто-нибудь последствия выполненной оптимизации в реальной программе. Компьютеры стали столь мощными, что для многих распространенных типов программ виды оптимизации, описанные в этой главе, стали нерелевантными.

В то же время другие аспекты производительности совсем не изменились. Возможно, программистов, создающих приложения для настольных ПК, вопросы оптимизации уже почти не тревожат, но разработчики программ для встроенных систем, систем, работающих в реальном времени, и других систем, обладающих ограниченными ресурсами, все еще должны владеть методиками оптимизации кода.

Необходимость оценки результатов каждой попытки оптимизации кода не теряет актуальности с 1971 г., когда Дональд Кнут опубликовал свое исследование программ Fortran. Судя по данным, приведенным в этой главе, результаты отдельных видов оптимизации на самом деле сейчас *менее предсказуемы*, чем 10 лет назад. Они зависят от языка, компилятора, и его версии, используемых библиотек, их версий, параметров компилятора и многого другого.

Оптимизация кода неизбежно подразумевает нахождение компромисса между сложностью, удобочитаемостью, простотой и удобством сопровождения программы, с одной стороны, и желанием повысить производительность — с другой. Не-

обходимое при оптимизации перепрофилирование кода приводит к значительному росту затрат на сопровождение программы.

Есть один хороший способ сопротивления соблазну преждевременной оптимизации и содействия созданию ясного и простого кода: требуйте, чтобы оптимизация приводила к *измеримому улучшению*. Если оптимизация оправдывает профилирование кода и оценку результатов, наверное, ее следует выполнить, если будет показано, что она работает. Но если оптимизация не оправдывает проведения профилирования, она не может оправдать ухудшения удобочитаемости, удобства сопровождения и других характеристик кода. Влияние неоцененной оптимизации кода на производительность в лучшем случае может быть только теоретическим, тогда как ее влияние на удобочитаемость столь же определенно, сколь пагубно.

## Дополнительные ресурсы

http://cc2e.com/2679

По-моему, лучшая работа по оптимизации кода — *Writing Efficient Programs* (Bentley, Englewood Cliffs, NJ: Prentice Hall, 1982). Это довольно старая книга, и все же постарайтесь ее найти. В ней вы найдете экспертное обсуждение общих вопросов оптимизации кода. Бентли описывает методики обмена времени на пространство и пространства на время, а также приводит несколько примеров перепроектирования типов данных, позволяющего и ускорить код, и сделать его компактнее. Подход Бентли чуть более описателен, чем тот, что принял я, но приведенные им случаи весьма интересны. Бентли проводит несколько методов через несколько этапов оптимизации, позволяя увидеть результаты первой, второй и третьей попыток. Описание главной идеи занимает 135 страниц. Эта книга отличается необычайно высоким отношением «сигнал/шум», что делает ее одним из редких бриллиантов, которые следует иметь каждому практикующему программисту.

В приложении 4 книги Бентли *Programming Pearls, 2d ed.* (Bentley, Boston, MA: Addison-Wesley, 2000) вы можете найти резюме правил оптимизации кода, описанных в его более ранней книге.

http://cc2e.com/2686

Кроме того, есть целый ряд книг, в которых вопросы оптимизации рассматриваются в контексте конкретных технологий. Некоторые из них перечислены ниже, а самый свежий список вы найдете на Web-странице, адрес которой указан слева.

Booth, Rick. *Inner Loops: A Sourcebook for Fast 32-bit Software Development*. Boston, MA: Addison-Wesley, 1997.

Gerber, Richard. *Software Optimization Cookbook: High-Performance Recipes for the Intel Architecture*. Intel Press, 2002.

Hasan, Jeffrey and Kenneth Tu. *Performance Tuning and Optimizing ASP.NET Applications*. Berkeley, CA: Apress, 2003.

Killelea, Patrick. *Web Performance Tuning*, 2d ed. Sebastopol, CA: O'Reilly & Associates, 2002.

Larman, Craig and Rhett Guthrie. *Java 2 Performance and Idiom Guide*. Englewood Cliffs, NJ: Prentice Hall, 2000.

Shirazi, Jack. *Java Performance Tuning*. Sebastopol, CA: O'Reilly & Associates, 2000.

Wilson, Steve and Jeff Kesselman. *Java Platform Performance: Strategies and Tactics*. Boston, MA: Addison-Wesley, 2000.

## Ключевые моменты

- Результаты конкретных видов оптимизации во многом зависят от языка, компилятора и среды. Не оценив результатов оптимизации, вы не сможете сказать, помогает она программе или вредит.
- Первый вид оптимизации часто далеко не самый лучший. Обнаружив эффективный вид оптимизации, продолжайте пробовать и, возможно, найдете еще более эффективный.
- Оптимизация кода похожа на ядерную энергию — это противоречивая и эмоциональная тема. Кто-то считает, что оптимизация настолько ухудшает надежность и удобство сопровождения программы, что ее вообще выполнять не следует. Другие думают, что при соблюдении должной предосторожности она приносит пользу. Если вы решите использовать методики, описанные в этой главе, будьте внимательны и осторожны.

Часть VI

# СИСТЕМНЫЕ ВОПРОСЫ

- **Глава 27.** Как размер программы влияет на конструирование
- **Глава 28.** Управление конструированием
- **Глава 29.** Интеграция
- **Глава 30.** Инструменты программирования

ГЛАВА 27

# Как размер программы влияет на конструирование

http://cc2e.com/2761

**Содержание**



**Связанные темы**

- Предварительные требования к конструированию: глава 3
- Определение вида ПО, над которым вы работаете: раздел 3.2
- Управление конструированием: глава 28

Масштабирование в области разработки ПО — это не просто вопрос увеличения составных частей небольшого проекта. Допустим, вы разработали программный комплекс Gigatron, содержащий 25 000 строк кода, за 20 человеко-месяцев и нашли 500 ошибок во время производственного тестирования. Допустим, Gigatron 1.0 имел успех, так же, как и Gigatron 2.0, и вы начали работу над Gigatron Deluxe — значительно улучшенной версией программы, которая предположительно будет состоять из 250 000 строк кода.

Хотя ее размер в 10 раз превышает размер начальной версии, Gigatron Deluxe потребует не 10-кратных усилий при разработке, а 30-кратных. Более того, 30-кратные общие затраты не означают 30-кратных затрат только на конструирование. Возможно, потребуется потратить в 25 раз больше усилий на конструирование, и в 40 — на архитектуру и системное тестирование. Точно так же вы не получите 10-кратный прирост ошибок — он может быть 15-кратным и даже больше.

Если вы привыкли работать над маленькими проектами, то ваш первый проект среднего или большого размера может вырваться из-под контроля, превратившись в необузданного зверя, вместо благоприятного исхода, который вы себе представляли. Эта глава рассказывает, какие виды хищников можно встретить и где найти хлыст и кольцо для их укрощения. С другой стороны, если вы привыкли работать

с большими проектами, то используемые вами подходы могут оказаться слишком формализованными применительно к малым задачам. Эта глава описывает, как сэкономить, чтобы предохранить небольшие проекты от обрушения под грузом накладных расходов.

## 27.1. Взаимодействие и размер

Если вы единственный человек, работающий над проектом, то единственное направление взаимодействия проходит между вами и заказчиком, если не считать путь между левым и правым полушариями вашего мозга. С ростом числа участников проекта, увеличивается и число вариантов взаимодействия. Причем это число растет не аддитивно в соответствии с количеством людей, а мультипликативно — пропорционально квадрату числа участников (рис. 27-1).

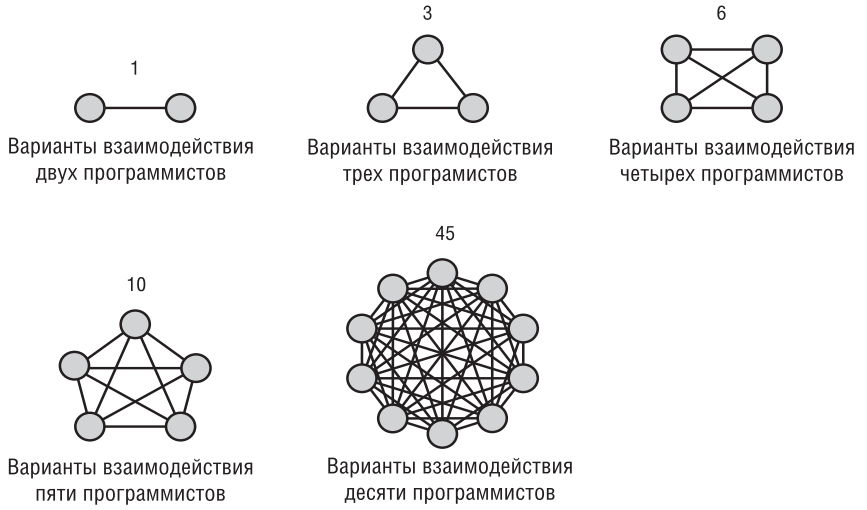



***Рис. 27-1.*** *Количество вариантов взаимодействия растет пропорционально квадрату числа участников команды*

Как видите, в проекте с двумя участниками только одно направление взаимодействия. Проект с пятью участниками содержит 10 возможных вариантов. В проекте с 10 участниками может быть 45 направлений, при условии, что каждый человек может контактировать с любым другим. 10% проектов, в которых занято 50 или более программистов, содержат минимум 1200 потенциальных вариантов. Чем больше путей взаимодействия, тем больше времени вы тратите на общение и тем больше вероятность появления ошибок в процессе взаимодействия. Проекты большего размера требуют принятия организационных мер, которые упорядочивают взаимодействие или существенно его ограничивают.

Типичный подход к упорядочению взаимодействия состоит в его формализации с помощью документов. Вместо непосредственного общения 50 человек друг с другом в любом возможном сочетании эти 50 человек пишут и читают документы. Некоторые документы — текстовые, некоторые — графические. Некоторые напечатаны на бумаге, а некоторые — хранятся в электронной форме.

## 27.2. Диапазон размеров проектов

Является ли размер проекта, над которым вы работаете, типичным? Широкий диапазон размеров проектов означает, что вы не можете считать типичным никакой из возможных размеров. Одним из способов оценить размер проекта служит оценка численности команды разработчиков, работавших над этим проектом. Вот как примерно выглядит процентное соотношение всех проектов, выполненных командами разных размеров:

| Размер команды разработчиков | Примерная доля проектов (в %) |
|---|---|
| 13 | 25% |
| 410 | 30% |
| 1125 | 20% |
| 2650 | 15% |
| 50+ | 10% |

**Источники:** Получено на основе данных из «A Survey of Software Engineering Practice: Tools, Methods, and Results» (Beck and Perkins, 1983), «Agile Software Development Ecosystems» (Highsmith, 2002) и «Balancing Agility and Discipline» (Boehm and Turner, 2003).

Один из неочевидных аспектов информации о размерах проектов заключается в различии между процентным соотношением проектов разной величины и числом программистов, работающих над проектами каждого размера. Поскольку над большими проектами работает больше программистов, чем над маленькими, то в них занят больший процент всех программистов. Приведем приблизительную оценку процентного соотношения количества программистов, работающих над проектами разного размера:

| Размер команды разработчиков | Примерная доля программистов (в %) |
|---|---|
| 13 | 5% |
| 410 | 10% |
| 1125 | 15% |
| 2650 | 20% |
| 50+ | 50% |

**Источники:** Получено на основе данных из «A Survey of Software Engineering Practice: Tools, Methods, and Results» (Beck and Perkins, 1983), «Agile Software Development Ecosystems» (Highsmith, 2002) и «Balancing Agility and Discipline» (Boehm and Turner, 2003).

## 27.3. Влияние размера проекта на возникновение ошибок

**Перекрестная ссылка** Подробнее об ошибках см. раздел 22.4.

Размер проекта влияет как на количество, так и на тип возникающих ошибок. Вы могли и не предполагать, что тип ошибок также подвержен влиянию, но с ростом размера проекта обычно все больший процент ошибок можно отнести к просчетам в технических требованиях и проектировании (рис. 27-2).

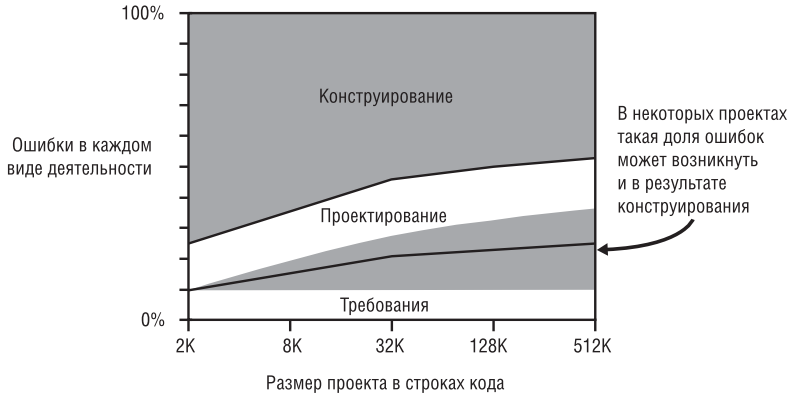



***Рис. 27-2.*** *При увеличении размера проекта обычно возрастает количество ошибок, возникающих из-за просчетов в технических требованиях и проектировании. И все же иногда ошибки в основном возникают при конструировании (Boehm, 1981, Grady, 1987, Jones, 1998)*

В маленьких проектах ошибки конструирования составляют до 75% найденных ошибок. Методология меньше влияет на качество кода, а наибольшее влияние на систему часто оказывает мастерство индивидуума, разрабатывающего эту программу (Jones, 1998).

В больших проектах ошибки конструирования могут сократиться примерно до 50%; все остальные можно отнести к ошибкам в требованиях и архитектуре. По-видимому, это связано с тем, что в больших проектах разработке требований и архитектурному проектированию нужно уделять больше времени, поэтому при выполнении этих операций пропорционально возрастает и вероятность возникновения ошибок. Однако в некоторых очень больших проектах доля ошибок конструирования остается высокой — иногда даже в системах с 500 000 строк кода, 75% ошибок можно отнести к области конструирования (Grady 1987).

Аналогично изменению типа при увеличении размера проекта меняется и количество ошибок. Вы могли ожидать, что проект, превышающий другой вдвое, закономерно содержит вдвое больше ошибок. Но на самом деле плотность дефектов — число ошибок на 1000 строк кода — возрастает. Вдвое больший продукт содержит более чем в два раза больше ошибок. Табл. 27-1 показывает диапазоны плотностей дефектов, появление которых вы можете ожидать в проектах разных размеров.

**Табл. 27-1. Размер проекта и типичная плотность ошибок**

| Размер проекта (число строк кода) | Типичная плотность ошибок |
|---|---|
| Менее 2K | 0–25 ошибок на 1000 строк кода (thousand lines of code, KLOC) |
| 2K–16K | 0–40 ошибок на KLOC |
| 16K–64K | 0,5–50 ошибок на KLOC |
| 64K–512K | 2–70 ошибок на KLOC |
| 512K или больше | 4–100 ошибок на KLOC |

**Источники:** «Program Quality and Programmer Productivity» (Jones, 1977), «Estimating Software Costs» (Jones, 1998).

**Перекрестная ссылка** Данные представляют собой среднее значение. Горстка организаций сообщает о лучшем соотношении ошибок, чем приведенные здесь минимальные величины. Примеры см. в подразделе «Сколько ошибок вы можете найти?» раздела 22.4.

Данные для этой таблицы получены на основе специализированных проектов, и эти числа могут иметь мало общего с данными проектов, над которыми вы работаете. Однако в качестве моментального снимка для отрасли они весьма показательны: число ошибок значительно возрастает при увеличении размера проекта, и очень большие проекты имеют до четырех раз больше ошибок на 1000 строк кода, чем маленькие. Для достижения одинакового соотношения ошибок в больших проектах придется приложить больше усилий.

## 27.4. Влияние размера проекта на производительность

Когда речь заходит о размере проекта, производительность имеет много общего с качеством ПО. При небольших размерах (2000 строк кода и менее) наибольшее влияние на производительность оказывает мастерство отдельного программиста (Jones, 1998). С увеличением размера проекта все больше на производительность начинают влиять численность команды и организация.

Насколько большим должен быть проект, чтобы численность команды разработчиков стала влиять на производительность? В «Prototyping Versus Specifying: a Multiproject Experiment» Бом, Грей и Сиволдт (Boehm, Gray and Seewaldt) сообщали, что команды меньших размеров выполняли проекты с производительностью на 39% выше, чем более многочисленные команды. Размер команд? Два человека для маленьких проектов и три — для больших (1984). Табл. 27-2 позволяет получить представление о взаимосвязи между размером проекта и производительностью.

**Табл. 27-2. Размер проекта и производительность**

| Размер проекта (число строк кода) | Число строк кода на человека в год (в скобках указано номинальное значение Cocomo II*) |
|---|---|
| 1K | 2500–25 000 (4000) |
| 10K | 2000–25 000 (3200) |
| 100K | 1000–20 000 (2600) |
| 1 000K | 700–10 000 (2000) |
| 10 000K | 300–5000 (1600) |

**Источники:** Составлено на основе данных из «Measures for Excellence» (Putnam and Meyers, 1992), «Industrial Strength Software» (Putnam and Meyers, 1997), «Software Cost Estimation with Cocomo II» (Boehm et al., 2000), и «Software Development Worldwide: The State of the Practice» (Cusumano et al., 2003).

* Constructive Cost Model (конструктивная стоимостная модель) — метод оценки затрат на разработку ПО. — *Прим. перев.*

Производительность в значительной степени определяется видом ПО, над которым вы работаете, квалификацией персонала, языком программирования, методологией, сложностью продукта, программной средой, инструментарием, способом подсчета «строк кода», тем, как усилия по поддержке, не относящиеся напрямую к программированию, переводятся в «количество строк кода на человека в год», и другими факторами. Поэтому конкретные цифры в табл. 27-2 очень сильно отличаются.

Однако тенденция, представленная этими числами, очень показательна. Производительность в малых проектах может быть в 2–3 раза выше, чем в больших, а разница в производительности между самыми маленькими и самыми большими проектами может достигать 5–10 раз.

# 27.5. Влияние размера проекта на процесс разработки

Если вы работаете над проектом в одиночку, то наибольшее влияние на его успех или провал оказываете вы сами. Если вы работаете над проектом, в котором заняты 25 человек, то потенциально все еще можете оказывать наибольшее влияние, но скорее всего отдельный человек не может присвоить себе такое достижение — на успех или провал сильнее будет влиять организация.

## Соотношение между выполняемыми операциями и размер

По мере того как растет размер проекта и увеличивается необходимость формальных взаимодействий, меняются и виды операций, выполняемых в проекте. Соотношение операций в процессе разработки для проектов различных размеров можно представить так (рис. 27-3):

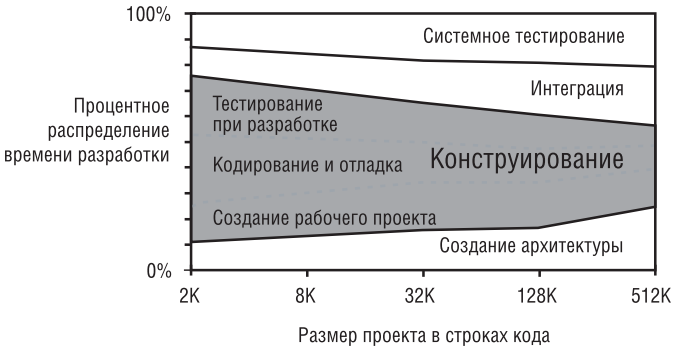



***Рис. 27-3.*** *Конструирование преобладает в маленьких проектах. Проекты большего размера требуют больших затрат на архитектуру, интеграцию и системное тестирование. Работа по составлению технических требований на этой диаграмме не показана, потому что в отличие от других операций эти усилия не зависят от размера программы напрямую. (Albrecht, 1979; Glass, 1982; Boehm, Gray and Seewaldt, 1984; Boddie, 1987; Card, 1987; McGarry, Waligora and McDermott, 1989; Brooks, 1995; Jones, 1998; Jones, 2000; Boehm et al., 2000)*

В малых проектах конструирование является наиболее заметной частью проекта: на него приходится до 65% всего времени разработки. В средних по величине проектах конструирование все еще доминирует, но его доля снижается до 50% от всех трудозатрат. В очень больших проектах архитектура, интеграция и системное тестирование занимают все больше времени, и доля конструирования становится все менее заметной. Короче, при возрастании размера проекта доля конструирования в общих затратах уменьшается. Этот график выглядит так, будто при его продолжении вправо конструирование исчезнет совсем. Поэтому, чтобы не остаться без работы, я ограничился размером в 512K.

Доля конструирования снижается, потому что с ростом размера проекта все составляющие конструирования: проектирование, кодирование, отладка и модульное тестирование — пропорционально масштабируются, однако затраты на другие виды деятельности растут гораздо быстрее (рис. 27-4).

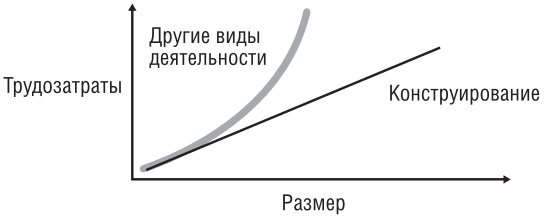



***Рис. 27-4.*** *Объем работ по конструированию ПО — практически линейная функция от размера проекта. Трудозатраты на другие виды работ при увеличении проекта растут нелинейно*

В близких по размеру проектах будут выполняться примерно одинаковые операции, однако при расхождении в размерах виды выполняемых действий тоже начнут различаться. Как сказано во вступлении к этой главе, если Gigatron Deluxe в 10 раз больше начального Gigatron, он потребует в 25 раз больше затрат на конструирование, в 25–50 — на планирование, в 30 — на интеграцию и в 40 — на архитектуру и системное тестирование.

Соотношение между видами деятельности изменяется, потому что при разных размерах проекта разные виды деятельности становятся критическими. Барри Бом и Ричард Тернер (Barry Boehm and Richard Turner) выяснили, что расходы на проработку архитектуры в пределах 5% от общей стоимости проекта позволяют минимизировать затраты для проектов, содержащих примерно 10 000 строк кода. Но наилучшие результаты можно получить, если в проектах размером в 100 000 строк кода израсходовать на архитектуру 15–20% средств (Boehm and Turner, 2004).

Далее приведены виды деятельности, затраты на которые с ростом размера проекта увеличиваются нелинейно:

- взаимодействие;
- планирование;
- управление;
- разработка требований;
- функциональное проектирование системы;

- проектирование и спецификация интерфейса;
- архитектура;
- интеграция;
- удаление дефектов;
- системное тестирование;
- документирование.

Независимо от размера проекта некоторые технологии всегда имеют большое значение: дисциплинированная практика кодирования, проверка проекта и кода другими разработчиками, хороший инструментарий и использование высокоуровневых языков. Эти приемы представляют ценность в маленьких проектах и бесценны в больших.

## Программы, продукты, системы и системные продукты

Не только количество строк кода и размер команды влияют на размер проекта. Более тонкое воздействие оказывают качество и сложность окончательной версии программного продукта. Для создания и отладки первой версии Gigatron мог потребоваться всего месяц. Это была отдельная программа, написанная, протестированная и задокументированная одним человеком. Если Gigatron длиной 2500 строк занял один месяц, то почему окончательный вариант Gigatron, длиной 25 000 строк потребовал 20 месяцев?

**Дополнительные сведения** Другое объяснение этой точки зрения см. в главе 1 книги «The Mythical Man-Month» (Brooks, 1995).

Простейший вид ПО — это отдельная «программа», используемая человеком, ее написавшим, или несколькими другими.

Более сложный вид программы — это программный «продукт», предназначенный для использования другими людьми, а не самим разработчиком. Программный продукт используется в средах, отличающихся о той, в которой этот продукт был создан. Он тщательно протестирован перед сдачей в эксплуатацию, задокументирован, и другие люди могут осуществлять его сопровождение. Разработка программного продукта стоит примерно втрое дороже разработки программы.

Другой уровень сложности требуется и для разработки пакета программ, работающих вместе. Такой пакет называется программной «системой». Разработать систему гораздо сложнее, чем простую программу из-за сложности разработки интерфейсов между отдельными частями и необходимости интеграции этих частей. В целом система также примерно втрое дороже, чем простая программа.

Создание «системного продукта» предполагает наведение глянца, присущего продукту, и наличие нескольких частей системы. Системный продукт примерно в девять раз дороже простой программы (Brooks, 1995, Shull et al., 2002).

Неучитываемые различия в сложности и глянце между программами, продуктами, системами и системными продуктами и являются частой причиной ошибок в оценках. Программисты, использующие свой опыт в написании программ для составления плана создания программного продукта, могут ошибиться в меньшую сторону примерно в 10 раз. Рассматривая следующий пример, обратитесь к графику на рис. 27-3. Если вы воспользуетесь своим опытом в написании 2K строк

кода, чтобы оценить, сколько времени займет разработка программы длиной в 2K строк, ваши расчеты будут учитывать только 65% времени, необходимого для выполнения всех операций по созданию программы. Написание 2K строк кода не занимает столько же времени, сколько создание целой программы, состоящей из 2K строк. Если вы не учитываете время, затрачиваемое не на конструирование, разработка займет на 50% больше времени, чем вы рассчитывали.

При увеличении проекта конструирование требует все меньшей доли общих затрат. Если вы основываете свои расчеты только на опыте конструирования, оценочная ошибка растет. Если вы использовали свой опыт написания 2K строк для оценки времени, необходимого для разработки программы длиной 32K, ваша оценка будет включать только 50% от необходимого времени; весь процесс разработки может занять на 100% больше времени, чем вы предполагали.

Ошибку в оценке можно полностью приписать вашему непониманию влияния размера на разработку больших программ. Кроме того, если вы не учли дополнительных усилий по наведению лоска, в отличие от простой программы оценочная ошибка может легко увеличиться в три или более раз.

## Методология и размер

Методология используется в проектах любых размеров. В маленьких проектах методология скорее случайна и инстинктивна — в больших она скрупулезна и тщательно спланирована.

Порой методология бывает столь неопределенной, что программисты даже не подозревают, что ее используют. Некоторые программисты доказывают, что методология лишена гибкости, поэтому они не будут ее использовать. Хотя они могут не использовать методологию сознательно, любой подход к программированию состоит из методологии независимо от простоты этого подхода. Обычный утренний подъем и поездка на работу являются рудиментарной методологией, хотя и не слишком вдохновляющей. Программисты, отказывающиеся использовать методологию, на самом деле просто не используют ее явно — куда от нее денешься!

Формальный подход не всегда доставляет удовольствие, а если он применяется неправильно, накладные расходы могут поглотить получаемые от него преимущества. Однако возрастающая сложность больших проектов требует более осознанного отношения к методологии. Строить небоскреб и собачью будку нужно по-разному. То же относится и к программным проектам разных размеров. В больших проектах случайный выбор методологии не соответствует задаче. Успешные проектировщики выбирают стратегии для больших проектов осознанно.

В светском обществе чем формальней событие, тем более неудобную одежду нужно надевать (высокие каблуки, галстуки и т. д.). В разработке ПО чем формальней проект, тем больше бумажных документов необходимо создать, чтобы убедиться, что вы выполнили ваше домашнее задание. Кейперс Джонс указывает, что в проекте длиной в 1000 строк примерно 7% затрат приходится на бумажную работу, тогда как в проекте в 100 000 строк затраты на бумажную работу увеличиваются в среднем до 26% (Jones, 1998).

Эта работа проводится не из чистой любви к писанине. Она является прямым результатом интересного феномена: чем больше человек вам приходится координировать, тем больше требуется документов (рис. 27-1).

Вы не создаете эти документы ради самого факта их существования. Смысл написания плана управления конфигурацией, например, не в том, чтобы потренировать пальцы рук, но в том, чтобы заставить вас тщательно продумать управление конфигурацией и объяснить ваш план любому желающему. Документация — это осязаемый побочный эффект проделанной вами реальной работе по планированию и конструированию системы. Если вам кажется, что вы делаете это ради проформы и пишете обобщенные документы, значит, что-то не в порядке.

«Больше» не значит лучше, во всяком случае в том, что касается методологии. В своем сравнении быстрых и четко спланированных вариантов методологии Барри Бом и Ричард Тернер предупреждают, что обычно лучше сначала создать небольшие методы, а затем промасштабировать их для большого проекта, чем начать с создания универсального метода, а затем урезать его для маленького проекта (Boehm and Turner, 2004). Некоторые ученые мужи в области ПО говорят о «легковесной» и «тяжеловесной» методологии, но на практике главное — это учесть конкретный размер и тип вашего проекта, а затем найти «правильновесную» методологию.

## Дополнительные ресурсы

Для дальнейшего изучения темы данной главы ознакомьтесь со следующими источниками.

http://cc2e.com/2768

Boehm, Barry и Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA: Addison-Wesley, 2004. Бом и Тернер рассказывают, как размер проекта влияет на применение быстрых и четко спланированных методов, а также рассматривают другие вопросы, касающиеся быстроты и четкого планирования.

Cockburn, Alistair. *Agile Software Development*. Boston, MA: Addison-Wesley, 2002. В главе 4 обсуждаются вопросы выбора подходящей методологии проекта, в том числе учитываются размеры проекта. Глава 6 представляет Кристаллическую методологию Кокберна (Cockburn's Crystal Methodologies), в которой определены подходы к разработке проектов различной величины и степени критичности.

Boehm, Barry W. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981. Бом всестороннее рассматривает такие зависящие от размера проекта величины, как стоимость, производительность и качество, а также другие переменные, участвующие в процессе разработки ПО. Книга включает обсуждение влияния размера на конструирование и другие операции. Глава 11 содержит отличное объяснение отрицательных последствий масштабирования ПО. Другая информация о размере проекта распределена по всей книге. Выпущенная в 2000 г. книга Бома *Software Cost Estimation with Cocomo II* содержит больше современной информации относительно оценочной модели Бома Cocomo, но более ранняя книга включает более глубокое обсуждение вопроса, все еще представляющее интерес.

Jones, Capers. *Estimating Software Costs*. New York, NY: McGraw-Hill, 1998. Эта книга заполнена таблицами и графиками, анализирующими источники производительности разработки ПО. Что касается конкретного вопроса влияния размера проекта, то раздел «The Impact of Program Size» главы 3 изданной в 1986 г. книги Джонса *Programming Productivity* содержит отличное обсуждение.

Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition* (2d ed.). Reading, MA: Addison-Wesley, 1995. Брукс работал в IBM менеджером разработки OS/360 — гигантского проекта, занявшего 5000 человеко-лет. В своем очаровательном эссе он обсуждает вопросы управления, свойственные маленьким и большим командам, и высказывает исключительно яркое мнение о командах главных программистов.

DeGrace, Peter, и Leslie Stahl. *Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms*. Englewood Cliffs, NJ: Yourdon Press, 1990. Как следует из названия, книга классифицирует подходы к разработке ПО. Как упоминалось на протяжении всей этой главы, подход должен изменяться при изменении размера проекта, и Дегрейс и Стал делают эту точку зрения очевидной. В разделе «Attenuating and Truncating» главы 5 обсуждается настройка процессов разработки ПО в соответствии с размером проекта и другими формальностями. Книга содержит описания моделей из НАСА и Минобороны, а также большое количество поучительных иллюстраций.

Jones, T. Capers. «Program Quality and Programmer Productivity». *IBM Technical Report TR 02.764* (January 1977): 42–78. Статья также доступна в книге Джонса *Tutorial: Programming Productivity: Issues for the Eighties*, 2d ed. Los Angeles, CA: IEEE Computer Society Press, 1986. Эта статья содержит первый глубокий анализ причин, по которым распределение расходов в больших проектах отлично от маленьких. Это исчерпывающее обсуждение различий между большими и маленькими проектами, включающее требования и меры по обеспечению качества. Статья старая, но все еще интересная.

## Ключевые моменты

- С ростом размера проекта появляется необходимость поддерживать взаимодействие. Смысл большинства подходов к методологии состоит в уменьшении проблем взаимодействия, поэтому методология должна жить или умереть в зависимости от ее вклада в облегчение взаимодействия.
- При прочих равных, производительность в больших проектах будет ниже, чем в маленьких.
- При прочих равных большие проекты будут содержать больше ошибок на 1000 строк кода, чем маленькие.
- Деятельность, которая в малых проектах сама собой разумеется, в больших проектах должна быть тщательно спланирована. С ростом размера проекта конструирование занимает все меньшую ее часть.
- Увеличение масштаба легковесной методологии обычно работает лучше, чем уменьшение масштаба тяжеловесной. Наиболее эффективный подход состоит в использовании «правильновесной» методологии.

ГЛАВА 28

# Управление конструированием

**Содержание**



http://cc2e.com/2836

**Связанные темы**

- Предварительные требования к конструированию: глава 3
- Определение вида ПО, над которым вы работаете: раздел 3.2
- Размер программы: глава 27
- Качество ПО: глава 20

На протяжении нескольких последних десятилетий управление разработкой ПО является задачей повышенной сложности. Обсуждение общих вопросов управления программными проектами выходит за рамки данной книги, но в этой главе рассматриваются некоторые специальные темы управления, напрямую связанные с конструированием (рис. 28-1). Если вы разработчик, эта глава поможет вам понять те вопросы, которые менеджеры должны принимать во внимание; если же менеджер — понять, как менеджеры рассматривают разработчиков, а также как эффективно управлять конструированием. Поскольку глава охватывает широкий спектр вопросов, то в некоторых разделах также приведены источники дополнительной информации.

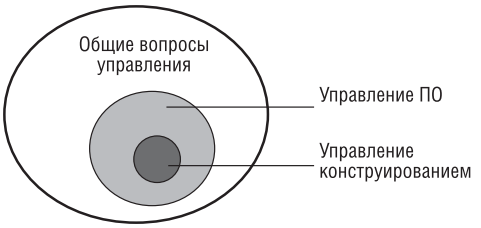



***Рис. 28-1.*** *Эта глава охватывает вопросы управления ПО, относящиеся к конструированию*

Если вы интересуетесь управлением разработкой ПО, прочитайте раздел 3.2, чтобы понять различие между традиционными последовательными подходами к разработке и современными итеративными подходами. Не забудьте также прочесть главы 20 и 27, поскольку и требования к качеству, и размеры проекта влияют на то, как должно осуществляться управление данным конкретным проектом.

# 28.1. Поощрение хорошего кодирования

Поскольку код — это основной результат конструирования, то ключевой вопрос в управлении конструированием звучит так: «Как содействовать выработке хорошей практики кодирования?» Вообще внедрение строгого набора технических стандартов с позиций менеджера — не очень хорошая идея. Программисты склонны рассматривать менеджеров как низшую ступень технической эволюции, где-то между одноклеточными организмами и мамонтами, вымершими в ледниковый период. Поэтому, если внедрение программных стандартов необходимо, программисты должны в этом участвовать.

Если какой-то участник проекта собирается определять стандарты, это должен быть скорее архитектор, пользующийся уважением, а не менеджер. Программные проекты опираются в такой же степени на «профессиональную иерархию», как и на «должностную». Если этот архитектор считается идейным лидером проекта, команда в большинстве случаев будет придерживаться стандартов, установленных им.

Если вы выбрали этот подход, убедитесь, что архитектора действительно уважают. Иногда архитектор проекта — это просто старший по возрасту человек, который работает достаточно давно и не имеет представления о проблемах промышленного кодирования. Программисты будут протестовать против стандартов, навязанных таким «архитектором», поскольку они не имеют ничего общего с их работой.

## Вопросы внедрения стандартов

В одних организациях стандарты полезны, в других — не очень. Некоторые разработчики приветствуют стандартизацию, потому что она снижает вероятность возникновения случайных разногласий. Если ваша группа сопротивляется применению строгих стандартов, рассмотрите несколько альтернатив: гибкие правила, предложения вместо правил или набор примеров, иллюстрирующих наилучшую практику.

## Технологии поощрения хорошего кодирования

Ниже описаны способы достижения хорошего качества кодирования, не столь тяжеловесные, как утверждение жестких стандартов:

***Назначьте двух человек на каждую часть проекта*** Если над каждой строкой кода приходится работать двоим, то у вас есть гарантия, что хотя бы два человека думают, что код работает и его можно читать. Механизм работы команды из двух человек может варьироваться от парного программирования до пары «наставник — стажер» или рецензирования кода методом близнецов.

**Перекрестная ссылка** О парном программировании см. раздел 21.2.

***Рецензируйте каждую строку кода*** В рецензировании кода обычно участвует программист и минимум два рецензента. Это значит, что каждую строку кода читают не менее трех человек. Другое название парного рецензирования — «парное давление» (peer pressure). Кроме предоставления страховки на тот случай, если первоначальный программист покинет проект, рецензирование кода улучшает его качество, так как программист знает, что его код будут читать другие. Даже если у вас нет явных стандартов кодирования, рецензирование позволяет незаметно продвигаться к созданию группового стандарта — в процессе рецензирования группа принимает решения, которые со временем могут стать ее стандартами.

**Перекрестная ссылка** О рецензировании см. разделы 21.3 и 21.4.

***Введите процедуру подписания кода*** В других отраслях технические чертежи утверждаются и подписываются управляющим инженером. Подпись означает, что в соответствии с уровнем квалификации инженера, этот чертеж технически компетентен и не содержит ошибок. Некоторые компании обращаются с кодом так же: только код, подписанный старшим техническим персоналом, считается завершенным.

***Распространяйте для ознакомления хорошие примеры кода*** Важной составляющей хорошего управления является способность ясно донести свои требования. Для этих целей вы можете распространять хороший код между программистами или выставлять его на всеобщее обозрение. Так вы предоставляете ясный пример того качества, которого вы добиваетесь. Точно так же кодовые стандарты могут состоять главным образом из «листингов отличного кода». Присвоение определенному листингу «знака качества» — пример для подражания. Подобные руководства легче обновлять, чем писаные (словами) стандарты. Кроме того, так легко показать тонкости стиля кодирования, которые тяжело по пунктам описать словами.

***Подчеркивайте, что код — это общее имущество*** Иногда программисты считают, что код, который они написали, — «их» личная собственность. Хотя это результат их работы, но он является частью всего проекта и должен быть доступен любому участнику проекта. Он должен просматриваться другими хотя бы во время рецензирования и сопровождения.

**Перекрестная ссылка** Большая часть программирования состоит в сообщении результатов вашей работы другим людям (см. разделы 33.5 и 34.3).

Один из самых успешных проектов, о котором когда-либо сообщалось, был реализован за 11 лет и состоял из 83 000 строк кода. За первые 13 месяцев эксплуатации была выявлена только одна ошибка, приведшая к системному сбою. Это достижение еще больше ошеломляет, если учитывать, что проект был завершен в конце 1960-х, без использования онлайн-компиляции и интерактивной отладки. Производительность проекта —7500 строк кода в год в конце 60-х — и по сегодняшним стандартам достаточно впечатляет. Главный программист проекта сообщил, что одним из залогов успеха было признание всех машинных прогонов (ошибочных и нет) общим, а не частным достоянием (Baker and Mills, 1973). Эта идея нашла свое воплощение и в современной ситуации, например, в ПО с открытым исходными кодом (Open Source Software) (Raymond, 2000), экстремальном программировании с его понятием о коллективном владении (Beck, 2000) и во многих других случаях.

***Награждайте за хороший код*** Используйте систему поощрений для закрепления практики хорошего кодирования. При разработке таких поощрений принимайте во внимание следующие соображения.

- Награда должна представлять интерес для программиста. (Многим из них поощрения типа «Какой молодец!» неприятны, особенно когда исходят от нетехнических менеджеров.)
- Код, поощряемый таким образом, должен быть исключительно хорошим. Награждая программиста, которого все остальные считают плохим работником, вы уподобляетесь Гомеру Симпсону, пытающемуся запустить ядерный реактор. Не имеет значения, что этот программист всегда демонстрирует готовность к сотрудничеству или никогда не опаздывает на работу. Вы потеряете кредит доверия, если ваша награда не будет соответствовать технической стороне вопроса. Если вы недостаточно технически подкованы, чтобы судить о качестве кода, — не делайте этого. Лучше совсем не вручать награду или позволить команде выбрать достойного.

***Один простой стандарт*** Если вы управляете программным проектом и в прошлом сами были программистом, то простым и эффективным способом добиться хорошего результата будет фраза: «Я должен быть в состоянии прочесть и понять любой код, написанный в проекте». То, что менеджер — не самый технически продвинутый специалист, может быть преимуществом, так как препятствует созданию заумного или хитрого кода.

## Роль этой книги

Большая часть книги содержит обсуждение хорошей практики программирования. Она не предназначена для оправдания жестких стандартов — используйте ее как повод для дискуссии, источник сведений о хорошей программной практике, а также для поиска таких методик, которые могут дать преимущества в вашей среде программирования.

# 28.2. Управление конфигурацией

Программный проект динамичен. Меняется код, меняется проект, меняются и требования. Более того, изменения в требованиях влекут еще большие изменения в проекте, а изменения в проекте приводят к еще более значительным изменениям в коде и тестовых примерах.

## Что такое управление конфигурацией?

Управление конфигурацией — это практика определения элементов проекта и систематическая обработка изменений таким образом, чтобы система могла поддерживать свою целостность длительное время. Другое название этого процесса — «контроль изменений». Он включает в себя методики определения предполагаемых изменений, отслеживание изменений и хранение копий системы в том виде, в каком она существовала в разные моменты.

Если вы не контролируете изменения в технических требованиях, это может закончиться написанием кода для тех частей системы, которые в какой-то момент были отменены. Вы можете создать код, несовместимый с новыми элементами системы. Эти несовместимости можно не обнаружить до момента интеграции, который также станет началом поиска виноватых, потому что на самом деле никто не будет знать, что происходит.

Если не контролируются изменения в коде, вы можете изменить содержимое метода, который в то же время меняет кто-то другой, причем успешно объединить ваши и чужие изменения будет проблематично. При неконтролируемых изменениях код может выглядеть протестированным лучше, чем на самом деле. Версия, прошедшая тестирование, возможно, была старой, еще не измененной. Без хорошего контроля изменений вы можете внести в метод несколько исправлений, найти новые ошибки, но при этом не будете иметь возможности восстановления предыдущей рабочей версии метода.

Если изменения не обрабатываются систематически, то вы блуждаете в тумане, вместо того чтобы двигаться прямиком к ясной цели. Без хорошего контроля изменений вместо разработки кода вы впустую тратите время. Управление конфигурацией позволяет использовать время эффективно.

Несмотря на очевидную необходимость управления конфигурацией, многие программисты избегают его десятилетиями. Опрос, проведенный более 20 лет назад, выяснил, что около трети программистов даже не были знакомы с этой идеей (Beck and Perkins, 1983), и нет никаких признаков, что это положение изменилось. Более свежее исследование Института Разработки ПО показало, что среди организаций, использующих неформальные методики разработки ПО менее 20% применяют адекватное управление конфигурацией (SEI, 2003).

Управление конфигурацией изобрели не программисты, но поскольку программные проекты чрезвычайно изменчивы, то для них оно особенно полезно. Применительно к программным проектам управление конфигурацией обычно называется «управление конфигурацией ПО» (software configuration management, SCM). SCM сосредотачивается на программных требованиях, исходном коде, документации и тестах.

Систематическая ошибка SCM заключается в избыточном контроле. Абсолютно надежный способ прекратить автомобильные аварии — это запретить всем водить машину, а абсолютно надежный способ предотвратить проблемы с разработкой ПО — прекратить всю разработку. Хотя это и один из вариантов контроля изменений, это неудачный способ разрабатывать ПО. Необходимо тщательно планировать SCM, чтобы оно давало преимущества, а не было камнем на шее.

**Перекрестная ссылка** О влиянии размера проекта на конструирование см. главу 27.

В маленьком проекте, разрабатываемом в одиночку, вы, вполне возможно, обойдетесь без всякого SCM, исключая планирование периодического нерегулярного создания резервных копий. И все же управление конфигурацией еще способно принести пользу (на самом деле я пользовался такой системой при создании этой рукописи). В больших проектах, выполняемых 50 участниками, вам, вероятно, потребуется полнофункциональная схема SCM, включающая довольно формальные процедуры резервного копирования, контроль изменений требований и проекта, а также контроль над документами, исходным кодом, содержанием, тестовыми вариантами и другими элементами проекта. Если ваш проект не слишком велик и не слишком мал, вы должны установить уровень формальности, находящийся между двумя экстремумами. Следующие подразделы описывают некоторые варианты реализации SCM.

## Изменения в требованиях и проекте

**Перекрестная ссылка** Одни подходы к разработке лучше поддерживают изменения, другие — хуже (см. раздел 3.2).

Во время разработки вас переполняют идеи о том, как улучшить систему. Если вы будете реализовывать каждое изменение, вы скоро почувствуете, что бежите на месте: хотя система будет постоянно меняться, она не будет приближаться к завершению. Вот некоторые советы по контролю изменений в проекте.

***Следуйте систематической процедуре контроля изменений*** Систематическая процедура контроля изменений — это настоящая находка в случае, когда у вас большое количество запросов на изменение (см. раздел 3.4). Установив систематическую процедуру, вы даете понять, что изменения будут рассмотрены в контексте наибольшей пользы для проекта в целом.

***Обрабатывайте запросы на изменения группами*** Реализация простых изменений по мере возникновения идей выглядит заманчиво. Проблема такого подхода в том, что хорошие изменения могут затеряться. Если вы задумали простое изменение при 25%-й готовности проекта и сроки это позволяют, вы внесете это изменение. Если вы придумали еще одно простое изменение, когда готово 50% проекта, но вы уже опаздываете, вы не будете его вносить. Когда сроки начинают поджимать в конце проекта, уже не имеет значения, что второе изменение в 10 раз лучше, чем первое: вы не в том положении, чтобы делать несущественные исправления. Часть самых лучших изменений может уйти сквозь пальцы просто потому, что подумали о них слишком поздно.

Решение этой проблемы состоит в записи всех идей и предложений (независимо от легкости их реализации) и сохранения их до тех пор, пока не появится возможность ими заняться. Тогда, рассматривая их целой группой, выберите из них наиболее полезные.

***Оценивайте затраты на каждое изменение*** Каждый раз, когда ваш заказчик, босс или вы сами намерены изменить систему, оценивайте время, необходимое на внесение изменения, включая рецензирование кода этого изменения и повторное тестирование всей системы. Учтите в вашей оценке расходы, касающиеся возникновения волнового эффекта от этого изменения, распространяющегося по направлению от требований к проектированию, кодированию, тестированию и обновлениям в пользовательской документации. Пусть все заинтересованные стороны знают, что ПО имеет сложнопереплетенную структуру, и что временная оценка требуется, даже если изменение кажется небольшим.

Независимо от того, как оптимистично вы оцениваете предложенное изменение, воздержитесь от немедленной оценки. Такие оценки обычно ошибочны в два и более раз.

***Относитесь с подозрением к изменениям большого объема*** Некоторое количество изменений неизбежно, но большой объем запросов на изменение сигнализирует о том, что требования, архитектура или высокоуровневое проектирование не были выполнены достаточно качественно, чтобы способствовать эффективному конструированию. Возврат к работе над требованиями или архитектурой может показаться накладным, но он гораздо дешевле, чем конструирование ПО более одного раза или выбрасывание кода тех функций системы, которые, как выяснилось, вам не нужны.

**Перекрестная ссылка** Другую точку зрения на работу с изменениями можно найти в подразделе «Что делать при изменении требований во время конструирования программы?» раздела 3.4. Советы по безопасным изменениям в коде см. в главе 24.

***Учредите комитет контроля изменений*** Работа комитета контроля изменений состоит в отделении зерен от плевел в запросах на изменение. Любой, кто хочет предложить изменение, отправляет запрос этому комитету. Термин «запрос на изменение» относится к любому запросу, который приводит к изменению ПО: идея новой функции, изменение в существующей функциональности, «отчет об ошибке», который сигнализирует о действительной или мнимой ошибке, и т. д. Комитет периодически собирается и рассматривает предложенные изменения. Он одобряет, отвергает или откладывает каждое изменение. Комитеты контроля изменений считаются лучшим решением для расстановки приоритетов и контроля изменений в требованиях, но они еще довольно редко встречаются в коммерческих структурах (Jones, 1998; Jones, 2000).

***Соблюдайте бюрократические процедуры, но не позволяйте страху перед бюрократией препятствовать эффективному контролю изменений*** Нехватка дисциплинированного контроля изменений — одна из важнейших проблем управления. Значительный процент проектов, которые считались выполненными с опозданием, на самом деле могли быть сделаны в срок, если бы в них учитывалось влияние неотслеживаемых, но согласованных изменений. Плохой контроль изменений позволяет накапливаться нерегистрируемым изменениям, что подрывает возможность видеть текущее положение дел, делать долгосрочные прогнозы, планировать выполнение работ, а также влияет на управление рисками в частности и управление проектом в целом.

Контроль изменений имеет тенденцию к бюрократизации, поэтому важно искать способы по рационализации этого процесса. Если вы предпочитаете не исполь-

зовать традиционные запросы на изменения, заведите почтовый адрес «ChangeBoard» и обяжите сотрудников посылать на него запросы на изменение. Или попросите высказывать предложения по изменениям интерактивно на заседаниях комитета контроля. Особенно действенная мера — запись запросов на изменения в качестве дефектов в систему отслеживания дефектов. Ревнители порядка могут классифицировать такие изменения как «дефекты требований», или их можно считать изменениями, а не дефектами.

Вы можете создать официальный Комитет контроля изменений, или Группу планирования продукта или Военный совет, которые будут выполнять традиционные обязанности комитета контроля изменений. Или вы можете выбрать отдельного человека в качестве Царя изменений. Но как бы вы это ни назвали, сделайте это!

Время от времени я встречаю проекты, страдающие от неумелой реализации контроля изменений. Но в 10 раз чаще я вижу проекты, страдающие от полного отсутствия вразумительного контроля изменений. Главное — это сама сущность контроля изменений, поэтому не позволяйте страху перед бюрократизацией помешать реализации преимуществ этого подхода.

## Изменения в коде программного обеспечения

Другое назначение управления конфигурацией — контроль исходного кода. Если вы изменили код и возникла новая ошибка, которая, вроде бы, никак не связана со сделанными вами правками, то при поиске ее источника, вы, возможно, захотите сравнить новую и старые версии кода. Если это ничем вам не поможет, вы, вероятно, захотите взглянуть на еще более старую версию. Совершить такой экскурс в историю просто, если у вас есть инструменты управления версиями, которые отслеживают многочисленные версии исходного кода.

***Программы управления версиями*** С хорошим ПО для управления версиями работать так легко, что вы едва замечаете его существование. Оно особенно полезно в групповых проектах. Один вариант управления версиями блокирует исходные файлы так, что только один человек может модифицировать файл в некоторый момент времени. Обычно, когда вам нужно поработать с каким-то исходным кодом, вы должны снять этот файл с учета (check out) в системе управления версиями. Если кто-то уже проделал эту процедуру, вы получите сообщение, что этого сделать нельзя. Когда вы извлечете этот файл, вы сможете с ним работать так же, как и без использования системы управления версиями до тех пор, пока не будете готовы снова зарегистрировать его (check in). Другой вариант управления версиями позволяет нескольким людям работать с файлами одновременно и следит за необходимостью слияния изменений во время регистрации файлов. В обоих случаях, когда вы регистрируете файл, система спрашивает вас, почему вы его изменили, и вы указываете причину.

При таких скромных усилиях вы получаете несколько больших преимуществ:

- вы не наступаете никому на ноги, работая с файлом в тот момент, когда с ним работает кто-то другой (или хотя бы вы знаете об этом);
- вы легко можете обновить свои копии всех файлов проекта, чтобы они соответствовали текущим версиям, обычно одной командой;

- вы можете вернуться к любой версии любого файла, когда-либо зарегистрированной в системе управления версиями;
- вы можете получить список изменений, выполненных в любой версии любого файла;
- вам не надо заботиться о персональных резервных копиях, поскольку копия в системе контроля версий служит гарантией безопасности.

Контроль версий — необходимая составляющая командных проектов. Он становится еще более мощным оружием при интеграции управления версиями, отслеживания дефектов и управления изменениями. Подразделение прикладного ПО Microsoft считает собственный инструментарий управления версиями «важнейшим преимуществом» (Moore, 1992).

## Версии инструментария

Для некоторых видов проектов может понадобиться реконструкция точной среды, используемой для разработки каждой конкретной версии ПО, включая компиляторы, компоновщики, библиотеки кода и т. д. В этом случае вам также следует поместить эти инструменты в систему управления версиями.

## Конфигурации компьютеров

Многие компании (включая мою) на своем опыте познали преимущества создания стандартизованных машинных конфигураций. На стандартной рабочей станции, содержащей необходимый инструментарий, офисные приложения и другие программы, создается образ диска. Этот образ копируется на машину каждого разработчика. Стандартная конфигурация позволяет избежать уймы проблем, связанных со слегка различающимися конфигурационными параметрами, версиями применяемых инструментов и т. п. Стандартизованный образ диска также сильно упрощает подготовку новой машины по сравнению с необходимостью устанавливать каждую программу отдельно.

## План резервного копирования

План резервного копирования — концепция не новая. Идея в том, чтобы периодически делать копию вашей работы. Если вы пишете книгу, вы не оставите рукопись на крыльце, так как ее может намочить дождь, сдуть ветер или прихватить соседская собака — почитать на сон грядущий. Вы положите свой труд в безопасное место. ПО менее осязаемо, поэтому гораздо проще забыть, что на вашей машине есть нечто очень ценное.

С компьютерными данными может произойти множество неприятностей: может повредиться жесткий диск, вы или кто-то другой можете случайно удалить самые важные файлы, рассерженный сотрудник может устроить акцию саботажа, по разным причинам (кража, наводнение, пожар) вы можете лишиться своей машины. Предпринимайте шаги для защиты своей работы. Ваш план резервного копирования должен включать периодическое создание копий и их хранение в других помещениях. Кроме исходного кода, в нем должны присутствовать все важные для вашего проекта материалы: документы, чертежи и другие записи.

Один часто забываемый аспект разработки плана резервного копирования состоит в проверке процедуры создания резервных копий. Попробуйте восстановить данные на некоторый момент времени, чтобы убедиться, что резервная копия содержит необходимую информацию и восстановленная версия работоспособна.

Завершив проект, создайте его архив. Сохраните в надежном месте копию всего: исходного кода, компиляторов, инструментов, требований, проекта, документации — всего, что может понадобиться для полного воссоздания продукта.

http://cc2e.com/2843

### Контрольный список: управление конфигурацией

**Общие вопросы**

- ❑ Разработан ли план управления конфигурацией так, чтобы помочь работе программистов и минимизировать накладные расходы?
- ❑ Лишен ли подход к SCM излишнего контроля над проектом?
- ❑ Группируются ли запросы на изменение либо с помощью неформальных средств (например, списка планируемых изменений), либо посредством более систематического подхода (такого, как комитет контроля изменений)?
- ❑ Выполняется ли систематическая оценка влияния каждого предложенного изменения на стоимость, график и качество проекта?
- ❑ Рассматриваете ли вы большие изменения как предупреждение о том, что разработка требований завершена не полностью?

**Инструментарий**

- ❑ Используется ли ПО управления версиями для облегчения управления конфигурацией?
- ❑ Используется ли ПО управления версиями для уменьшения сложностей в координации работы команды?

**Резервное копирование**

- ❑ Выполняется ли периодическое резервное копирование всех материалов проекта?
- ❑ Выполняется ли периодическое перемещение резервных копий проекта в сторонние хранилища?
- ❑ Копируются ли все материалы, включая исходный код, документы, чертежи и важные записи?
- ❑ Протестирована ли процедура восстановления из резервной копии?

## Дополнительные ресурсы по управлению конфигурацией

http://cc2e.com/2850

Поскольку эта книга посвящена конструированию, в данном разделе рассматривается управление изменениями с точки зрения конструирования. Но изменения затрагивают все уровни проекта, и всеобъемлющая стратегия управления изменениями должна делать то же самое.

Hass, Anne Mette Jonassen. *Configuration Management Principles and Practices*. Boston, MA: Addison-Wesley, 2003. Эта книга дает общую картину управления конфигурацией ПО, а также практические советы, как его внедрить в процесс разработки.

Berczuk, Stephen P. and Brad Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Boston, MA: Addison-Wesley, 2003. Как и в

предыдущей книге, здесь дается общий обзор SCM и много практической информации. Преимуществом этой книги является наличие советов, помогающих группам разработчиков изолировать и координировать свою работу.

SPMN. *Little Book of Configuration Management*. Arlington, VA: Software Program Managers Network, 1998. Эту брошюра знакомит с процессом управления конфигурацией и определяет критически важные факторы успеха. Она находится в свободном доступе на сайте SPMN по адресу *www.spmn.com/products_guidebooks.html*.

http://cc2e.com/2857

Bays, Michael. *Software Release Methodology*. Englewood Cliffs, NJ: Prentice Hall, 1999. Эта книга обсуждает управление конфигурацией ПО с акцентом на выпуске промышленной версии продукта.

Bersoff, Edward H. and Alan M. Davis. «Impacts of Life Cycle Models on Software Configuration Management». *Communications of the ACM 34*, no. 8 (August, 1991): 104–118. В этой статье описано, как влияют на SCM новые подходы к разработке ПО, особенно касающиеся создания прототипов. Статья в особенности актуальна для сред, использующих практику быстрой (agile) разработки приложений.

# 28.3. Оценка графика конструирования

Управление программным проектом — одна из исключительно трудоемких задач XXI века, причем оценка размера проекта и усилий, требуемых для его завершения, — один из сложнейших аспектов управления проектом. Большой программный проект в среднем завершается на год позже заявленного срока и на 100% превышает первоначальный бюджет (Standish Group, 1994; Jones, 1997; Johnson, 1999). При опросах программистов по поводу разницы между ожидаемым и реальным графиками выполнения проекта выяснилось, что разработчики имеют склонность к оптимистичным оценкам в пределах 20-30% (van Genuchten, 1991). Это в той же степени связано с ошибочной оценкой размера проекта и требуемых усилий, как и с плохой разработкой. В данном разделе очерчен круг вопросов, связанных с оценкой программных проектов, а также указано, где найти более подробную информацию.

## Подходы к оценке

Вы можете оценить размер проекта и усилия, требуемые для его завершения, одним из нескольких способов:

**Дополнительные сведения** Подробнее о методиках оценки графика работ см. главу 8 в «Rapid Development» (McConnell, 1996) и «Software Cost Estimation with Cocomo II» (Boehm et al., 2000).

- применить оценочное ПО;
- использовать алгоритмический подход, такой как Cocomo II, оценочная модель Барри Бома (Boehm et al., 2000);
- привлечь внешних экспертов для оценки проекта;
- обсудить предполагаемые затраты на собрании;
- оценить составные части проекта, а затем сложить их вместе;
- предложить каждому оценить их собственные задания, а затем просуммировать полученные предположения;
- применить опыт предыдущих проектов;
- сохранить предыдущие оценки и посмотреть, насколько они были аккуратны, а затем применять их для коррекции новых предположений.

Ссылки на подробную информацию об этих подходах даны в подразделе «Дополнительные ресурсы, посвященные оценке ПО» в конце этого раздела. Далее описан правильный подход к оценке проекта.

**Дополнительные сведения** Эта методика основана на рекомендациях, предложенных в «Software Engineering Economics» (Boehm, 1981).

***Определите цели*** Зачем нужна оценка? Что вы оцениваете? Только операции конструирования или весь процесс разработки? Только затраты на проект или на проект плюс отпуска, праздники, обучение и другие мероприятия, не относящиеся к проекту? Насколько аккуратной должна быть оценка, чтобы соответствовать вашим целям? Какую она должна иметь степень достоверности? Приведут ли оптимистичная и пессимистичная оценки к существенно различным результатам?

***Выделите время для оценки*** Скоропалительные оценки неаккуратны. Если вы оцениваете большой проект, рассматривайте процесс оценки как минипроект и выделите время для минипланирования оценки, чтобы хорошо ее выполнить.

**Перекрестная ссылка** О требованиях к ПО см. раздел 3.4.

***Выясните требования к программе*** Точно так же, как архитектор не может сказать, сколько будет стоить «достаточно большой» дом, так и вы не сможете надежно оценить «достаточно большой» программный проект. Бессмысленно ожидать от вас оценки объема работ, требуемых для реализации чего-то, если это «что-то» еще не определено. Определите требования или запланируйте подготовительный этап для исследований, прежде чем что-то оценивать.

***Делайте оценки на низком уровне детализации*** В зависимости от обозначенных целей основывайте оценки на подробном изучении свойств проекта. В целом чем подробней будет ваша экспертиза, тем аккуратней будет оценка. По закону больших чисел, если на одном большом интервале существует 10%-я погрешность, ошибка составляет 10% либо в сторону увеличения, либо в сторону уменьшения. На 50 небольших интервалах часть 10%-х погрешностей будет в сторону увеличения, а часть — в сторону уменьшения, и погрешности будут уравновешивать друг друга.

**Перекрестная ссылка** Сложно найти область разработки ПО, в которой итерация не имеет важного значения. Процесс предварительной оценки — один из примеров, где итерация может принести пользу. Об итеративных методиках см. раздел 34.8.

***Используйте несколько способов оценки и сравнивайте полученные результаты*** Не все методики оценки приведут к одинаковым результатам, так что попробуйте несколько. Изучите результаты, полученные разными способами. Дети быстро смекают, что, если попросить третий шарик мороженного у каждого родителя в отдельности, больше шансов услышать хотя бы одно «да», чем если спрашивать только одного родителя. Иногда родители догадываются, в чем дело, и дают одинаковый ответ, а иногда — нет. Выясните, какие варианты ответов вы можете получить при использовании разных методик оценки.

Ни один подход не является лучшим в любых обстоятельствах, а разница между ними может многое объяснить. Например, работая над первым изданием этой книги, я навскидку оценивал ее размер в 250–300 страниц. Когда я наконец выполнил углубленный расчет, предполагаемый объем оказался равен 873 страницам. «Этого не может быть», — подумал я. Поэтому я воспользовался абсолютно другим способом оценки. В результате второй попытки я получил 828 страниц.

Исходя из того, что оценки различались примерно на 5%, я пришел к заключению, что объем книги будет гораздо ближе к 850 страницам, чем к 250, и скорректировал свои планы.

***Периодически делайте повторную оценку*** После первоначальной оценки факторы, влияющие на проект, могут измениться, поэтому планируйте периодически обновлять ваши оценки. Точность ваших расчетов будет увеличиваться по мере приближения к завершению проекта (рис. 28-2). Время от времени сравнивайте реальные результаты с предполагаемыми и используйте это значение в целях уточнения расчетов для оставшейся части проекта.

http://cc2e.com/2864

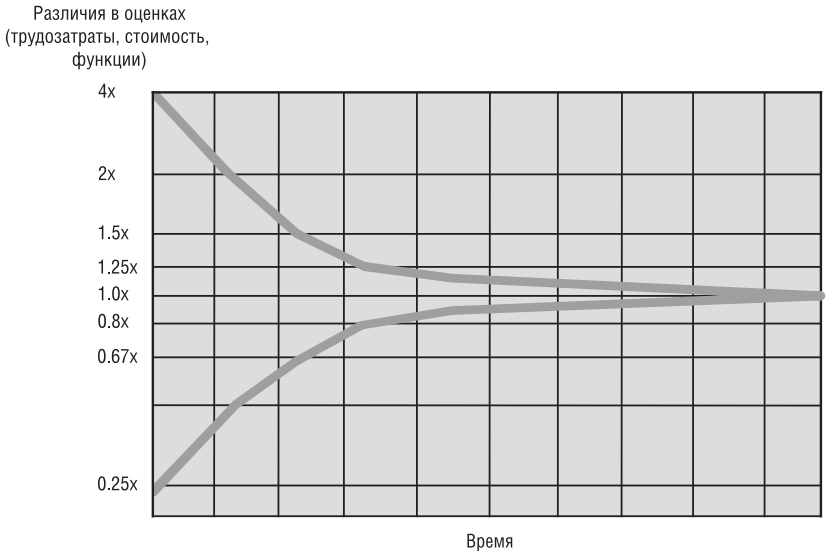



***Рис. 28-2.*** *Оценки, полученные на ранних стадиях проекта, в действительности не отличаются точностью. По мере продвижения проекта оценки могут стать более правильными. На протяжении проекта периодически делайте повторные расчеты и используйте данные, полученные во время каждого этапа, для уточнения оценки следующей операции*

## Оценка объема работ по конструированию

Степень влияния конструирования на график выполнения проекта частично зависит от того, какая доля проекта относится к конструированию, под которым понимается создание рабочего проекта, кодирование, отладка и модульное тестирование. Взгляните еще раз на рис. 27-3: пропорции меняются в зависимости от размера проекта. Пока ваша компания не создаст свою историю проектов, доля времени для каждой операции, показанная на рисунке, может стать хорошей отправной точкой для оценки ваших проектов.

**Перекрестная ссылка** Об объеме кодирования в проектах различных размеров см. в подразделе «Соотношение между выполняемыми операциями и размер» раздела 27.5.

Самый правильный ответ на вопрос, каких затрат на конструирование потребует проект, таков: пропорции будут меняться от проекта к проекту и от организации

к организации. Храните сведения об опыте проектов в вашей организации и используйте их для оценки времени, необходимого будущим проектам.

## Факторы влияния на график работ

**Перекрестная ссылка** Влияние размера программы на производительность и качество не всегда интуитивно понятно (см. главу 27).

Наибольшее влияние на график программного проекта оказывает размер создаваемой программы. Но многие другие факторы также влияют на план разработки ПО. При изучении коммерческих программ были выделены следующие факторы (табл. 28-1):

**Табл. 28-1. Факторы, влияющие на успех программного проекта**

| Фактор | Потенциально полезное влияние | Потенциально вредное влияние |
|---|---|---|
| Централизованная/распределенная разработка | –14% | 22% |
| Размер базы данных | –10% | 28% |
| Соответствие документации нуждам проекта | –19% | 23% |
| Гибкость, возможная при интерпретации требований | –9% | 10% |
| Степень активности в обслуживании рисков | –12% | 14% |
| Опыт использования языка и инструментария | –16% | 20% |
| Преемственность персонала (текучесть кадров) | –19% | 29% |
| Изменчивость платформы | –13% | 30% |
| Совершенство процесса | –13% | 15% |
| Сложность продукта | –27% | 74% |
| Способности программиста | –24% | 34% |
| Требуемая надежность | –18% | 26% |
| Способности аналитиков по изучению требований | –29% | 42% |
| Требования повторного использования | –5% | 24% |
| Соответствие приложения современным требованиям | –11% | 12% |
| Ограничения хранилища (какая часть доступного пространства будет израсходована) | 0% | 46% |
| Сплоченность команды | –10% | 11% |
| Опыт команды в данной прикладной области | –19% | 22% |
| Опыт команды в работе с данной технологической платформой | –15% | 19% |
| Временные ограничения (самого приложения) | 0% | 63% |
| Использование специализированных программных инструментов | –22% | 17% |

**Источник:** «Software Cost Estimation with Cocomo II» (Boehm et al.,. 2000).

А вот факторы, влияние которых на график разработки ПО измерить труднее; эти факторы извлечены из книг Барри Бома «Software Cost Estimation with Cocomo II» (2000) и Кейперса Джонса «Estimating Software Costs» (1998).

- опыт и способности разработчика требований;
- опыт и способности программиста;
- мотивация команды;
- качество управления;
- объем кода, использованного повторно;
- текучесть кадров;
- изменчивость требований;
- отношения с заказчиком;
- участие пользователя в разработке требований;
- опыт работы заказчика с данным типом приложений;
- степень участия программистов в разработке требований;
- обеспечение безопасности компьютера, программ и данных;
- объем документации;
- цели проекта (выполнение по графику, качество, удобство использования или какие-то другие возможные цели).

Каждый из этих факторов может оказаться важным, так что имейте их в виду наряду с перечисленными в табл. 28-1 (в нее включены некоторые из приведенных факторов).

## Оценка или контроль

Оценка — это важная часть планирования, необходимая для своевременного завершения проекта. Когда у вас есть срок поставки и спецификация продукта, то основная проблема в том, как управлять распределением человеческих и технических ресурсов для своевременной готовности продукта. С этой точки зрения, правильность начальной оценки гораздо менее важна, чем ваши последующие успехи в управлении ресурсами с целью соответствия графику.

> Важный вопрос состоит в том, что вы хотите получить: прогноз или контроль?
>
> *Том Гилб (Tom Gilb)*

## Что делать, если вы отстаете

Как я уже говорил, средний проект выбивается из запланированного графика примерно на 100%. Когда вы опаздываете, увеличить время на разработку не всегда возможно. Если сроки можно сдвинуть — сдвиньте. В противном случае вы можете принять одно или несколько из следующих решений.

***Надеяться, что вы сможете наверстать упущенное*** Обнадеживающий оптимизм — типичная реакция на отставание проекта от графика. Обычно объяснение выглядит так: «Составление требований заняло чуть больше времени, чем мы ожидали, но теперь они утверждены, и мы сможем сэкономить немного времени позже. Мы восполним недостачу при кодировании и тестировании». Ну, это вряд ли. В одном обзоре, охватывающем более 300 программных проектов, был сделан вывод, что задержки и отклонения от графика обычно увеличиваются по мере приближения к концу проекта (van Genuchten, 1991). Проекты не восполняют потерянное время позже — они отстают еще больше.

***Увеличить команду*** Согласно закону Фреда Брукса (Fred Brooks) ввод дополнительных людей на последних стадиях проекта приводит к задержке его выпол-

нения (Brooks, 1995). Это все равно, что подливать масло в огонь. Объяснение Брукса выглядит убедительно: новым людям требуется время на ознакомление с проектом. Их обучение отнимет время у обученных работников. А само увеличение числа участников увеличивает сложность и количество вариантов взаимодействий в проекте. Брукс подчеркивает: то, что одна женщина может родить ребенка через девять месяцев, не значит, что девять женщин могут родить ребенка через месяц.

Несомненно, предупреждающий закон Брукса следует принимать во внимание гораздо чаще, чем это делается. Существует тенденция бросать людей на проект и надеяться, что они выполнят его вовремя. Менеджеры должны понимать, что разрабатывать ПО не то же самое, что ковать железо: большее число работников не обязательно означает больший объем выполненной работы.

В то же время простое утверждение, что подключение программистов к работе над тормозящим проектом задерживает его еще больше, маскирует тот факт, что при некоторых обстоятельствах подобные меры способны ускорить работу. Как Брукс отмечает в анализе своего закона, подключение людей к программному проекту не поможет, если задачи в этом проекте нельзя разбить на составные части и выполнять их независимо. Но если задачи делятся на части, вы можете распределить их между разными людьми, даже недавно включившимися в работу. Другие исследователи смогли формально определить обстоятельства, при которых вы можете подключать людей к проекту на поздних стадиях и не вызывать еще большую его задержку (Abdel-Hamid, 1989; McConnell, 1999).

***Сократить проект*** О таком мощном способе, как сокращение проекта, часто забывают. Если вы исключаете какую-то функцию, вы избавляетесь от проектирования, кодирования, отладки, тестирования и документирования этой функции, а также от создания интерфейса между этой и другими функциями.

**Дополнительные сведения** Аргументы в защиту реализации только наиболее необходимой функциональности см. в главе 14 «Feature-Set Control» книги «Rapid Development» (McConnell, 1996).

При первоначальном планировании продукта разделите его возможности на категории «должны быть», «хорошо бы сделать» и «необязательные». Если вы отстаете, расставьте приоритеты в категориях «необязательные» и «хорошо бы сделать» и отбросьте те, что имеют меньшее значение.

При невозможности отмены каких-то функций вы можете представить более дешевую версию той же функциональности. Так, вы можете сдать версию вовремя, не обеспечив при этом максимальной производительности. Или в этой версии менее важная функциональность будет реализована лишь в общих чертах. Вы можете решить отбросить требования к скорости выполнения, так как медленную версию сделать проще. Или можете решить отбросить требования к объему памяти, поскольку версию, интенсивно использующую память, сделать быстрее.

Повторно оцените время разработки для менее важных функций. Какую функциональность вы можете предоставить за два часа, два дня или две недели? Какую выгоду вы получите от двухнедельной версии в отличие от двухдневной и чем двухдневная версия будет отличаться от двухчасовой?

## Дополнительные ресурсы, посвященные оценке ПО

Далее приведены ссылки на дополнительную литературу, посвященную вопросу оценки ПО.

http://cc2e.com/2871

Boehm, Barry, et al. *Software Cost Estimation with Cocomo II*. Boston, MA: Addison-Wesley, 2000. Здесь описана оценочная модель Cocomo II — несомненно, самая популярная на сегодняшний день.

Boehm, Barry W. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981. Эта более старая книга содержит исчерпывающее толкование вопросов оценки программных проектов, более общее, чем в новой книге Бома.

Humphrey, Watts S. *A Discipline for Software Engineering*. Reading, MA: Addison-Wesley, 1995. В главе 5 этой книги описывается Метод зондирования Хамфри (Humphrey's Probe method) — способ оценки объема работ с точки зрения отдельного программиста.

Conte, S. D., H. E. Dunsmore and V. Y. Shen. *Software Engineering Metrics and Models*. Menlo Park, CA: Benjamin/Cummings, 1986. Глава 6 содержит хороший обзор методик оценки, включая историю оценки, статистические модели, теоретически обоснованные модели и составные модели. Книга также демонстрирует использование каждого способа оценки для набора проектов и сравнивает полученные расчетные значения с реальной продолжительностью проектов.

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley, 1988. Назвав главу 16 «Десять принципов оценки атрибутов ПО» («Ten Principles for Estimating Software Attributes»), автор немного лукавит. Гилб возражает против оценки проектов и приводит аргументы в защиту контроля проектов. Указывая на то, что людям на самом деле нужны не точные прогнозы, а управление конечным результатом, Гилб выводит 10 принципов, которые можно использовать, чтобы заставить проект соответствовать намеченным срокам, стоимости или другим целям проекта.

# 28.4. Измерения

Программные проекты можно измерить по-разному. Далее приведены важные причины, по которым вам стоит проводить измерение процесса.

***Для любого атрибута проекта существует возможность его измерения, что в любом случае не означает отказа от его измерения*** Измерение может быть не абсолютно верным, его может быть трудно сделать и, возможно, придется временами уточнять, но измерение даст вам такой рычаг управления процессом разработки ПО, который вы не сможете получить иным способом (Gilb, 2004).

Если данные предназначены для использования в научном эксперименте, то их надо измерить. Можете ли вы представить ученого, рекомендующего запретить новый пищевой продукт, потому что группа белых крыс «кажется, чаще болеет» по сравнению с контрольной группой? Бред! Вы бы потребовали более точного ответа, например: «Крысы, которые ели новый пищевой продукт, болели на 3,7 дня дольше, чем крысы, которые его не ели». Чтобы оценить методы разработки ПО,

вы должны измерить их. Заявления типа «Этот новый метод выглядит более эффективным» недостаточно хороши.

> Что измеряется, то выполняется.
>
> *Том Питерс (Tom Peters)*

***Отдавайте себе отчет о побочных эффектах измерения*** Измерение влияет на мотивацию. Люди обращают внимание на выполняемые измерения, предполагая, что измерения служат для их оценки. Выбирайте объекты для измерения осторожно. Люди имеют склонность сосредоточиваться на работе, которая измеряется, и игнорировать остальную.

***Возражать против измерений означает утверждать, что лучше не знать о том, что на самом деле происходит*** Если вы измерите какой-то аспект проекта, вы узнаете о нем нечто, чего не знали ранее. Вы сможете увидеть, стал ли проект больше или меньше или остался таким же. Измерение предоставляет вам окно, через которое вы можете увидеть хотя бы этот аспект проекта. Окошко может быть маленьким и мутным до тех пор, пока вы не уточните свои измерения, но это все равно лучше, чем не иметь окна вообще. Возражать против любых измерений лишь потому, что некоторые из них неубедительны — все равно, что возражать против наличия окон, потому что некоторые из них могут быть мутными.

Вы можете измерить практически любой аспект процесса разработки ПО. Вот некоторые виды измерений, которые некоторые профессионалы посчитали полезными (табл. 28-2).

**Табл. 28-2. Полезные объекты для измерения в области разработки ПО**

| **Размер** | **Качество в целом** |
|---|---|
| Общее количество строк | Общее число дефектов |
| Общее количество строк комментариев | Число дефектов в каждом классе или методе |
| Общее число классов или методов | Среднее количество дефектов на тысячу строк кода |
| Общее количество объявлений данных | Среднее время между сбоями |
| Общее число пустых строк | Ошибки, выявленные компилятором |
| **Отслеживание дефектов** | **Удобство сопровождения** |
| Серьезность каждого дефекта | Число открытых методов каждого класса |
| Местонахождение каждого дефекта (класс или метод) | Число параметров, передаваемых каждому методу |
| Происхождение каждого дефекта (требования, проект, конструирование, тестирование) | Число закрытых методов и/или переменных в каждом классе |
| Способ, каким исправлялся каждый из дефектов | Число локальных переменных, используемых каждым методом |
| Лицо, ответственное за каждый дефект | Число методов, вызываемых в каждом классе или методе |
| Число строк, задействованных в исправлении каждого дефекта | Число точек принятия решений в каждом методе |
| Количество рабочих часов, потребовавшихся для исправления каждого дефекта | Сложность управляющей логики в каждом методе |
| Среднее время, требуемое для поиска дефекта | Число строк кода в каждом классе или методе |

**Табл. 28-2.** (*продолжение*)

| | |
|---|---|
| Среднее время, требуемое для исправления дефекта | Число строк комментариев в каждом классе или методе |
| Число попыток, предпринятых для исправления каждого дефекта | Количество объявлений данных в каждом классе или методе |
| Число новых ошибок, появившихся в результате исправления дефекта | Число пустых строк в каждом классе или методе |
| | Количество операторов *goto* в каждом классе или методе |
| | Количество операторов ввода или вывода в каждом классе или методе |
| **Производительность** | |
| Количество человеко-часов, затраченных на проект | |
| Количество человеко-часов, потраченных на каждый класс или метод | |
| Количество изменений каждого класса или метода | |
| Сумма в долларах, потраченная на проект | |
| Сумма в долларах, потраченная на строку кода | |
| Сумма в долларах, потраченная на каждый дефект | |

Вы можете получить результаты для большинства этих измерений, используя современные программные средства. На протяжении всей книги обсуждались причины, по которым то или иное измерение может быть полезно. В настоящее время большинство этих измерений нужно не для поиска явных различий между программами, классами и методами (Shepperd and Ince, 1989). Они нужны в основном для выявления методов, которые резко отличаются от других; необычные показатели измерений в каком-то методе предупреждают о том, что вам следует пересмотреть этот метод, дабы выяснить причину необычно низкого качества.

Не начинайте сбор данных для всех возможных измерений — вы закопаетесь в таких сложных данных, что не сможете выяснить, что они означают. Начните с простого набора измерений, скажем, с количества дефектов, человеко-месяцев, общей суммы в долларах и общего количества строк кода. Стандартизуйте эти измерения во всех своих проектах, а затем уточняйте их показатели и добавляйте к ним новые, по мере того как ваше понимание необходимых измерений улучшается (Pietrasanta, 1990).

Убедитесь, что вы знаете причину, по которой собираете данные. Установите цели, определите вопросы, которые необходимо задать, чтобы добиться этих целей, а затем проводите измерения, чтобы узнать ответы (Basili and Weiss, 1984). Убедитесь, что существует возможность получить ту информацию, которую вы запрашиваете, и не забывайте, что сбор данных всегда играет второстепенную роль по сравнению со сроками сдачи проектов (Basili et al., 2002).

## Дополнительные ресурсы, касающиеся измерения ПО

http://cc2e.com/2878

Oman, Paul and Shari Lawrence Pfleeger, eds. *Applying Software Metrics*. Los Alamitos, CA: IEEE Computer Society Press, 1996. Под обложкой этого тома собрано более 25 ключевых статей, посвященных измерению ПО.

Jones, Capers. *Applied Software Measurement: Assuring Productivity and Quality*, 2d ed. New York, NY: McGraw-Hill, 1997. Джонс — лидер в области измерения ПО, и его книга аккумулирует все знания в этой области. В ней представлена наиболее полная теория и практика существующих методик измерения и описаны проблемы, возникающие при использовании традиционных способов измерения. В книге приведена полная программа по сбору числа реализуемых функций. Джонс собрал и проанализировал огромный объем данных, касающийся качества и производительности, и его труд содержит выжимку этих результатов, в том числе и захватывающую главу, посвященную средним значениям показателей в области разработки ПО в США.

Grady, Robert B. *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs, NJ: Prentice Hall PTR, 1992. Гради рассказывает о внедрении программы по измерению ПО в Hewlett-Packard и о том, как внедрить такую программу в вашей организации.

Conte, S. D., H. E. Dunsmore and V. Y. Shen. *Software Engineering Metrics and Models*. Menlo Park, CA: Benjamin/Cummings, 1986. Эта книга классифицирует существующие знания об измерениях ПО по состоянию на 1986 год, включая часто используемые измерения, экспериментальные методики и критерии оценки результатов эксперимента.

Basili, Victor R., et al. 2002. «Lessons learned from 25 years of process improvement: The Rise and Fall of the NASA Software Engineering Laboratory». *Proceedings of the 24th International Conference on Software Engineering*. Orlando, FL, 2002. В статье освещен опыт, полученный организацией, разрабатывающей одни из самых сложных программных продуктов в мире. В центре внимания — вопросы измерения.

http://cc2e.com/2892

NASA Software Engineering Laboratory. *Software Measurement Guidebook*, June 1995, NASA-GB-001-94. Это 100-страничное руководство возможно, лучший источник практической информации о том, как настроить и использовать измерительную программу. Его можно загрузить с Web-сайта NASA.

http://cc2e.com/2899

Gilb, Tom. *Competitive Engineering*. Boston, MA: Addison-Wesley, 2004. Книга представляет ориентированный на измерения подход к определению требований, разработке проекта, измерению качества и управлению проектом в целом. Ее можно загрузить с вебсайта автора.

# 28.5. Гуманное отношение к программистам

Атмосфера абстрактности в деятельности программистов требует создания непринужденной обстановки в офисе и поддержания широких контактов между коллегами. Высокотехнологичные компании предлагают пар-

ковые корпоративные кампусы, органичные организационные структуры, комфортабельные офисы и прочие удобства, чтобы компенсировать интенсивную, а порой и скучную интеллектуальность самой работы. Наиболее успешные компании сочетают элементы высоких технологий и бережного отношения к людям (Naisbitt, 1982). В этом разделе программисты рассматриваются как нечто большее, чем органическое отражение их силиконовых alter ego.

## Как программисты проводят свое время?

Программисты большую часть рабочего времени программируют, но, кроме того, они тратят его на заседания, обучение, чтение почты и просто размышления. Исследование, проведенное в 1964 году в Bell Laboratories, показало, что программисты тратят время так (табл. 28-3):

**Table 28-3. Одно из представлений того, как программисты тратят свое время**

| Деятельность | Исходный Код | Деловые вопросы | Личные дела | Заседания | Обучение | Почта и др. документы | Технические руководства | Обслуживающие процедуры, Разное | Тестирование программ | Итого |
|---|---|---|---|---|---|---|---|---|---|---|
| Говорят или слушают | 4% | 17% | 7% | 3% | | | | 1% | | 32% |
| Говорят с менеджером | 1% | | | | | | | | 1% | |
| Говорят по телефону | 2% | 1% | | | | | | | 3% | |
| Читают | 14% | | | | | 2% | 2% | | | 18% |
| Пишут | 13% | | | | | 1% | | | | 14% |
| Отсутствуют | 4% | 1% | 4% | 6% | | | | | 15% | |
| Гуляют | 2% | 2% | 1% | | | 1% | | | | 6% |
| Разное | 2% | 3% | 3% | | | 1% | | 1% | 1% | 11% |
| Итого | 35% | 29% | 13% | 7% | 6% | 5% | 2% | 2% | 1% | 100% |

**Источник:** «Research Studies of Programmers and Programming» (Bairdain, 1964, приведено в Boehm, 1981).

Эти данные основаны на изучении времяпрепровождения 70 программистов. Данные старые, и пропорции времени, которое тратится на различные виды деятельности, у разных программистов будут иными. И все же результаты наводят на размышления. Около 30% времени тратится на нетехнические виды деятельности: прогулки, личные вопросы и т. д. Программисты в этом исследовании тратят 6% времени на прогулки — это примерно 2,5 часа в неделю и 125 часов в год. Это может показаться несущественным, пока вы не поймете, что программисты каждый год тратят на прогулки столько же времени, сколько на обучение, втрое больше, чем на чтение технической документации, и в шесть раз больше, чем ний этого показателя за прошедшие годы.

## Различия в производительности и качестве

Способности и прилагаемые усилия у разных программистов очень разные, как, впрочем, и в других сферах человеческой деятельности. Исследования показали, что в различных профессиях (писательство, футбол, изобретательство, полицейская работа и пилотирование самолетов) первые 20% людей производят примерно 50% результатов (Augustine, 1979). Выводы этого исследования базируются на анализе данных производительности, таких как забитые голы, патенты, решенные задачи и т. д. Так как некоторые люди вообще не вносят никакого вклада и они не учитывались в исследовании (защитники, не забивающие голы, изобретатели, не имеющие собственных патентов, детективы, не раскрывавшие преступлений, и т. д.), то данные, возможно, преуменьшают фактическую разницу в производительности.

Конкретно в программировании многие исследования показывают разницу на порядки в качестве написанных программ, их размерах и в производительности программистов.

### Индивидуальные различия

Первоначальное исследование, показавшее огромные различия в производительности отдельных программистов, было проведено в конце 1960-х Секменом, Эриксоном и Грантом (Sackman, Erikson, Grant, 1968). Они изучали профессиональных программистов с примерно 7-летним стажем и выяснили, что соотношение первоначального времени кодирования между лучшим и худшим программистами — примерно 20:1, соотношение времени отладки — более, чем 25:1, соотношение размера программы — 5:1, а соотношение скорости выполнения — 10:1.Они не нашли взаимосвязи между опытом программиста и качеством кода или производительностью.

Хотя такие определенные значения соотношений, как 25:1, не имеют особого смысла, более общие утверждения, скажем: «Программисты различаются между собой на порядки,» — достаточно содержательны и подтверждаются другими исследованиями профессиональных программистов (Curtis, 1981; Mills, 1983; DeMarco and Lister, 1985; Curtis et al., 1986; Card, 1987; Boehm and Papaccio, 1988; Valett and McGarry, 1989; Boehm et al., 2000).

### Командные различия

Команды программистов также показывают заметные различия в качестве ПО и производительности. Хорошие программисты стремятся к объединению, так же как и плохие программисты. Это наблюдение подтверждается исследованием 166 профессиональных программистов из 18 организаций (Demarco and Lister, 1999).

В одном исследовании семи идентичных проектов потраченные усилия различались в 3,4 раза, а размеры программ — в 3 раза (Boehm, Gray and Seewaldt, 1984). Несмотря на такую разницу в производительности, программистов в этом исследовании нельзя отнести к разным группам. Все они были профессионалами с несколькими годами стажа, окончившими учебные заведения по специальности, связанной с вычислительной техникой. Можно предположить, что исследование менее гомогенной группы показало бы еще большие различия.

Более раннее исследование программистских команд показывало соотношение 5:1 в размерах программ и 2,6:1 во времени, необходимом команде для завершения одного и того же проекта (Weinberg and Schulman, 1974).

После обработки данных, полученных более чем за 20 лет, для разработки оценочной модели Cocomo II, Барри Бом и другие исследователи пришли к выводу, что разработка программы с участием первых 15% программистов, отсортированных по способностям, обычно требует примерно в 3,5 раз больше человеко-месяцев, чем разработка программы с участием 90% программистов (Boehm et al., 2000). Бом и другие исследователи выяснили, что 80% работы выполняют 20% сотрудников (Boehm, 1987b).

Выводы, с точки зрения найма персонала, очевидны. Если вам приходится платить больше, чтобы заполучить программиста из первых 10%, а не программиста из последних 10%, не упускайте этот шанс. Вы получите мгновенное вознаграждение в качестве и производительности нанятого вами программиста, а кроме того, вы получите остаточный эффект в качестве и производительности остальных программистов, которых ваша организация может нанять, так как хорошие программисты стремятся к объединению.

## Вопросы религии

Менеджеры программных проектов не всегда осознают, что некоторые аспекты программирования сродни религиозным вопросам. Если вы менеджер и пытаетесь требовать исполнения определенных правил программирования, вы рискуете вызвать гнев ваших подопечных. Вот список таких «религиозных» вопросов:

- язык программирования;
- стиль отступов;
- размещение скобок;
- выбор среды разработки (IDE);
- стиль комментариев;
- компромиссы между эффективностью и читабельностью;
- выбор методологии — например, между методом Scrum, экстремальным программированием или эволюционной разработкой;
- программные утилиты;
- соглашения по именованию;
- применение *goto*;
- использование глобальных переменных;
- измерения, особенно меры производительности, например, количество строк кода в день.

Общий знаменатель у всех этих вопросов в том, что позиция программиста по каждому из них отражает его личный стиль. Если вы считаете, что должны контролировать программиста в каких-то из этих «религиозных» сфер, учтите следующее.

***Вы вторгаетесь в область, требующую деликатного обращения*** Разузнайте мнения программистов по поводу каждого эмоционального вопроса, прежде чем окунуться в него с головой.

***Из уважения к теме используйте термины «предложения» и «советы»*** Избегайте установки жестких «правил» и «стандартов».

***Старайтесь обходить спорные вопросы, предпочитая давать явные поручения*** В выборе стиля отступов или размещения скобок поможет такая хитрость; потребуйте прогнать код через средства создания «красивой» печати, прежде чем объявлять его законченным. Пусть форматирование выполняется специальными средствами. Чтобы решить вопрос со стилем комментариев, требуйте, чтобы весь код рецензировался и неочевидный код исправлялся до тех пор, пока не станет ясным.

***Предложите программистам выработать собственные стандарты*** Как я уже говорил, детали конкретного стандарта зачастую менее важны, чем факт самого существования стандарта. Не устанавливайте стандарты вашим программистам, но настаивайте на том, чтобы они стандартизовали области, важные для вас.

Какие из «религиозных» тем настолько важны, чтобы из-за них ввязываться в спор? Подчинение в небольших вопросах стиля в любой области, возможно, не принесет особой выгоды по сравнению с ухудшением моральной атмосферы в команде. Если вы встречаете беспорядочное использование операторов *goto* или глобальных переменных, нечитаемый стиль или другие признаки, влияющие на проект в целом, то для улучшения качества кода будьте готовы мириться с некоторыми разногласиями. Если ваши программисты добросовестны, это редко становится проблемой. Самые большие сражения обычно разворачиваются вокруг стилистических нюансов кодирования, и вы можете стоять в стороне от этого без всяких потерь для проекта.

## Физическая среда

Проведите эксперимент: поезжайте за город, найдите ферму, отыщите фермера и спросите его, во что ему обошлось оборудование. Фермер посмотрит в амбар и увидит несколько тракторов, фургонов, зерновой комбайн, молотилку, и скажет, что сумма превышает $100 000 на работника.

После этого найдите в городе фирму, занимающуюся разработкой ПО, отыщите менеджера и спросите его, каковы его затраты на оборудование. Менеджер заглянет в офис, увидит стол, стул, несколько книг и компьютер и скажет, что они не превышают $25 000 на каждого работника.

Физическая среда сильно влияет на производительность. Демарко и Листер (DeMarco and Lister) опросили 166 программистов из 35 организаций о качестве их физического окружения. Большинство работников оценило свои рабочие места как неприемлемые. После проведения соревнования по программированию оказалось, что программисты, результаты которых попадают в первые 25%, имеют более просторные, тихие и уединенные кабинеты, а также реже отвлекаются на других людей и телефонные звонки. Вот сводка различий в офисном пространстве между лучшими и худшими программистами:

| Фактор среды | Первые 25% | Последние 25% |
|---|---|---|
| Офисная площадь | 9 м² | 5 м² |
| Достаточно тихое рабочее место | 57% «да» | 29% «да» |
| Достаточно уединенное место | 62% «да» | 19% «да» |
| Возможность выключить телефон | 52% «да» | 10% «да» |
| Возможность переадресовать звонки | 76% «да» | 19% «да» |
| Частые ненужные прерывания | 38% «да» | 76% «да» |
| Рабочее место, позволяющее программистам чувствовать себя оцененным по достоинству | 57% «да» | 29% «да» |

**Источник:** «Peopleware» (DeMarco and Lister, 1999).

Эти данные показывают сильную корреляцию между производительностью и качеством рабочего места. Программисты из первых 25% были в 2,6 раза производительнее, чем программисты из последних 25%. Демарко и Листер подумали, что более квалифицированным программистам могли быть предоставлены лучшие условия в качестве поощрения, но дальнейшая проверка показала, что это не так. Программисты из одной организации имели сходные удобства независимо от разницы в их производительности.

Большие организации, нацеленные на разработку ПО, подтверждают эти данные. Xerox, TRW, IBM и Bell Labs отмечают, что они получили значительный прирост в производительности, увеличив инвестиции с $10 000 до $30 000 на одного сотрудника. Возросшая производительность неоднократно компенсировала эти затраты (Boehm, 1987a). По собственным оценкам рост производительности в таких «производительных офисах» составил от 39 до 47% (Boehm et al., 1984).

Подводя итоги, можно сказать, что если ваше рабочее место попадает в худшие 25%, вы можете увеличить свою производительность примерно на 100%, улучшив его до соответствия первым 25%. Если у вашего рабочего места средние показатели, вы все еще можете получить 40%-е увеличение производительности, улучшив его качество до первых 25%.

## Дополнительные ресурсы, посвященные программистам как человеческим существам

http://cc2e.com/2806

Weinberg, Gerald M. *The Psychology of Computer Programming*, 2d ed. New York, NY: Van Nostrand Reinhold, 1998. Это первая книга, которая явно определяет программистов как людей, и в ней лучше всего рассматривается программирование как человеческая деятельность. Она наполнена проницательными замечаниями о человеческой природе программистов и ее последствиях.

DeMarco, Tom and Timothy Lister. *Peopleware: Productive Projects and Teams*, 2d ed. New York, NY: Dorset House, 1999. Как сказано в названии, эта книга также рассматривает человеческий фактор в программировании. Она содержит множество анекдотов о менеджерах, офисном окружении, найме и развитии нужных людей, увеличении команд и любви к своей работе. Авторы переусердствовали с анекдотами для поддержки некоторых необычных точек зрения, и их логика порой не-

убедительна, но важен душевный настрой книги, ставящий людей на первое место, и авторам, несомненно, удалось донести свою мысль.

http://cc2e.com/2820

McCue, Gerald M. «IBM's Santa Teresa Laboratory — Architectural Design for Program Development», *IBM Systems Journal* 17, no. 1 (1978): 4–25. Маккью рассказывает о том, как в IBM создавали офисный комплекс в Санта Тереза. IBM изучила нужды программистов и разработала здания с учетом их пожеланий. Программисты принимали участие в проекте на всем его протяжении. Результат: в ежегодных опросах мнений комплекс в Санта Тереза оценивается в компании как лучший.

McConnell, Steve. *Professional Software Development*. Boston, MA: Addison-Wesley, 2004. В главе 7 «Orphans Preferred» подводятся итоги демографических исследований в среде программистов, включая типы личностей, образование и перспективы работы.

Carnegie, Dale. *How to Win Friends and Influence People*, Revised Edition. New York, NY: Pocket Books, 1981. Написав заголовок для первого издания этой книги в 1936 году, Дейл Карнеги не мог подозревать, какой скрытый смысл он приобретет сегодня. Он звучит так, как будто книга взята с полки самого Макиавелли. Однако дух книги диаметрально противоположен манипуляциям в стиле Макиавелли, и один из ключевых моментов говорит о важности развития неподдельного интереса к другим людям. Карнеги глубоко проникает в ежедневные взаимоотношения и объясняет, как работать с другими людьми с помощью лучшего их понимания. Книга полна запоминающихся историй, иногда по две-три на страницу. Любой, кто работает с людьми, должен когда-нибудь прочесть ее, а тот, кто управляет людьми, должен прочесть ее *немедленно*.

## 28.6. Управление менеджером

В области разработки ПО часто встречаются как нетехнические менеджеры, так и такие, кто имеет технический опыт, но отстал от жизни лет на 10. Технически компетентные, технически современные менеджеры редки. Если вы работаете с таким, делайте все, чтобы сохранить свою работу. Это необычайное везение.

> В иерархии каждый работник стремится подняться до своего уровня некомпетенции.
>
> *Принцип Питера (The Peter Principle)*

Если ваш менеджер более типичен, вы сталкиваетесь с незавидной задачей управления собственным менеджером. «Управление менеджером» означает, что вам нужно объяснить менеджеру, что надо делать, а не наоборот. Фокус в том, что это надо сделать так, чтобы он продолжал считать, что это он вами управляет. Вот несколько подходов к работе с вашим менеджером:

- посейте идеи того, что вы хотите сделать, а затем дождитесь, пока вашего менеджера посетит мысль, что вам нужно делать именно то, что вы хотите;
- просвещайте вашего менеджера, как делать правильно; это непрерывная работа, потому что менеджеров часто повышают, переводят или увольняют;
- сосредоточьтесь на интересах менеджера, делая то, что он или она действительно от вас хочет, и не отвлекайте его внимание несущественными деталями реализации (думайте об этом, как об «инкапсуляции» вашей работы);

- откажитесь делать то, что говорит ваш менеджер, настаивайте на выполнении работы правильным образом;
- найдите другую работу.

Наилучшим долгосрочным решением будет попытка просветить вашего менеджера. Это не всегда легкая задача, но одним из способов подготовиться к ней будет чтение книги Дейла Карнеги «How to Win Friends and Influence People».

## Дополнительные ресурсы по управлению конструированием

Следующие книги освещают общие вопросы управления программными проектами.

http://cc2e.com/2813

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley, 1988. Гилб прокладывает собственный курс на протяжении 30 лет, и большую часть времени он обгоняет остальных независимо от того, отдают ли они себе в этом отчет. Эта книга — хороший тому пример. Она была одной из первых работ, в которых обсуждались эволюционные методики разработки, управление рисками и применение формальных проверок. Гилб хорошо осведомлен о наиболее передовых подходах; действительно, эта книга, опубликованная более 15 лет назад, содержит большинство хороших методик, которые преподносятся в настоящее время в качестве быстрой (agile) разработки. Гилб исключительно практичен, и его книга до сих пор является одной из лучших в сфере управления ПО.

McConnell, Steve. *Rapid Development*. Redmond, WA: Microsoft Press, 1996. Эта книга охватывает вопросы руководства и управления проектами, на выполнение которых остро не хватает времени, что, по моему опыту, относится к большинству проектов.

Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition* (2d ed). Reading, MA: Addison-Wesley, 1995. Эта книга — смесь метафор и фольклора, связанного с управлением программными проектами. Она увлекательна и может пролить свет на содержимое ваших собственных проектов. В ее основе лежит рассказ о трудностях, с которыми сталкивался Брукс при разработке операционной системы OS/360, что позволяет мне сделать несколько замечаний. Она полна советов наподобие «Мы сделали так, и это не сработало» и «Нам следовало сделать вот так, потому что тогда бы это работало». Наблюдения Брукса по поводу неудачных методик хорошо обоснованы, но его заявления, что другие технологии были бы удачны, слишком гипотетичны. Читайте эту книгу критически, чтобы отделять наблюдения от умозаключений. Это предупреждение не умаляет значимости книги. Ее до сих пор чаще других цитируют в компьютерной литературе, и, хотя впервые она была опубликована в 1975 году, до сих пор кажется актуальной. Ее сложно читать без того, чтобы не восклицать «Точно!» каждые пару страниц.

## Соответствующие стандарты

*IEEE Std 1058-1998*, Standard for Software Project Management Plans.

*IEEE Std 12207-1997*, Information Technology — Software Life Cycle Processes.

*IEEE Std 1045-1992*, Standard for Software Productivity Metrics.

*IEEE Std 1062-1998*, Recommended Practice for Software Acquisition.

*IEEE Std 1540-2001*, Standard for Software Life Cycle Processes — Risk Management.

*IEEE Std 828-1998*, Standard for Software Configuration Management Plans

*IEEE Std 1490-1998*, Guide — Adoption of PMI Standard — A Guide to the Project Management Body of Knowledge.

## Ключевые моменты

- Хорошая практика кодирования может быть достигнута путем внедрения стандартов или более ловкими способами.
- Управление конфигурацией при правильном применении делает работу программистов проще. Это особенно касается контроля изменений.
- Правильная оценка ПО — сложная задача. Ключ к успеху — использование нескольких подходов, уточнение оценок по мере продвижения проекта и применение накопленных данных при выполнении оценки.
- Измерения — это ключ к успешному управлению конструированием. Вы всегда сможете найти способы измерить любой аспект проекта, что будет лучше, чем полное отсутствие измерений. Точное измерение — ключ к точному составлению плана, контролю качества и улучшению процесса разработки.
- Программисты и менеджеры — в первую очередь люди, и они лучше всего работают тогда, когда к ним относятся по-человечески.

ГЛАВА 29

# Интеграция

**Содержание**



http://cc2e.com/2985

**Связанные темы**

- Тестирование во время разработки: глава 22
- Отладка: глава 23
- Управление конструированием: глава 28

Термин «интеграция» относится к такой операции в процессе разработки ПО, при которой вы объединяете отдельные программные компоненты в единую систему. В небольших проектах интеграция может занять одно утро и заключаться в объединении горстки классов. В больших — могут потребоваться недели или месяцы, чтобы связать воедино весь набор программ. Независимо от размера задач в них применяются одни и те же принципы.

Тема интеграции тесно переплетается с вопросом последовательности конструирования. Порядок, в котором вы создаете классы или компоненты, влияет на порядок их интеграции: вы не можете интегрировать то, что еще не было создано. Последовательности интеграции и конструирования имеют большое значение. В этой главе мы рассмотрим оба вопроса с точки зрения интеграции.

## 29.1. Важность выбора подхода к интеграции

В инженерных отраслях, отличных от разработки ПО, важность правильной интеграции хорошо известна. На северо-западном побережье Тихого океана, где я живу, столкнулись с драматическими последствиями плохой интеграции, когда футбольный стадион Университета Вашингтон частично обрушился во время строительства (рис. 29-1).

***Рис. 29-1.*** *Навес над футбольным стадионом Вашингтонского университета обрушился, не выдержав собственного веса во время строительства. Он скорее всего был бы достаточно прочным после завершения работ, но строительство велось в неправильном порядке — налицо ошибка интеграции*

Не имеет значения, что после завершения строительства стадион был бы достаточно прочным — он должен был быть таким и на других этапах работы. Если вы создаете и интегрируете ПО в неправильном порядке, его сложнее кодировать, сложнее тестировать и сложнее отлаживать. Если ничего не работает до того, как заработает все вместе, можно предположить, что проект никогда не будет завершен. Он также может рухнуть под собственным весом во время конструирования: количество дефектов может оказаться непреодолимым, прогресс — невидимым, а сложность — чрезмерной, даже если законченный продукт был бы вполне работоспособен.

Поскольку интеграция выполняется после того, как разработчик завершил свое тестирование, и одновременно с системным тестированием, ее иногда считают операцией, относящейся к тестированию. Однако она достаточно сложна, и поэтому ее следует рассматривать как независимый вид деятельности.

Аккуратная интеграция обеспечивает:

- упрощенную диагностику дефектов;
- меньшее число ошибок;
- меньшее количество «лесов»;
- раннее создание первой работающей версии продукта;
- уменьшение общего времени разработки;
- лучшие отношения с заказчиком;
- улучшение морального климата;
- увеличение шансов завершения проекта;
- более надежные оценки графика проекта;
- более аккуратные отчеты о состоянии;

- лучшее качество кода;
- меньшее количество документации.

Интеграцию часто недооценивают, несмотря на ее важность, и именно поэтому я посвятил ей отдельную главу.

## 29.2. Частота интеграции — поэтапная или инкрементная?

Интеграция программ выполняется посредством поэтапного или инкрементного подхода.

### Поэтапная интеграция

За исключением последних нескольких лет поэтапная интеграция была нормой. Она состояла из хорошо определенных этапов, перечисленных ниже.

1. «Модульная разработка»: проектирование, кодирование, тестирование и отладка каждого класса.
2. «Системная интеграция»: объединение классов в одну огромную систему.
3. «Системная дезинтеграция» [спасибо Мейлиру Пейдж-Джонсу (Meilir Page-Jones) за это остроумное замечание]: тестирование и отладка всей системы.

Проблема поэтапной интеграции в том, что, когда классы в системе впервые соединяются вместе, неизбежно возникают новые проблемы и их причины могут быть в чем угодно. Поскольку у вас масса классов, которые никогда раньше не работали вместе, виновником может быть плохо протестированный класс, ошибка в интерфейсе между двумя классами или ошибка, вызванная взаимодействием двух классов. Все классы находятся под подозрением.

Неопределенность местонахождения любой из проблем сочетается с тем фактом, что все эти проблемы вдруг проявляют себя одновременно. Это заставляет вас иметь дело не только с проблемами, вызванными взаимодействием классов, но и другими ошибками, которые трудно диагностировать, так как они взаимодействуют. Поэтому поэтапную интеграцию называют еще «интеграцией большого взрыва» (рис. 29-2).

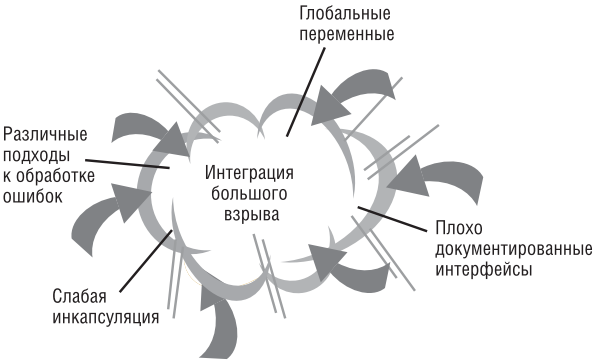



***Рис. 29-2.*** *Поэтапную интеграцию также называют интеграцией «большого взрыва» — и заслуженно!*

Поэтапную интеграцию нельзя начинать до начала последних стадий проекта, когда будут разработаны и протестированы все классы. Когда классы наконец будут объединены и проявится большое число ошибок, программисты тут же ударятся в паническую отладку вместо методического определения и исправления ошибок.

Для небольших программ — нет, а для крошечных — поэтапная интеграция может быть наилучшим подходом. Если программа состоит из двух-трех классов, поэтапная интеграция может сэкономить ваше время, если вам повезет. Но в большинстве случаев другой подход будет лучше.

## Инкрементная интеграция

**Перекрестная ссылка** О метафорах, подходящих для инкрементной интеграции, см. подразделы «Метафора жемчужины: медленное приращение системы» и «Строительная метафора: построение ПО» раздела 2.3.

При инкрементной интеграции вы пишете и тестируете маленькие участки программы, а затем комбинируете эти кусочки друг с другом по одному. При таком подходе — по одному элементу за раз — вы выполняете перечисленные далее действия.

1. Разрабатываете небольшую, функциональную часть системы. Это может быть наименьшая функциональная часть, самая сложная часть, основная часть или их комбинация. Тщательно тестируете и отлаживаете ее. Она послужит скелетом, на котором будут наращиваться мускулы, нервы и кожа, составляющие остальные части системы.
2. Проектируете, кодируете, тестируете и отлаживаете класс.
3. Прикрепляете новый класс к скелету. Тестируете и отлаживаете соединение скелета и нового класса. Убеждаетесь, что эта комбинация работает, прежде чем переходить к добавлению нового класса. Если дело сделано, повторяете процесс, начиная с п. 2.

У вас может возникнуть желание интегрировать большие модули, чем отдельный класс. Например, если компонент был тщательно протестирован и каждый из его классов прошел мини-интеграцию, вы можете интегрировать весь компонент, и это все еще будет инкрементная интеграция. По мере того, как вы добавляете новые куски, система разрастается и ускоряется, как разрастается и ускоряется снежный ком, катящийся с горы (рис. 29-3).


***Рис. 29-3.*** *Инкрементная интеграция дает проекту движущую силу и напоминает снежный ком, катящийся с горы*

## Преимущества инкрементной интеграции

Инкрементный подход имеет массу преимуществ перед традиционным поэтапным подходом независимо от того, какую инкрементную стратегию вы используете:

***Ошибки можно легко обнаружить*** Когда во время инкрементной интеграции возникает новая проблема, то очевидно, что к этому причастен новый класс. Либо его интерфейс с остальной частью программы неправилен, либо его взаимодействие с ранее интегрированными классами приводит к ошибке. В любом случае вы точно знаете, где искать проблему (рис. 29-4). Более того, поскольку вы сталкиваетесь с меньшим числом проблем одновременно, вы уменьшаете риск того, что несколько ошибок будут взаимодействовать или маскировать друг друга. Чем больше интерфейсных ошибок может возникнуть, тем больше преимуществ от инкрементной интеграции получат ваши проекты. Учет ошибок в одном проекте показал, что 39% составляли ошибки межмодульных интерфейсов (Basili и Perricone, 1984). Поскольку разработчики многих проектов проводят до 50% времени за отладкой, максимизация эффективности отладки путем упрощения поиска ошибок дает выигрыш в качестве и производительности.

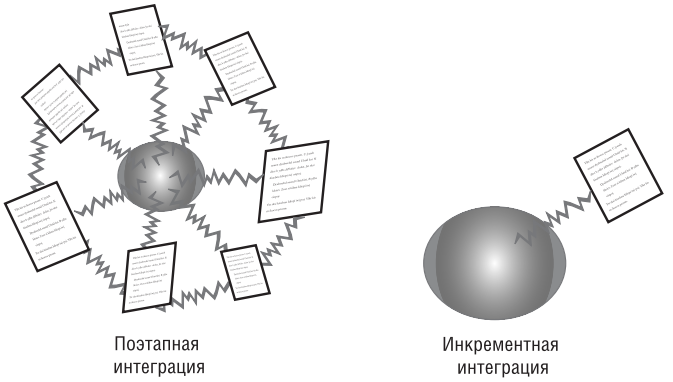



***Рис. 29-4.*** *При поэтапной интеграции вы объединяете так много компонентов одновременно, что тяжело понять, где находится ошибка. Она может быть в любом компоненте или их соединениях. При инкрементной интеграции ошибка обычно таится либо в новом компоненте, либо в месте соединения нового компонента и остальной системы*

***В таком проекте система раньше становится работоспособной*** Когда код интегрирован и способен выполняться, даже если система еще не пригодна к использованию, это выглядит так, будто это скоро произойдет. При инкрементной интеграции программисты раньше видят результаты своей работы, поэтому их моральное состояние лучше, чем в том случае, когда они подозревают, что их проект может никогда не сделать первый вдох.

***Вы получаете улучшенный мониторинг состояния*** При частой интеграции реализованная и нереализованная функциональность видна с первого взгляда. Менеджеры будут иметь лучшее представление о состоянии проекта, видя, что 50% системы уже работает, а не слыша, что кодирование «завершено на 99%».

***Вы улучшите отношения с заказчиком*** Если частая интеграция влияет на моральное состояние разработчиков, то она также оказывает влияние и на моральное состояние заказчика. Клиенты любят видеть признаки прогресса, а инкрементная интеграция предоставляет им такую возможность достаточно часто.

***Системные модули тестируются гораздо полнее*** Интеграция начинается на ранних стадиях проекта. Вы интегрируете каждый класс по мере его готовности, а не ожидая одного внушительного мероприятия по интеграции в конце разработки. Программист тестирует классы в обоих случаях, но в качестве элемента общей системы они используются гораздо чаще при инкрементной, чем при поэтапной интеграции.

***Вы можете создать систему за более короткое время*** Если интеграция тщательно спланирована, вы можете проектировать одну часть системы в то время, когда другая часть уже кодируется. Это не уменьшает общее число человеко-часов, требуемых для полного проектирования и кодирования, но позволяет выполнять часть работ параллельно, что является преимуществом в тех случаях, когда время имеет критическое значение.

Инкрементная интеграция поддерживает и поощряет другие инкрементные стратегии. Преимущества инкрементного подхода в отношении интеграции — лишь верхушка айсберга.

# 29.3. Стратегии инкрементной интеграции

При поэтапной интеграции вам не нужно планировать порядок создания компонентов проекта. Все компоненты интегрируются одновременно, поэтому вы можете разрабатывать их в любом порядке — главное, чтобы они все были готовы к часу X.

При инкрементной интеграции вы должны планировать более аккуратно. Большинство систем требует интеграции некоторых компонентов перед интеграцией других. Так что планирование интеграции влияет на планирование конструирования — порядок, в котором конструируются компоненты, должен обеспечивать порядок, в котором они будут интегрироваться.

Стратегии, определяющие порядок интеграции, различны по форме и размерам, и ни одна из них не будет являться оптимальной во всех случаях. Наилучший подход к интеграции меняется от проекта к проекту, и лучшим решением будет то, что будет соответствовать конкретным требованиям конкретного проекта. Знание делений на методологической шкале поможет вам представить варианты возможных решений.

## Нисходящая интеграция

При нисходящей интеграции класс на вершине иерархии пишется и интегрируется первым. Вершина иерархии — это главное окно, управляющий цикл приложения, объект, содержащий метод *main()* в программе на Java, функция *WinMain()* в программировании для Microsoft Windows или аналогичные. Для работы этого верхнего класса пишутся заглушки. Затем, по мере интеграции классов сверху вниз, классы заглушек заменяются реальными (рис. 29-5).

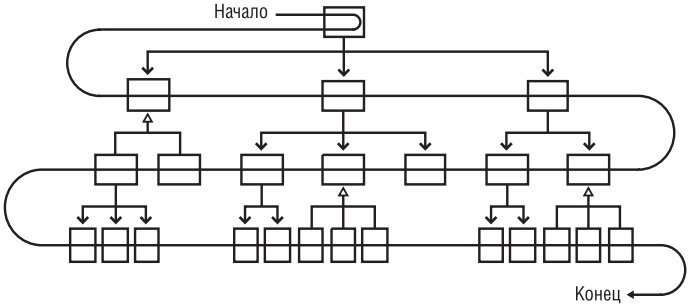



***Рис. 29-5.*** *При нисходящей интеграции вы создаете те классы, которые находятся на вершине иерархии, первыми, а те, что внизу, — последними*

При нисходящей интеграции интерфейсы между классами нужно задать очень тщательно. Самые трудные для отладки не те ошибки, что влияют на отдельные классы, а те, что проявляются из-за незаметного взаимодействия между классами. Аккуратное определение интерфейсов может уменьшить проблему. Спецификация интерфейсов не относится к интеграции, однако проверка того, что интерфейсы были спроектированы правильно, является ее задачей.

В дополнение к преимуществам, получаемым от любого типа инкрементной интеграции, нисходящая интеграция позволяет относительно просто протестировать управляющую логику программы. Все классы на вершине иерархии выполняются большое количество раз, поэтому концептуальные проблемы и ошибки проектирования проявляются достаточно быстро.

Другое преимущество нисходящей интеграции в том, что при тщательном планировании вы получите работающую систему на ранних стадиях проекта. Если часть, реализующая пользовательский интерфейс, находится на вершине иерархии, вы можете быстро получить работающую базовую версию интерфейса, а деталями заняться потом. Моральное состояние пользователей и программистов выиграет от раннего получения работающей версии.

Нисходящая инкрементная интеграция также позволяет начать кодирование до того, как все детали низкоуровневого проектирования завершены. Когда все разделы проекта проработаны достаточно подробно, можно начинать реализовывать и интегрировать все классы, стоящие на более высоких уровнях иерархии, не ожидая, когда будут расставлены все точки над «i».

Несмотря на все эти преимущества, чистый вариант нисходящей интеграции часто содержит недостатки, с которыми вы вряд ли захотите мириться. Нисходящая интеграция в чистом виде оставляет на потом работу со сложными системными интерфейсами. Если они содержат много дефектов или имеют проблемы с производительностью, вы, вероятно, хотели бы получить их задолго до конца проекта. Не так редко встречается ситуация, когда низкоуровневая проблема всплывает на самый верх системы, что приводит к высокоуровневым изменениям и снижает выгоду от раннего начала работы по интеграции. Эту проблему можно минимизировать посредством тщательного и раннего тестирования во время разработки и анализа производительности классов, из которых состоят системные интерфейсы.

Другая проблема чистой нисходящей интеграции в том, что при этом нужен целый самосвал заглушек. Масса классов более низкого уровня еще не разработана, а значит, на промежуточных этапах интеграции потребуются заглушки. Их проблема в том, что, как тестовый код, они с большей вероятностью содержат ошибки, чем тщательно спроектированный промышленный код. Ошибки в новых заглушках, используемых в новом классе, сводят на нет цель инкрементной интеграции, состоящую в том, чтобы ограничить источник ошибок одним новым классом.

**Перекрестная ссылка** Нисходящая интеграция не имеет ничего общего, кроме названия, с нисходящим проектированием (о нем см. подраздел «Нисходящий и восходящий подходы к проектированию» раздела 5.4).

Кроме того, нисходящую интеграцию практически невозможно реализовать в чистом виде. Если подходить к ней буквально, то надо начинать с вершины (назовем ее Уровнем 1), а затем интегрировать все классы следующего уровня (Уровня 2). Когда вы полностью интегрируете классы Уровня 2, и не раньше, вы начинаете интегрировать классы Уровня 3. Жесткость чистой нисходящей интеграции абсолютно деспотична. Сложно представить кого-нибудь, кто стал бы использовать нисходящую интеграцию в чистом виде. Большинство применяет гибридный подход, такой как интеграция сверху вниз с разбиением на разделы.

И, наконец, вы не можете использовать нисходящую интеграцию, если набор классов не имеет вершины. Во многих интерактивных системах понятие «вершины» субъективно. В одних системах вершиной является пользовательский интерфейс, в других — функция *main()*.

Хорошей альтернативой нисходящей интеграции в чистом виде может стать подход с вертикальным секционированием (рис. 29-6). При этом систему реализуют сверху вниз по частям, возможно, по очереди выделяя функциональные области и переходя от одной к другой.

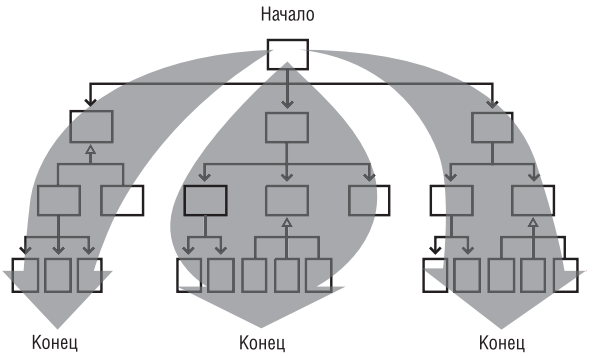



***Рис. 29-6.*** *В качестве альтернативы строгого продвижения сверху вниз можно выполнять интеграцию сверху вниз, разбив систему на вертикальные слои*

Хотя чистая нисходящая интеграция и неработоспособна, представление о ней поможет вам выбрать основной подход. Некоторые преимущества и недостатки, свойственные нисходящему подходу в чистом виде, точно так же, но менее очевидно относятся и к более свободным нисходящим методикам, таким как интеграция с вертикальным секционированием, так что их следует иметь в виду.

## Восходящая интеграция

При восходящей интеграции вы пишете и интегрируете сначала классы, находящиеся в низу иерархии. Добавление низкоуровневых классов по одному, а не всех одновременно — вот что делает восходящую интеграцию инкрементной стратегией. Сначала вы пишете тестовые драйверы для выполнения низкоуровневых классов, а затем добавляете эти классы к тестовым драйверам, пристраивая их по мере готовности. Добавляя класс более высокого уровня, вы заменяете классы драйверов реальными. На рис. 29-7 показан порядок, в котором происходит интеграция классов при восходящем подходе.

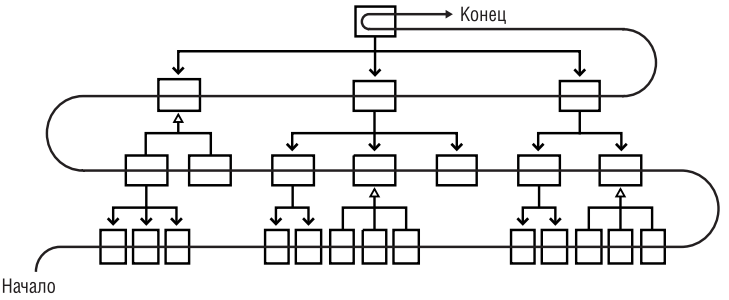



***Рис. 29-7.*** *При восходящей интеграции классы, находящиеся в низу иерархии, объединяются в начале, а находящиеся на вершине иерархии — в конце*

Восходящая интеграция дает некоторые преимущества, свойственные инкрементной стратегии. Она ограничивает возможные источники ошибок единственным классом, добавляемым в данный момент, поэтому ошибки легко обнаружить. Интеграция начинается на ранних стадиях проекта. Кроме того, при этом довольно рано реализуются потенциально ненадежные системные интерфейсы. Поскольку системные ограничения часто определяют саму возможность достижения целей, поставленных перед проектом, то применение данного подхода может быть оправданным для того, чтобы убедиться, что система удовлетворяет всему набору требований.

Главная проблема восходящей интеграции в том, что она оставляет напоследок объединение основных, высокоуровневых интерфейсов системы. Если на верхних уровнях существуют концептуальные проблемы проектирования, конструирование не выявит их, пока все детали не будут реализованы. Если проект понадобится значительно изменить, часть низкоуровневой разработки может оказаться ненужной.

Восходящая интеграция требует, чтобы проектирование всей системы было завершено до начала интеграции. Иначе в код низкого уровня могут глубоко внедриться неверные предпосылки проектирования, что поставит вас в затруднительное положение, в котором вам придется проектировать высокоуровневые классы так, чтобы обойти проблемы в низкоуровневых классах. А позволив низкоуровневым деталям реализации управлять дизайном высокоуровневых классов, вы нарушите принципы сокрытия информации и объектно-ориентированного проектирования. Проблемы интеграции классов более высокого уровня покажутся каплей в море по сравнению с проблемами, которые вы получите, если не за-

вершите проектирование высокоуровневых классов до начала низкоуровневого кодирования.

Как и нисходящую, восходящую интеграцию в чистом виде используют редко — вместо нее можно применять гибридный подход, реализующий секционную интеграцию (рис. 29-8).

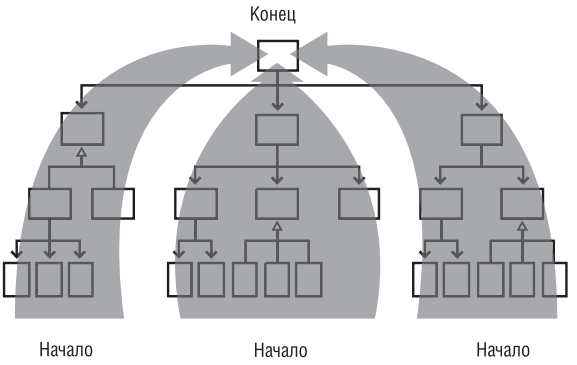



***Рис. 29-8.*** *В качестве альтернативы продвижения исключительно снизу вверх вы можете выполнять восходящую интеграцию посекционно. Это размывает различие между восходящей и функционально-ориентированной интеграцией, о которой речь пойдет ниже*

## Сэндвич-интеграция

Проблемы с нисходящей и восходящей интеграциями в чистом виде привели к тому, что некоторые эксперты стали рекомендовать сэндвич-подход (Myers, 1976). Сначала вы объединяете высокоуровневые классы бизнес-объектов на вершине иерархии. Затем добавляете классы, взаимодействующие с аппаратной частью, и широко используемые вспомогательные классы в низу иерархии. Эти высоко- и низкоуровневые классы — хлеб для сэндвича.

Напоследок вы оставляете классы среднего уровня — мясо, сыр и помидоры для сэндвича. Если вы вегетарианец, они могут представлять собой тофу и пророщенные бобы, хотя автор сэндвич-интеграции ничего не сообщает на этот счет — возможно, он не мог говорить с набитым ртом (рис. 29-9).

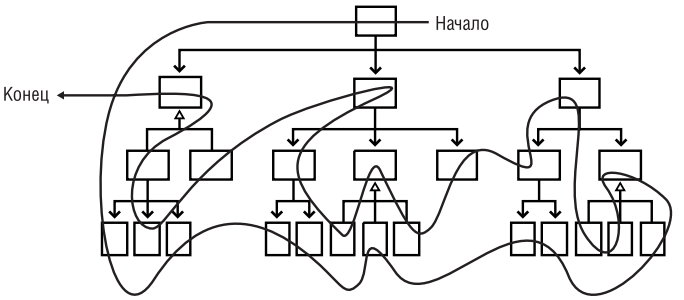



***Рис. 29-9.*** *При сэндвич-интеграции вы сначала объединяете верхний уровень и широко используемые классы нижнего уровня, оставив средний уровень напоследок*

Этот подход позволяет избежать жесткости чистых нисходящих и восходящих вариантов интеграции. Сначала интегрируются классы, вызывающие наибольшее беспокойство, при этом потенциально снижается количество лесов, которые могут вам понадобиться. Это реалистичный и практичный подход. Следующий вариант аналогичен данному, но в нем делается акцент на другом.

## Риск-ориентированная интеграция

Риск-ориентированную интеграцию, которую также называют «интеграцией, начиная с самых сложных частей» (hard part first integration), похожа на сэндвич-интеграцию тем, что пытается избежать проблем, присущих нисходящей или восходящей интеграциям в чистом виде. Кроме того, в ней также есть тенденция к объединению классов верхнего и нижнего уровней в первую очередь, оставляя классы среднего уровня напоследок. Однако суть в другом.

При риск-ориентированной интеграции (рис. 29-10) вы определяете степень риска, связанную с каждым классом. Вы решаете, какие части системы будут самыми трудными, и реализуете их первыми. Опыт показывает, что это относится к интерфейсам верхнего уровня, поэтому они часто присутствуют в начале списка рисков. Системные интерфейсы, обычно расположенные внизу, тоже представляют опасность, поэтому они также находятся в числе первых в этом списке рисков. Кроме того, вам может быть известно, что какие-то классы в середине иерархии могут также создавать трудности. Возможно, это класс, реализующий сложный для понимания алгоритм или к которому предъявляются повышенные требования по производительности. Такие классы тоже могут быть обозначены как имеющие повышенный риск, и их интеграция должна происходить на ранних стадиях.

Оставшаяся несложная часть кода может подождать. Какие-то из этих классов позже могут оказаться сложнее, чем вы предполагали, но это неизбежно.

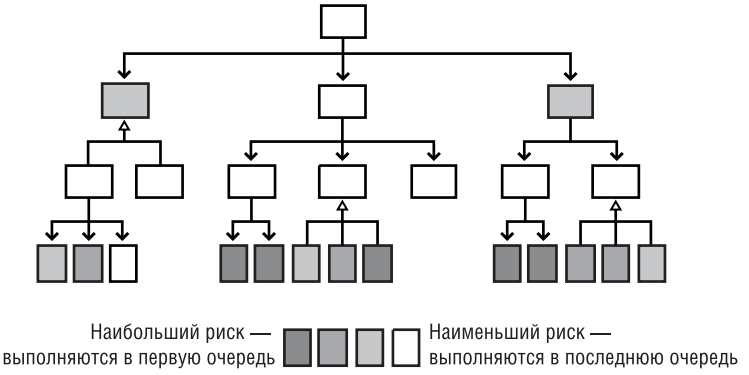



***Рис. 29-10.*** *При риск-ориентированной интеграции вы начинаете работу с тех классов, которые, по вашему мнению, могут доставить наибольшие трудности, более простые классы вы реализуете позже*

## Функционально-ориентированная интеграция

Еще один поход — интеграция одной функции в каждый момент времени. Под «функцией» понимается не нечто расплывчатое, а какое-нибудь поддающееся определению свойство системы, в которой выполняется интеграция. Если вы пишете текстовый процессор, то функцией может считаться отображение подчеркиваний, или автоматическое форматирование документа, или еще что-либо подобное.

Когда интегрируемая функция превышает по размерам отдельный класс, то «единица приращения» инкрементной интеграции становится больше отдельного класса. Это немного снижает преимущество инкрементного подхода в том плане, что уменьшает вашу уверенность об источнике новых ошибок. Однако если вы тщательно тестировали классы, реализующие эту функцию, перед интеграцией, то это лишь небольшой недостаток. Вы можете использовать стратегии инкрементной интеграции рекурсивно, сформировав сначала из небольших кусков отдельные свойства, а затем инкрементно объединив их в систему.

Обычно процесс начинается с формирования скелета, поскольку он способен поддерживать остальную функциональность. В интерактивной системе такой изначальной опцией может стать система интерактивного меню. Вы можете прикреплять остальную функциональность к той опции, которую интегрировали первой (рис. 29-11).

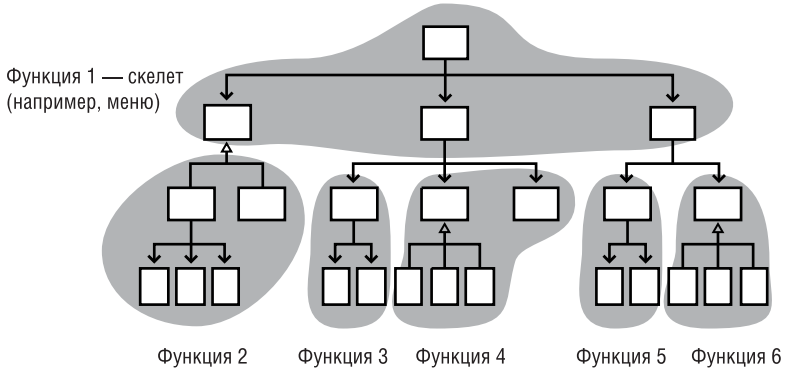



***Рис. 29-11.*** *При функционально-ориентированной интеграции вы работаете с группами классов, представляющими собой отдельные функции, поддающиеся определению, которые часто, но не всегда состоят из нескольких классов*

Компоненты добавляются в «дерево функциональности» — иерархический набор классов, реализующих отдельную функцию. Интеграцию выполнять легче, если функции относительно независимы. Например, классы, относящиеся к разной функциональности, могут вызывать один и тот же код низкоуровневых библиотек, но не использовать общий код среднего уровня. (Общие низкоуровневые классы не показаны на рис. 29-11.)

Функционально-ориентированная интеграция имеет три основных преимущества. Во-первых, она позволяет обойтись без лесов практически везде, кроме низкоуровневых библиотечных классов, Какие-то заглушки могут понадобиться скелету, или же некоторые его части просто могут быть неработоспособными, пока не будут

добавлена конкретная функциональность. Однако когда каждая функция будет добавлена к скелету, дополнительных лесов не потребуется. Поскольку вся функции изолированы друг от друга, в них содержится весь необходимый ей код.

Второе главное преимущество состоит в том, что каждая новая функция расширяет функциональность. Это наглядно показывает, что проект постоянно продвигается вперед. Кроме того, создается работоспособное ПО, которое можно представить заказчикам для оценки или сдать в эксплуатацию раньше намеченного срока, реализовав меньшую функциональность, чем планировалось изначально.

Третье преимущество в том, что функционально-ориентированная интеграция хорошо сочетается с объектно-ориентированным проектированием. Как правило, можно легко провести соответствие между объектами и функциями, что делает функционально-ориентированную интеграцию естественным выбором для объектно-ориентированных систем.

Придерживаться чистой функционально-ориентированной интеграции так же сложно, как и нисходящей и восходящей интеграций в чистом виде. Обычно часть низкоуровневого кода должна быть реализована прежде, чем можно будет добавлять какую-либо функциональность.

## T-образная интеграция

Последний подход, который часто упоминается в связи с проблемами нисходящей и восходящей методик, называется «Т-образной интеграцией». При таком подходе выбирается некоторый вертикальный слой, который разрабатывается и интегрируется раньше других. Этот слой должен проходить сквозь всю систему от начала до конца и позволять выявлять основные проблемы в допущениях, сделанных при проектировании системы. Реализовав этот вертикальный участок (и устранив все связанные с этим проблемы), можно разрабатывать основную канву системы (например, системное меню для настольного приложения). Этот подход часто комбинируют с риск-ориентированной и функционально-ориентированной интеграциями (рис. 29-12).

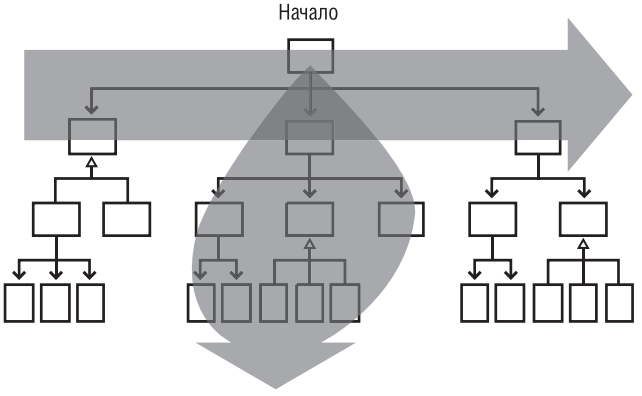



***Рис. 29-12.*** *При Т-образной интеграции вы создаете и интегрируете вертикальный срез системы, чтобы проверить архитектурные допущения. После этого вы создаете и интегрируете основную горизонталь системы, чтобы предоставить каркас для разработки остальной функциональности*

### Краткий итог методик интеграции

Восходящая, нисходящая, сэндвич, риск-ориентированная, функционально-ориентированная, Т-образная — не кажется ли вам, что эти названия придумываются на ходу? Так и есть. Ни один из этих подходов не является жесткой процедурой, которой вы должны методично следовать, начиная с шага 1 и заканчивая шагом 47, а затем объявить задание выполненным. Как и другие подходы к проектированию ПО, они скорее эвристические, чем алгоритмические. Поэтому вместо того, чтобы безусловно следовать какой-то процедуре, вам придется разработать свою уникальную стратегию, подходящую именно вашему проекту.

## 29.4. Ежедневная сборка и дымовые тесты

**Дополнительные сведения** Большая часть этого материала позаимствована из главы 18 книги «Rapid Development» (McConnell, 1996). Если вы ее читали, можете переходить к разделу «Непрерывная интеграция».

Какую бы стратегию интеграции вы и выбрали, хорошим подходом к разработке ПО является «ежедневная сборка и дымовые тесты» (daily build and smoke test). Ежедневно каждый файл компилируется, компонуется и собирается в выполняемую программу. После чего прогоняется «дымовой тест» — относительно простая проверка, определяющая, «дымится» ли продукт во время выполнения[1].

Этот простой процесс дает несколько преимуществ. Он уменьшает риск низкого качества, который связан с риском неуспешной или неправильной интеграции. Проверяя «на дым» весь код ежедневно, вы не позволяете проблемам с качеством получить контроль над проектом. Вы приводите систему в понятное, правильное состояние и сохраняете ее в таком виде. Вы просто не позволяете ей ухудшаться до такой степени, когда могут возникнуть проблемы с качеством, отнимающие много времени.

Этот процесс также поддерживает упрощенную диагностику ошибок. Когда продукт собирается и тестируется каждый день, легко засечь, почему его работа в определенный день была нарушена. Если проект работал на 17-й день, а на 18-й перестал, то нечто, происшедшее между этими двумя сборками, и нарушило работу продукта.

Он улучшает моральный климат. Осознание того, что продукт работает, дает потрясающий положительный заряд. Практически не имеет значения, что он делает. Разработчики будут рады, даже если продукт будет рисовать прямоугольник на экране! При ежедневных сборках с каждым днем все большая часть продукта начинает работать, и это поддерживает дух на высоте.

У частой интеграции есть побочный эффект: она выявляет такие проблемы, какие в противном случае могли бы незаметно накапливаться и неожиданно проявиться в конце проекта. Такое накопление скрытых результатов может стать занозой в конце проекта и потребовать для устранения недели или месяцы. Команды, не применяющие ежедневных сборок, иногда считают, что они могут снизить эффективность их работы до скорости улитки. На самом же деле ежедневные сборки

[1] Термин «дымовой тест» появился в электронике. Разработчики на определенном этапе включают испытываемое устройство в сеть и смотрят, не задымится ли оно. — *Прим. перев.*

распределяют работу в проекте более равномерно, и команда просто получает более аккуратную картину того, как быстро продвигается проект.

***Создавайте сборку ежедневно*** Основной смысл ежедневной сборки заключен в слове «ежедневная». Как говорит Джим Маккарти, рассматривайте ежедневную сборку как пульс проекта (McCarthy, 1995). Пульса нет — проект мертв. Менее метафорично Майкл Кьюсумано и Ричард Селби определяют ежедневную сборку как синхроимпульс проекта (Cusumano and Selby, 1995). Части кода могут немного рассинхронизироваться между этими импульсами, но каждый раз во время импульса код должен приводиться в соответствие. Когда вы настаиваете на достаточно частых импульсах, вы предотвращаете полное рассогласование частей проекта.

Некоторые организации выполняют сборки еженедельно, а не ежедневно. Проблема в том, что, если сборка на какой-то неделе была испорчена, вам может понадобиться несколько недель, прежде чем вы получите следующую хорошую сборку. Когда такое случается, вы теряете практически все преимущества частых сборок.

***Проверяйте правильность сборок*** Чтобы процесс ежедневных сборок действовал, собираемое ПО должно быть работоспособным. В противном случае сборка считается испорченной, и ее исправление становится приоритетной задачей.

Каждый проект задает свои стандарты того, что считается «сломанной сборкой». Устанавливаемый стандарт должен быть достаточно строгим, чтобы не допускать серьезных дефектов, но и достаточно снисходительным, чтобы пренебрегать простейшими ошибками, способными парализовать движение вперед, если им уделять чрезмерное внимание.

«Хорошая» сборка должна как минимум:

- успешно компилировать все файлы, библиотеки и другие компоненты;
- успешно компоновать все файлы, библиотеки и другие компоненты;
- не содержать серьезных дефектов, препятствующих запуску программы или делающих ее выполнение непредсказуемым, — иначе говоря, хорошая сборка должна проходить дымовой тест.

***Выполняйте дымовые тесты ежедневно*** Дымовой тест испытывает всю систему от начала до конца. Он не должен быть всесторонним, но должен быть способен выявить основные проблемы. Этот тест должен быть достаточно тщательным, чтобы прошедшая его сборка могла считаться стабильной для выполнения более основательного тестирования.

Ежедневная сборка не имеет большого значения без дымового теста. Этот тест как часовой, который охраняет от появления проблем, снижающих качество продукта и препятствующих интеграции. Без него ежедневные сборки становятся просто длительными упражнениями, позволяющими лишь убедиться в наличии еже-дневной безошибочной компиляции.

***Поддерживайте актуальность дымового теста*** Дымовой тест должен развиваться по мере развития системы. Поначалу он может проверять что-то простое, скажем, может ли система говорить «Hello, World». По мере разработки системы дымовой тест становится более изощренным. Если выполнение первой проверки может быть делом нескольких секунд, то с ростом системы дымовой тест может удлиниться до 10 минут, часа или больше. Если дымовой тест не под-

держивается в актуальном состоянии, то ежедневная сборка может стать самообманом: разрозненный набор тестовых данных создает ложную уверенность в качестве продукта.

***Автоматизируйте ежедневную сборку и дымовой тест*** Поддержка сборок может отнимать много времени. Автоматизация процессов ежедневной сборки и тестирования позволяет убедиться, что код собирается и проверка выполняется. Без автоматизации выполнять сборку и дымовой тест непрактично.

***Организуйте группу, отвечающую за сборки*** В большинстве проектов надзор за ежедневными сборками и поддержание актуальности дымовых тестов становится достаточно большой задачей, чтобы стать заметной частью чьей-то работы. В больших проектах эти задачи могут обеспечить полную занятость для нескольких человек. Например, при выпуске первой версии Microsoft Windows NT группа, ответственная за сборки, состояла из четырех человек, работающих полный рабочий день (Zachary, 1994).

***Вносите исправления в сборку, только когда имеет смысл это делать...*** Отдельные разработчики обычно пишут код не настолько быстро, чтобы делать значимый вклад в систему ежедневно. Им следует работать над фрагментом кода и интегрировать его, когда у них будет некая совокупность кода в целостном состоянии — обычно раз в несколько дней.

***...но не откладывайте внесение исправлений надолго*** Старайтесь регистрировать код не слишком редко. Существует вероятность, что разработчик настолько запутается в списке изменений, что потребуется исправить практически каждый файл в системе. Это снижает ценность ежедневной сборки. Остальная часть команды будет продолжать использовать преимущества инкрементной интеграции, а этот отдельный разработчик — нет. Если программист откладывает регистрацию изменений более, чем на пару дней, можно считать, что его работа подвергается риску. Как указывает Кент Бек, частая интеграция иногда заставляет разбивать конструирование отдельной функции на несколько частей. Такие накладные расходы — приемлемая цена за снижение рисков интеграции, улучшение видимости состояния, улучшение контролепригодности и прочие преимущества частой интеграции (Beck, 2000).

***Требуйте, чтобы разработчики проводили дымовое тестирование своего кода перед его добавлением в систему*** Программисты должны тестировать собственный код, прежде чем добавить его к сборке. Разработчик может это сделать, создав частную сборку системы на собственной машине и протестировав ее самостоятельно. Или он может передать частную сборку «партнеру по тестированию» — тестировщику, который сосредоточится на коде этого разработчика. В обоих случаях цель в том, чтобы убедиться, что новый код пройдет «проверку на дым», прежде чем ему будет позволено влиять на остальные части системы.

***Создайте область промежуточного хранения кода, который следует добавить к сборке*** Успешность ежедневной сборки частично зависит от знания, какая сборка хорошая, а какая — нет. При тестировании собственного кода у разработчиков должна быть возможность использовать систему, о которой точно известно, что она исправна.

Большинство групп решает эту проблему с помощью создания области промежуточного хранения кода, который, по мнению разработчиков, готов к добавлению к сборке. Новый код помещается в эту область, создается новая сборка, и, если ее состояние признано удовлетворительным, новый код перемещается в основное хранилище.

В небольших и средних проектах система управления версиями может обеспечить такую функцию. Разработчики регистрируют свой код в этой системе. Тот, кто хочет использовать заведомо хорошую сборку, просто устанавливает флаг даты в свойствах файла системы управления версиями. Значение этого флага сообщит системе, что следует извлекать файлы, основываясь на дате последней правильной сборки.

В больших проектах или там, где используется слишком простое ПО управления версиями, создавать область промежуточного хранения приходится выполнять вручную. Автор нового кода посылает письмо в группу, ответственную за сборку, сообщая, где можно найти новые файлы, предназначенные для регистрации в системе. Или группа заводит «область данных для регистрации» на файловом сервере, где разработчики размещают новые версии своих исходных файлов. Ответственная за сборку группа принимает на себя обязательства по регистрации нового кода в системе управления версиями после того, как удостоверится, что этот код не нарушит сборку.

***Назначьте наказание за нарушение сборки*** Большинство групп, применяющих ежедневную сборку, назначает наказание за ее нарушение. Сделайте ясным с самого начала, что поддержание работоспособности сборок — одна из приоритетных задач проекта. Испорченная сборка должна быть исключением, а не правилом. Настаивайте на том, чтобы разработчики, испортившие сборку, прекращали всю работу до тех пор, пока ее не восстановят. Если сборка ломается слишком часто, трудно всерьез относиться к вопросу поддержания ее работоспособности.

Забавное наказание поможет сделать акцент на этом приоритете. В некоторых группах на дверь офиса «сосунка», испортившего сборку, вешают леденцы, которые должны висеть до тех пор, пока проблема не будет исправлена. В других группах провинившиеся должны надевать козлиные рога или вносить по пять долларов в специальный фонд.

Другие проекты устанавливают более существенное наказание. Разработчики наиболее заметных проектов в Microsoft, например, Windows 2000 и Microsoft Office, на последних стадиях проектов должны носить с собой пейджеры. Если они разрушают сборку, их вызывают для ее починки, даже если дефект обнаружен в три часа утра.

***Выпускайте сборки по утрам*** В некоторых группах пришли к выводу, что предпочтительней создавать сборку ночью, проводить дымовой тест рано утром и выпускать новые сборки утром, а не ближе к вечеру. Утреннее тестирование и выпуск сборок имеют несколько преимуществ.

Во-первых, если вы выпускаете сборку утром, тестировщики могут работать в этот день со свежей сборкой. Если вы выпускаете сборку после обеда, тестировщики чувствуют себя обязанными запустить автоматические тесты перед уходом с работы. Если сборка задерживается, что случается довольно часто, им приходится

оставаться после работы, чтобы запустить свои тесты. Поскольку в таких задержках нет их вины, процесс сборки становится деморализующим.

Заканчивая сборку утром, вы получаете более надежный доступ к разработчикам, со сборкой возникли проблемы. Во время рабочего дня программисты находятся в офисе. Вечером они могут быть где угодно. Даже если разработчикам выданы пейджеры, не всегда легко их найти.

Возможно, было бы круче начинать дымовой тест в конце дня и при возникновении проблем вызывать людей на работу посреди ночи, но это усложняет работу команды, приводит к бесполезной трате времени — в результате же вы больше потеряете, чем приобретете.

***Создавайте сборку и проводите дымовой тест даже в экстремальных условиях*** Когда сроки начинают поджимать, ежедневные сборки могут показаться излишней роскошью. На самом деле все наоборот. В стрессовых условиях дисциплина программистов ухудшается. Под давлением обстоятельств они прибегают к таким методам ускорения конструирования, которые не использовали бы в менее критичных ситуациях. Они рецензируют и тестируют собственный код менее тщательно, чем обычно. Код стремится к состоянию энтропии быстрее, чем это происходит при меньшем стрессе.

На этом фоне ежедневные сборки ужесточают дисциплину и поддерживают на плаву критические проекты.

## Для каких проектов применять ежедневную сборку?

Некоторые разработчики считают, что проводить ежедневные сборки непрактично, так как их проекты слишком велики. Но, вероятно, в одном из самых сложных недавних программных проектов успешно использовалась практика ежедневных сборок. К моменту выпуска Microsoft Windows 2000 состояла из примерно 50 миллионов строк кода, распределенных по десяткам тысяч исходных файлов. Полная сборка занимала 19 часов на нескольких машинах, но разработчики Windows 2000 оказались в состоянии проводить сборки каждый день. Они не стали помехой — напротив, команда Windows 2000 видит одну из причин успеха в еже-дневных сборках. Чем больше проект, тем большую важность имеет инкрементная интеграция.

При рассмотрении 104 проектов в США, Индии, Японии и Европе выяснилось, что только 20–25% из них используют ежедневные сборки в начале или середине процесса разработки (Cusumano et al., 2003). Таким образом, в них имеются широкие возможности для улучшения.

## Непрерывная интеграция

Некоторые авторы расценивают ежедневные сборки как аргумент в пользу выполнения *непрерывной* интеграции (Beck, 2000). В большинстве публикаций под «непрерывной» понимается «по крайней мере ежедневная» интеграция (Beck, 2000), что мне кажется обоснованным. Но порой я встречаю людей, понимающих слово «непрерывная» буквально: они стремятся выполнять интеграцию каждого изменения с самой последней сборкой каждые пару часов. Думаю, для большинства проектов действительно непрерывная интеграция выходит за рамки разумного.

В свободное время я руководжу работой дискуссионной группы, главных технических менеджеров таких компаний, как Amazon.com, Boeing, Expedia, Microsoft, Nordstrom и др. Опрос показал, что *никто* из руководителей не считает непрерывную интеграцию предпочтительней ежедневной. В средних и больших проектах иногда имеет смысл допустить рассинхронизацию кода на короткое время. Разработчики часто нарушают синхронизацию, когда делают масштабные изменения. Через какое-то время они могут синхронизировать проект снова. Ежедневные сборки позволяют проектной команде организовать точки согласования достаточно часто. Если изменения синхронизируются ежедневно, нет нужды делать это непрерывно.

**Контрольный список: интеграция**

http://cc2e.com/2992

**Стратегия интеграции**

- ❑ Определяет ли стратегия оптимальный порядок, в котором должны интегрироваться подсистемы, классы и методы?
- ❑ Скоординирован ли порядок интеграции с порядком конструирования, чтобы классы были подготовлены к интеграции вовремя?
- ❑ Приводит ли стратегия к упрощению диагностики дефектов?
- ❑ Позволяет ли стратегия сократить использование «подпорок» до минимума?
- ❑ Имеет ли выбранная стратегия преимущества перед другими подходами?
- ❑ Хорошо ли определены интерфейсы между компонентами? (Определение интерфейсов не является задачей интеграции, а проверка правильности их определения — является.)

**Ежедневная сборка и дымовые тесты**

- ❑ Выполняется ли сборка проекта достаточно часто (в идеале каждый день), чтобы поддерживать инкрементную интеграцию?
- ❑ Выполняется ли дымовой тест для каждой сборки так, что вам становится известно, находится ли сборка в рабочем состоянии?
- ❑ Автоматизированы ли сборка и дымовой тест?
- ❑ Часто ли разработчики регистрируют свой код: не превышает ли время между исправлениями день или два?
- ❑ Поддерживается ли дымовой тест в соответствии с кодом, расширяясь при его расширении?
- ❑ Редко ли происходит нарушение работоспособности сборки?
- ❑ Выполняете ли вы сборку и дымовой тест даже в экстремальных обстоятельствах?

# Дополнительные ресурсы

Вопросам, обсуждаемым в этой главе, посвящены следующие публикации.

http://cc2e.com/2999

### Интеграция

Lakos, John. *Large-Scale C++ Software Design.* Boston, MA: Addison-Wesley, 1996. Лейкос доказывает, что «физическое представление» системы — иерархия файлов, каталогов и библиотек — заметно влияет на способность команды разработчиков

скомпилировать ПО. Если вы не обращаете должного внимания на физическое представление, то время сборки проекта может стать довольно долгим, что сведет на нет преимущества частой интеграции. Лейкос пишет о C++, но выводы, относящиеся к «физическому представлению» актуальны, и для проектов на других языках.

Myers, Glenford J. *The Art of Software Testing*. New York, NY: John Wiley & Sons, 1979. В этой классической книге о тестировании интеграция рассматривается как операция тестирования.

### Инкрементный подход

McConnell, Steve. *Rapid Development*. Redmond, WA: Microsoft Press, 1996. В главе 7 (Lifecycle Planning) «Планирование жизненного цикла» подробно рассмотрены плюсы и минусы более гибких и менее гибких моделей жизненного цикла. В главах 20, 21, 35 и 36 обсуждаются конкретные модели жизненных циклов, поддерживающие инкрементный подход в разной степени. Глава 19 содержит описание «*проектирования с поддержкой изменений*» (designing for change) — ключевой методики, необходимой для поддержки интерактивной и инкрементной моделей разработки.

Boehm, Barry W. «A Spiral Model of Software Development and Enhancement». *Computer*, May 1988: 61–72. В этой статье Бом описывает свою «спиральную модель» разработки ПО. Он предлагает эту модель в качестве подхода к управлению рисками в программном проекте, поэтому статья больше касается разработки в целом, чем конкретно интеграции. Бом — один из выдающихся экспертов в области мас-штабных вопросов разработки ПО, и доходчивость его объяснений отражает глубину его понимания темы.

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley, 1988. Главы 7 и 15 содержат исчерпывающее обсуждение эволюционной поставки — одного из первых подходов к инкрементной разработке.

Beck, Kent. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley, 2000. Эта книга содержит более современное, лаконичное и евангеличе-ское представление большинства идей, приведены в книге Гилба. Лично я отдаю предпочтение глубокому анализу Гилба, но некоторые читатели могут посчитать изложение Бека более доступным или применимым непосредственно к тому типу проекта, над которым они работают.

## Ключевые моменты

- Последовательность конструирования и интеграционный подход влияют на порядок, в котором классы проектируются, кодируются и тестируются.
- Порядок интеграции снижает затраты на тестирование и упрощает отладку.
- Инкрементная интеграция имеет несколько вариантов, и, помимо совсем тривиальных проектов, любой из них лучше, чем поэтапная интеграция.

- Лучший интеграционный подход для каждого конкретного проекта — обычно сочетание нисходящего, восходящего, риск-ориентированного и других интеграционных подходов. Т-образная интеграция и интеграция с вертикальным секционированием часто дают хорошие результаты.
- Ежедневные сборки могут уменьшить проблемы с интеграцией, улучшить моральный климат среди разработчиков и предоставить полезную информацию, касающуюся управления проектом.

ГЛАВА 30

# Инструменты программирования

http://cc2e.com/3084

**Содержание**



**Связанные темы**

- Инструменты для управления версиями: раздел 28.2
- Инструменты для отладки: раздел 23.5
- Инструменты для облегчения тестирования: раздел 22.5

Современные инструменты сокращают время конструирования. Передовой набор инструментов (и знание этих средств) позволяет повысить производительность более, чем на 50% (Jones, 2000; Boehm et al., 2000). Программный инструментарий также позволяет уменьшить объем однообразной, монотонной работы.

Собака, может, и лучший друг человека, но хорошие инструменты — лучшие друзья программиста. Как уже давно выяснил Барри Бом, 20% инструментария используются приблизительно в 80% случаев (1987b). Если вы не знакомы с каким-то из наиболее полезных инструментов, вы упускаете шанс облегчить себе жизнь.

В этой главе мы поговорим об инструментах для конструирования (об инструментах, применяемых в процессе определения требований, управления, а также на протяжении всего цикла разработки, см. раздел «Дополнительные ресурсы» в конце главы). Темой нашего разговора будут не конкретные средства, а их разновидности. Некоторые инструменты настолько широко распространены, что упоминаются по имени, но номера версий, имена продуктов и компаний меняются так быстро, что информация о большинстве из них оказалась бы устаревшей до того, как высохли бы чернила на этих страницах.

# 30.1. Инструменты для проектирования

**Перекрестная ссылка** О проектировании см. главы 5–9.

Существующие инструменты для проектирования в основном представляют собой графические средства, позволяющие создавать диаграммы проекта. Иногда они встроены в ПО для автоматизированной разработки (CASE-средства) с более широкими функциями. Некоторые поставщики рекламируют инструменты для проектирования как самостоятельные CASE-средства. Проектирование с помощью графических инструментов позволяет использовать для проекта стандартные графические нотации: UML, архитектурные блок-схемы, иерархические диаграммы, диаграммы связи сущностей или диаграммы классов. Некоторые инструменты поддерживают только одну нотацию, другие — несколько.

Может показаться, что инструменты для проектирования — всего лишь необычные графические пакеты. Однако они обладают возможностями, которых в обычных графических пакетах нет: если нарисовать схему с помощью кружков и стрелочек, а затем удалить один кружок, графическое средство проектирования автоматически перестроит другие элементы, включая соединительные стрелки и кружки более низкого уровня, которые взаимодействовали с удаленным кружком. Инструментарий берет на себя служебные операции и при добавлении нового кружка. Средство проектирования позволяет перемещаться между уровнями абстракции. Инструмент проверит целостность проекта, а некоторые могут создавать код на основе разработанного проекта.

# 30.2. Инструменты для работы с исходным кодом

Инструменты, для работы с исходным кодом обладают более широкими и развитыми возможностями в сравнении со средствами, предназначенными для проектирования.

## Редактирование

Эта группа инструментов служит для редактирования исходного кода.

### Интегрированные среды разработки (IDE)

По некоторым оценкам до 40% рабочего времени программист тратит на редактирование исходного кода (Parikh, 1986; Ratliff, 1987). В таком случае покупка IDE — хорошее вложение денег.

В дополнение к основным функциям по обработке текста хорошие среды разработки предлагают следующую функциональность:

- компиляция и поиск ошибок, не выходя из редактора;
- интеграция с системой управления версиями, средствами тестирования и отладки;
- сжатое или развернутое представление программы (вывод имен классов или логических структур без показа их содержания, также называемый «свертыванием»);

- переход к определениям классов, методов и переменных;
- переход к любым местам использования класса, метода или переменной;
- форматирование в соответствии с используемым языком;
- интерактивная подсказка для языка редактируемой программы;
- поиск соответствующих скобок (операторов *begin-end*);
- шаблоны для часто применяемых языковых конструкций (например, редактор может ввести полную структуру цикла *for* после того, как программист наберет слово *for*);
- интеллектуальные отступы (включая возможность простого изменения отступов в блоке выражений при изменении логики);
- автоматизированное преобразование и рефакторинг кода;
- создание макросов на знакомом языке программирования;
- список строк для поиска, который позволяет не набирать повторно часто используемые строки;
- регулярные выражения в операциях поиска и замены;
- поиск и замена в группе файлов;
- одновременное редактирование нескольких файлов;
- параллельное сравнение файлов;
- многоуровневая операция отмены.

Кстати, есть программы, поддерживающие все перечисленные возможности.

### Поиск и замена строк в нескольких файлах

Если ваш редактор не поддерживает поиск и замену строк в нескольких файлах одновременно, вы можете тем не менее найти дополнительные средства для выполнения этого задания. Эти инструменты позволяют искать все вхождения названия класса или метода. Обнаружив ошибку в своем коде, вы можете использовать такой инструмент для поиска похожих ошибок в других файлах.

Вы можете искать точное совпадение строки, похожие строки (без учета регистра символов) или использовать регулярные выражения. Регулярные выражения предоставляют особенно мощные возможности, поскольку позволяют производить поиск, применяя сложные строковые шаблоны. Если бы вы хотели найти все случаи применения массивов с использованием магических чисел (цифр от «0» до «9»), вы могли бы выполнить поиск символа «[», за которым идет 0 или несколько пробелов, за которыми следует одно или несколько чисел, потом опять 0 или больше пробелов и символ «]». Одна широко распространенная программа поиска называется «grep». Запрос для поиска магических чисел с помощью grep мог бы выглядеть так:

```
grep "\[ *[0-9]+ *\]" *.cpp
```

Критерий поиска можно усложнить, чтобы точней настроить поисковую строку.

Полезно иметь возможность заменять строки в нескольких файлах одновременно. Например, чтобы дать методу, константе или глобальной переменной лучшее имя, может понадобиться выполнить замену в нескольких файлах. Это легко сделать с помощью утилит, позволяющих менять строки в наборе файлов, и это хо-

рошо, потому что у вас должно быть как можно меньше препятствий для создания других имен классов, методов и констант. Для обработки изменений строк в нескольких файлах служат такие инструменты, как Perl, AWK и sed.

### Инструменты для сравнения

Если вы сделали несколько попыток исправить ошибку и хотите удалить неудачные попытки, то программа сравнения сопоставит исходный и исправленный файлы и перечислит измененные строки. Если вы работаете над проектом вместе с другими людьми и хотите увидеть, какие исправления они внесли с тех пор, как вы работали с кодом в последний раз, программа сравнения (например, Diff) сравнит текущую версию с последней версией кода, над которой вы работали, и покажет различия. Такая функциональность часто встроена в средства для контроля исправлений.

### Инструменты для слияния

Один из видов контроля исправлений заключается в блокировании исходных файлов, так что только один человек может их модифицировать. Можно также позволить нескольким сотрудникам работать с файлами одновременно, а при регистрации производить слияние изменений. Предназначенные для этого инструменты обычно выполняют слияние простых изменений автоматически, а в сложных случаях и при возникновении конфликтов обращаются к пользователю.

### Программы для украшения исходного кода

Программы украшения исходного кода выделяют имена классов и методов, стандартизуют стиль отступов, единообразно форматируют комментарии и т. д. Некоторые из них могут помещать каждый метод на отдельную Web-страницу или на печатную страницу и форматировать код. Большинство инструментов позволяет настраивать способы украшения кода.

**Перекрестная ссылка** О форматировании кода программы см. главу 31.

Программы для украшения кода бывают двух видов. Одни используют исходный код как входные данные и не изменяют исходного кода. Другие изменяют сам исходный код, стандартизуя отступы, форматируя списки параметров и т. д. Это полезно при работе с большими объемами унаследованного кода.

### Средства документирования интерфейса

Некоторые инструменты извлекают подробную документацию по программному интерфейсу из файлов с исходным кодом. Код в исходных файлах содержит некоторые подсказки, например, поля *@tag*, определяющие текст, который следует извлечь. Инструмент для документирования интерфейса извлекает помеченный текст и представляет его в красиво отформатированном виде. Прекрасным примером такого инструмента может служить Javadoc.

### Шаблоны

Шаблоны упрощают задачи ввода данных с клавиатуры. Допустим, вы хотите добавлять стандартный пролог комментариев перед началом ваших методов. Вы можете создать скелет такого пролога с использованием правильного синтаксиса-

са и поместить в него нужные элементы. Этот скелет станет «шаблоном», который вы сохраните в файле или назначите клавиатурному макросу. При создании нового метода вы легко вставите шаблон в файл с исходным кодом. Шаблоны позволяют настроить как крупные сущности, скажем, классы и файлы, так и небольшие структуры вроде циклов.

При работе над групповым проектом шаблоны могут стать простым способом достижения согласованного стиля кодирования и документирования.

### Средства создания перекрестных ссылок

Инструментарий по созданию перекрестных ссылок перечисляет (обычно на Web-страницах) переменные и методы и все места, где они применяются.

### Генераторы иерархии классов

Генератор иерархии классов предоставляет сведения о деревьях наследования. Это бывает полезно при отладке, но чаще всего применяется для анализа структуры программы или для разбиения программы на модули, пакеты или подсистемы. Такая функциональность иногда реализована и в IDE.

### Анализаторы качества кода

Инструментарий этой категории исследует статический исходный код с целью оценки его качества.

### Программы углубленного контроля синтаксиса и семантики

Средства контроля синтаксиса и семантики осуществляют более тщательную проверку кода, чем это обычно делает компилятор. Ваш компилятор может проверять наличие только элементарных синтаксических ошибок. При углубленном контроле могут учитываться нюансы языка, что позволяет проверить наличие более коварных ошибок — тех, что не выглядят таковыми с точки зрения компилятора. Например, в C++ выражение:

```
while ( i = 0 ) ...
```

абсолютно законно, но обычно имеется в виду:

```
while ( i == 0 ) ...
```

Первая строка синтаксически корректна, но путаница со знаками = и == является распространенной ошибкой, и данная строка, возможно, неправильна. Lint — углубленный анализатор синтаксиса и семантики, используемый во многих средах C/C++, — предупреждает о наличии неинициализированных переменных, переменных, которым присвоено значение, но которые никогда не используются, выходных параметрах метода, которым не было присвоено значение внутри метода, подозрительных операциях с указателями, подозрительных логических сравнениях (вроде приведенного выше), недостижимом коде и прочих распространенных проблемах. Другие языки предлагают похожие инструменты.

### Генераторы отчетов о метриках

Некоторые инструменты составляют отчет о качестве кода. Например, средства, сообщающие о сложности каждого метода, позволяют направить наиболее сложные функции на дополнительное рецензирование, тестирование или перепроектирование. Некоторые средства подсчитывают количество строк кода, объявлений данных, комментариев и пустых строк как для всей программы, так и для отдельных методов. Они отслеживают дефекты, внесенные конкретными программистами, Затем фиксируют изменения, сделанные для исправления дефектов, и программистов, внесших эти правки. Они подсчитывают количество модификаций ПО и выделяют процедуры, которые исправляются чаще всего. Установлено, что средства анализа сложности положительно влияют на производительность сопровождения, увеличивая ее примерно на 20% (Jones, 2000).

**Перекрестная ссылка** О метриках см. раздел 28.4.

## Рефакторинг исходного кода

Несколько инструментов оказывают помощь при конвертации исходного кода из одного формата в другой.

### Инструменты для рефакторинга

Программа рефакторинга поддерживает процесс рефакторинга кода как в автономном режиме, так и с интеграцией в IDE. Браузеры рефакторинга позволяют легко изменить имя класса по всему исходному коду. Они дают возможность создавать новый метод, просто выделив код, который в него нужно перенести, указав имя для этого нового метода и перечислив список параметров. Средства рефакторинга делают процесс изменения кода более быстрым и менее подверженным ошибкам. Они уже доступны для языков Java и Smalltalk и получают все большее распространение для других языков. Об инструментах рефакторинга см. также главу 14 «Refactoring Tools» в книге «Refactoring» (Fowler, 1999).

**Перекрестная ссылка** О рефакторинге см. главу 24.

### Инструменты для реструктуризации

Программа реструктуризации преобразует тарелку спагетти-кода с операторами *goto* в более питательное блюдо из структурированного кода без *goto*. Кейперс Джонс сообщает, что в процессе сопровождения средства реструктуризации могут повысить производительность на 25–30% (Jones, 2000). Программе реструктуризации при конвертации кода приходится делать массу допущений, и, если логика оригинала была ужасной, она останется такой и в сконвертированной версии. Однако если вы выполняете преобразование вручную, вы можете использовать этот инструмент для простых вариантов, а сложные участки обработать вручную. В качестве альтернативы можно прогнать код через программу реструктуризации и использовать результат как отправную точку для ручного преобразования.

### Трансляторы кода

Некоторые инструменты переводят код с одного языка программирования на другой. Транслятор позволяет перенести большой объем кода в другую среду. Учтите

однако, что, если вы изначально имеете плохой код, транслятор просто переведет этот плохой код на другой язык.

## Управление версиями

**Перекрестная ссылка** О таких инструментах см. подраздел «Изменения в коде программного обеспечения» раздела 28.2.

Справиться с быстро растущим количеством версий ПО позволяют инструменты управления версиями, которые предоставляют следующие возможности:

- управление исходным кодом;
- управление зависимостями наподобие того, что делает утилита make в операционных системах UNIX;
- управление версиями проектной документации;
- установка соответствия между элементами проекта, такими как требования, код и тестовые данные, чтобы в случае изменения требований вы могли определить, какой код и какие тесты будут затронуты.

## Словари данных

Так называются базы данных, которые описывают важную для проекта информацию. Во многих случаях словарь связан преимущественно со схемами баз данных. В больших проектах такой словарь служит для отслеживания сотен или тысяч определений классов. В больших групповых проектах он позволяет избежать конфликтов имен. Конфликт может быть просто синтаксическим (когда одно и то же имя используется дважды) и более запутанным, при котором различные имена служат для обозначения одного и того же понятия или одно и то же имя обозначает немного разные вещи. Для каждого элемента данных (таблицы базы данных или класса) словарь содержит имя и описание. Кроме того, в нем могут содержаться пояснения относительно применения элемента.

# 30.3. Инструменты для работы с исполняемым кодом

Средства для работы с исполняемым кодом столь же разнообразны, как и инструменты, предназначенные для исходного кода.

## Создание кода

Инструменты, описанные в этом разделе, оказывают помощь при создании программы.

### Компиляторы и компоновщики

Компиляторы преобразуют исходный код в исполняемый. Большинство программ предназначено для компиляции, хотя еще встречается и интерпретируемый код.

Типичный компоновщик связывает один или несколько объектных файлов, которые компилятор сгенерировал из ваших исходников, со стандартным кодом, необходимым для создания исполняемых программ. Компоновщики, как правило, могут объединять файлы, созданные на разных языках, позволяя вам выбирать

язык, наиболее подходящий для каждой части вашей программы, и не задумываться над деталями интеграции.

Оверлейные компоновщики помогают вместить 10 фунтов в 5-фунтовый мешок, создавая программы, выполняющиеся в меньшем объеме памяти, чем требуется для их размещения. Оверлейный компоновщик создает такой исполняемый файл, который в любой момент времени загружен в память только частично, а оставшаяся часть хранится на диске «до востребования».

### Инструменты для сборки

Инструменты сборки ускоряют создание программы из текущих версий файлов исходного кода. Для каждого объектного файла вы указываете исходные файлы, от которых он зависит, и правила его создания. Программы сборки также устраняют ошибки несогласованности исходных данных, они гарантируют, что все файлы будут приведены в согласованное состояние. К распространенным средствам сборки относится утилита make в UNIX и ant, используемая в Java-программах.

Допустим, у вас есть объектный файл *userface.obj*. В make-файле вы определяете, что для создания *userface.obj* нужно скомпилировать файл *userface.cpp*. Вы также указываете, что *userface.cpp* зависит от *userface.h, stdlib.h* и *project.h*. «Зависит от» просто означает, что если файлы *userface.h, stdlib.h* или *project.h* изменяются, то файл *userface.cpp* надо перекомпилировать.

При сборке программы утилита make проверяет перечисленные зависимости и определяет файлы, которые необходимо перекомпилировать. Если 5 из 250 исходных файлов зависят от определения данных в *userface.h*, в котором произошли изменения, то make автоматически перекомпилирует эти зависящие от него пять файлов. Она не трогает остальные 245 файлов, не связанные с *userface.h*. Использование make или ant предоставляет альтернативу перекомпиляции всех 250 файлов или ручной компиляции отдельных файлов, чреватую пропуском одного из них и получением загадочных ошибок, появляющихся в результате рассинхронизации. Такие средства, как make или ant, заметно повышают скорость и надежность цикла компиляции, компоновки и выполнения.

В некоторых группах найдены интересные альтернативы инструментам проверки зависимостей. Так, группа, работающая над Microsoft Word, выяснила, что полная сборка всех исходных файлов проходит быстрее, чем выполнение всесторонней проверки зависимостей с помощью make при условии оптимизации самих исходных файлов (в частности, содержимого заголовочных файлов и т. п.). При таком подходе средняя машина разработчика Word может полностью собрать исполняемый файл Word — а это несколько миллионов строк кода — примерно за 13 минут.

### Библиотеки кода

Хороший способ быстро написать высококачественный код состоит в том, чтобы не писать его полностью, а найти версию с открытым исходным кодом. Вы можете найти высококачественные библиотеки для:

- контейнерных классов;
- сервисов транзакций по кредитным картам (службы e-commerce);

- межплатформенных средств разработки: вы можете написать код, который сможет выполняться в Microsoft Windows, Apple Macintosh и X Window System путем простого перекомпилирования в каждой среде;
- средств для сжатия данных;
- типов данных и алгоритмов;
- средств для работы с базами данных и манипуляций с файлами данных;
- инструментов для создания диаграмм, графиков и схем;
- средств для работы с изображениями;
- управления лицензиями;
- математических операций;
- инструментов для сетевого взаимодействия и работы в Интернете;
- генераторов отчетов и конструкторов запросов;
- средств для шифрования и обеспечения безопасности;
- инструментов для создания электронных таблиц и сеток;
- текстовых редакторов и систем проверки правописания;
- инструментов для голосовой, телефонной и факсимильной передачи данных.

### Мастера для генерации кода

Как насчет того, чтобы кто-нибудь написал код вместо вас? Вам не придется надевать желтый клетчатый пиджак и учиться тараторить, как торговец автомобилями, чтобы уговорить кого-то другого написать для вас этот код. Вы можете использовать инструменты для создания кода, и эти средства часто интегрированы в IDE.

Инструменты генерации кода обычно ориентированы на приложения для баз данных, но к этому классу относятся и другие программы. Широко распространены кодовые генераторы для баз данных, пользовательского интерфейса и компиляторов. Генерируемый ими код редко так же хорош, как написанный программистом, но во многих приложениях разработанный вручную код и не требуется. Для некоторых пользователей предпочтительней иметь 10 работающих приложений, чем одно, но работающее исключительно хорошо.

Генераторы кода также помогают при создании прототипов промышленного кода. С посомощью кодогенератора вы за несколько часов смастерите прототип программы, который позволит продемонстрировать ключевые аспекты пользовательского интерфейса, или проведете эксперименты с несколькими подходами к проектированию. При ручном кодировании такой ассортимент функциональности потребовал бы нескольких недель работы. Если вы просто экспериментируете, почему бы не делать это наиболее дешевым способом?

Наиболее распространенный недостаток кодовых генераторов в том, что созданный ими код обычно практически нечитаем. Сопровождая такой код, вы можете пожалеть о том, что сразу не написали его вручную.

### Программы установки

Средства, предназначенные для создания программ установки, обычно поддерживают работу с дискетами, CD или DVD или позволяют создавать программы установки через Web. Они проверяют, существуют ли на целевой машине общие библиотечные файлы, контролируют номера версий и т. д.

### Препроцессоры

Препроцессоры и препроцессорные макросы помогают при отладке, поскольку упрощают процесс переключения между отладочной и промышленной версиями кода. Если во время разработки вы хотите проверять фрагментацию памяти в начале каждого метода, то можете разместить там макросы. В промышленной версии вы, возможно, не захотите оставлять эти проверки — тогда можно переопределить эти макросы так, что они вообще не будут генерировать кода. По похожим причинам макросы препроцессора подходят для написания кода, предназначенного для компиляции на нескольких различных платформах — например, и в Windows, и в Linux.

**Перекрестная ссылка** О добавлении в код и удалении из кода отладочных средств см. подраздел «Запланируйте удаление отладочных средств» раздела 8.6.

Используя язык с примитивными управляющими конструкциями, например ассемблер, вы можете написать препроцессор управляющей логики, чтобы эмулировать в языке такие структурированные конструкции, как *if-then-else* и циклы *while*.

Если в вашем языке нет препроцессора, вы можете задействовать его автономный вариант как составляющую процесса сборки. Один из препроцессоров — M4 — доступен по адресу *www.gnu.org/software/m4/*.

http://cc2e.com/3091

## Отладка

Следующие средства оказывают помощь при отладке:

- предупреждающие сообщения компилятора;
- тестовые леса;
- инструменты для сравнения (позволяющие сравнивать различные версии исходных файлов);
- средства профилирования;
- мониторы трассировки;
- интерактивные отладчики — как программные, так и аппаратные.

**Перекрестная ссылка** Об этих инструментах см. раздел 23.5.

Средства тестирования, обсуждаемые далее, имеют отношение и к отладке.

## Тестирование

Следующие инструменты помогут вам провести эффективное тестирование:

- автоматизированные системы тестирования, такие как JUnit, NUnit, CppUnit и пр.;
- автоматизированные генераторы тестов;
- утилиты для записи и воспроизведения тестовых примеров;
- мониторы покрытия (анализаторы логики и средства профилирования);
- символьные отладчики;
- программы для стрессового тестирования (заполняющие оперативную память, перемешивающие содержимое памяти, генерирующие ошибки при выборочных обращениях к памяти, проверяющие доступ к памяти);
- средства сравнения (сравнивающие файлы данных, результаты работы программы и экранные изображения);

**Перекрестная ссылка** Об этих инструментах см. раздел 22.5.

- тестовые леса;
- инструменты для внесения дефектов;
- ПО для отслеживания дефектов.

## Оптимизация кода

Эти инструменты помогут вам при тонкой настройке кода.

### Средства профилирования

Профилировщик следит за работой кода и сообщает, сколько раз выполнялся каждый оператор или сколько времени программа затратила на выполнение отдельных выражений или исполняемых ветвей. Профилирование кода во время выполнения действует, как доктор, который приставляет стетоскоп к вашей груди и просит вас покашлять. Профилировщик дает вам понимание того, как работает ваша программа, где находятся узкие места и где нужно настроить код.

### Ассемблерные листинги и дизассемблеры

Однажды вам захочется взглянуть на ассемблерный код, генерируемый вашим языком высокого уровня. Некоторые компиляторы могут генерировать ассемблерные листинги. Другие — нет, и вам придется использовать дизассемблер, чтобы заново создать ассемблерный текст из машинного кода, сгенерированного компилятором. Анализ ассемблерного кода, создаваемого компилятором, показывает, насколько эффективно ваш компилятор преобразует код на языке высокого уровня в машинный код. Он может объяснить, почему высокоуровневый код, кажущийся быстродействующим, выполняется медленно. В главе 26 я приводил несколько примеров эталонных тестов, результаты которых были нелогичны. При испытаниях этого кода я часто обращался к ассемблерному листингу, чтобы лучше понять результаты, не объяснимые при анализе программы на языке высокого уровня.

Если вы плохо знакомы с языком ассемблера и хотите пройти вводный курс, то нет ничего лучше, чем сравнение каждого оператора языка высокого уровня с соответствующими ассемблерными командами, генерируемыми компилятором. Первое обращение к ассемблеру часто бывает настоящим открытием. Когда вы увидите, как много кода создает компилятор — насколько больше, чем нужно, — вы уже не сможете смотреть на свой компилятор так, как прежде.

И наоборот, в некоторых средах компилятор должен генерировать исключительно сложный код. Изучение результатов его работы помогает по достоинству оценить те усилия, которые потребовалось бы приложить при программировании на языке более низкого уровня.

# 30.4. Инструменты и среды

Некоторые платформы считаются более подходящими для работы с инструментами, чем другие.

Платформа UNIX известна своей коллекцией небольших утилит со смешными именами, которые хорошо работают вместе: grep, diff, sort, make, crypt, tar, lint, ctags, sed, awk, vi и другие. Языки C и C++, тесно связанные с UNIX, воплощают ту же

философию: стандартная библиотека C++ состоит из небольших функций, из которых легко сформировать более сложные функции, потому что все они хорошо работают в связке.

http://cc2e.com/3026

Некоторые программисты настолько продуктивно работают в UNIX, что не хотят с ней расставаться. Они используют свои любимые UNIX-подобные средства в Windows и других средах. Свой вклад в успех парадигмы UNIX внесла доступность инструментов, которые можно применять на других машинах. Так, cygwin предоставляет UNIX-подобный инструментарий для работы под Windows (*www.cygwin.com*).

Книга Эрика Реймонда (Eric Raymond) «The Art of Unix Programming» (2004) содержит углубленное обсуждение культуры программирования под UNIX.

# 30.5. Создание собственного программного инструментария

Допустим, вам дано пять часов на выполнение работы, и у вас есть выбор:

- спокойно выполнить работу за пять часов;
- потратить 4 часа и 45 минут, лихорадочно создавая утилиту для выполнения работы, а затем за 15 минут сделать работу с помощью этой утилиты.

Большинство выберет первый вариант в одном случае из миллиона, а второй — во всех остальных случаях. Создание инструментария — это один из столпов программирования. Практически все большие организации (т. е. такие, в которых работает не менее 1000 программистов) имеют группы для разработки и поддержки собственных инструментов. Во многих из них создан свой инструментарий для работы с требованиями и проектированием, превосходящий предлагаемые на рынке аналоги (Jones, 2000).

Вы можете сами написать большую часть из перечисленных в этой главе инструментов. Возможно, это и нерентабельно, но особых технических препятствий к этому нет.

## Инструментальные средства для отдельного проекта

Большинству средних и больших проектов требуются специальные инструменты, уникальные для этого проекта. Так, вам может понадобиться средство для генерации специальных видов тестовых данных, чтобы проверить качество файлов данных, или программа для эмуляции аппаратной части, которой пока нет в наличии. Ниже приведено несколько примеров поддержки в проектах специализированных средств.

- Аэрокосмическая команда отвечала за разработку ПО контроля инфракрасного датчика и анализа его показаний. Для проверки производительности программы записывающее устройство документировало действия, выполняемые программой во время полета. Инженеры написали собственные средства анализа данных с целью оценить производительность полетных систем.

- Microsoft планировала включить новую технологию работы со шрифтами в графическую среду Windows. Поскольку и файлы с информацией о шрифтах, и ПО для отображения этих шрифтов были новыми, ошибки могли возникать как в данных, так и в программе. Разработчики Microsoft написали несколько специализированных утилит, которые проверяли наличие ошибок в файлах данных, что позволило проще различать места возникновения ошибок: в файлах данных или в ПО.
- Страховая компания разработала большую систему для расчета повышений ставок. Поскольку система была сложной, а к точности предъявлялись повышенные требования, сотни рассчитанных ставок необходимо было тщательно проверить, хотя вычисление одной ставки вручную занимало несколько минут. Компания написала отдельное программное средство для расчета одной ставки, позволяющее делать это за несколько секунд и проверить ставки, полученные из основной программы гораздо быстрее, чем при проверке ставок вручную.

Планируя проект, необходимо подумать об инструментах, которые могут понадобиться, и выделить время для их создания.

## Сценарии

Сценарий — это инструмент автоматизации повторяющихся рутинных операций. В некоторых системах сценариями называются командные файлы или макросы. Сценарии могут быть простыми и сложными, а наиболее полезные из них очень легко написать. Например, я веду дневник и для защиты конфиденциальной информации всегда держу его в зашифрованном виде, кроме тех моментов, когда делаю в нем записи. Чтобы быть уверенным, что я всегда шифрую и дешифрую его надлежащим образом, я использую сценарий, который дешифрует мой дневник, запускает текстовый процессор, а затем шифрует дневник снова. Сценарий может выглядеть так:

```
crypto c:\word\journal.* %1 /d /Es /s
word  c:\word\journal.doc
crypto c:\word\journal.* %1 /Es /s
```

Поле *%1* предназначено для пароля, который по понятным причинам в сценарии не записан. Сценарий делает за меня работу (причем без ошибок) по вводу всех параметров и гарантирует, что я всегда выполню все операции, причем в правильном порядке.

Если вы обнаружите, что набираете строку длиннее пяти символов по многу раз в день, то эта процедура — хороший кандидат для использования в сценарии или командном файле. В качестве примеров можно привести последовательности компиляции/компоновки, команды резервного копирования и любые команды с большим количеством параметров.

## 30.6. Волшебная страна инструментальных средств

**Перекрестная ссылка** Доступность инструментария зависит от степени развития технологической среды (см. раздел 4.3).

Десятилетиями поставщики инструментария и ученые мужи обещают, что создание средств, которые позволят отказаться от программирования, не за горами. Первым и, кажется, самым забавным случаем присвоения этого ярлыка, был язык Fortran. Fortran (Formula Translation Language, язык преобразований формул) задумывался как средство, которое даст ученым и инженерам возможность просто набирать формулы и, таким образом, обойтись без помощи программистов.

Fortran действительно позволил ученым и инженерам писать программы, но с сегодняшней точки зрения он выглядит сравнительно низкоуровневым языком программирования. Он едва ли позволил обойтись без программистов, а опыт, полученный при работе с Fortran, послужил прогрессу отрасли ПО в целом.

Индустрия ПО постоянно разрабатывает новые инструменты, которые уменьшают влияние (или совсем исключают) некоторых наиболее утомительных аспектов программирования: деталей размещения операторов исходного кода; шагов, предпринимаемых для редактирования, компиляции, компоновки и выполнения программы; работы по поиску несогласованных скобок; действий по созданию стандартных текстовых сообщений и т. д. Поскольку каждое из этих новых средств начинает демонстрировать выигрыш в производительности, ученые мужи экстраполируют этот выигрыш до бесконечности, предполагая, что благодаря ему в конце концов «исчезнет необходимость в программировании». Но на самом деле каждая инновация содержит некоторые изъяны. С течением времени изъяны исправляются, и потенциал инновации реализуется полностью. Однако после реализации фундаментальной концепции инструмента дальнейший выигрыш достигается путем удаления случайных трудностей, возникших в качестве побочного эффекта при создании нового инструмента. Устранение этих случайных проблем по сути не увеличивает производительность, а просто исключает «шаг назад» из типичного уравнения «два шага вперед, один шаг назад».

За последние десятилетия программисты видели массу инструментов, которые предположительно должны были устранить необходимость программирования. Сначала это были языки третьего поколения, потом — четвертого. Потом — автоматическое программирование. Потом — CASE-средства. Потом — визуальное программирование. Каждое из этих достижений привносило значительные улучшения, и общими усилиями они сделали программирование абсолютно неузнаваемым для тех, кто изучал его до этих нововведений. Но ни одна из этих инноваций не устранила программирования как такового.

**Перекрестная ссылка** О причинах сложности программирования см. подраздел «Существенные и несущественные проблемы» раздела 5.2.

Причина в том, что программирование — принципиально *сложный* процесс даже при наличии хорошего инструментария. Дело не в инструментах — программистам приходится бороться с несовершенством реального мира; нам нужно досконально продумывать последовательности, зависимости и исключения, иметь дело с конечными пользователями, которые никак не могут ничего решить. Нам всегда придется бороться с пло-

хо определенными интерфейсами с другими программными и аппаратными средствам и всегда принимать во внимание инструкции, бизнес-правила и другие источники сложных проблем, возникающие вне мира программирования.

Нам всегда будут нужны люди, способные заполнить брешь между задачей реального мира, которую нужно решить, и компьютером, предназначенным для решения этой задачи. Эти люди будут называться программистами независимо от того, манипулируют они машинными регистрами на ассемблере или диалоговыми окнами в Microsoft Visual Basic. Пока у нас есть компьютеры, нам будут нужны люди, которые говорят компьютерам, что́ делать, и эта деятельность будет называться программированием.

Когда вы слышите заявления о том, что «новый инструментарий устранит необходимость компьютерного программирования», бегите! Или хотя бы посмейтесь про себя над этим наивным оптимизмом.

## Дополнительные ресурсы

Дополнительную информацию о программном инструментарии содержат следующие источники.

http://cc2e.com/3098

*www.sdmagazine.com/jolts*. Web-сайт, посвященный ежегодной премии Jolt Productivity журнала «Software Development Magazine», — хороший источник информации о лучших на сегодняшний день инструментах.

http://cc2e.com/3005

Hunt, Andrew and David Thomas. *The Pragmatic Programmer*. Boston, MA: Addison-Wesley, 2000. Раздел 3 этой книги содержит всестороннее исследование программного инструментария, включая редакторы, кодогенераторы, отладчики, системы управления версиями и другие аналогичные инструменты.

http://cc2e.com/3012

Vaughn-Nichols, Steven. «Building Better Software with Better Tools», *IEEE Computer*, September 2003, pp. 12–14. В этой статье приводится обзор инициатив в области разработки инструментальных средств, выдвинутых IBM, Microsoft Research и Sun Research.

Glass, Robert L. *Software Conflict: Essays on the Art and Science of Software Engineering*. Englewood Cliffs, NJ: Yourdon Press, 1991. В главе «Recommended: A Minimum Standard Software Toolset» в противовес мнению о том, что чем больше инструментов, тем лучше, автор ратует за определение минимального набора инструментов, который должен быть доступен каждому разработчику и предлагаться в виде стартового комплекта.

Jones, Capers. *Estimating Software Costs*. New York, NY: McGraw-Hill, 1998.

Boehm, Barry, et al. *Software Cost Estimation with Cocomo II*. Reading, MA: Addison-Wesley, 2000. Книги Джонса и Бома содержат разделы, посвященные влиянию инструментальных средств на производительность.

**Контрольный список: программный инструментарий**

http://cc2e.com/3019

- ❑ Используете ли вы эффективную IDE?
- ❑ Поддерживает ли ваша IDE интеграцию с системой управления исходным кодом, средствами компоновки, тестирования и отладки и другую полезную функциональность?
- ❑ Есть ли у вас инструменты для автоматизации часто выполняемого рефакторинга?
- ❑ Используете ли вы систему управления версиями для управления исходным кодом, содержанием, требованиями, результатами проектирования, планами работ и другими элементами проекта?
- ❑ Используете ли вы при работе над очень большим проектом словарь данных или другой центральный репозиторий, содержащий официальное описание каждого класса, пирменяемого в системе?
- ❑ Рассматривался ли вопрос использования библиотек кода в качестве альтернативы написанию собственного кода там, где это возможно?
- ❑ Пользуетесь ли вы интерактивным отладчиком?
- ❑ Используете ли вы make или другое ПО для контроля зависимостей, чтобы создавать программы эффективно и надежно?
- ❑ Содержит ли ваша среда тестирования инструменты для автоматического тестирования, генераторы тестов, мониторы покрытия, средства для стрессового тестирования, инструменты для сравнения и ПО для отслеживания дефектов?
- ❑ Создаете ли вы специальные средства, способные помочь в поддержке специальных нужд проекта, особенно инструменты, автоматизирующие повторяющиеся задания?
- ❑ Получает ли ваша среда преимущества от применения надлежащего инструментария?

# Ключевые моменты

- Хороший инструментарий может значительно облегчить вам жизнь.
- Можно легко приобрести инструменты для редактирования, анализа качества кода, рефакторинга, управления версиями, отладки, тестирования и настройки кода.
- Вы можете создать множество инструментов специального назначения.
- Хорошие инструменты могут упростить наиболее утомительные аспекты разработки ПО, но они не могут исключить необходимость программирования, хотя и способствуют эволюции того понятия, которое мы вкладываем в слово «программирование».

Часть VII

# МАСТЕРСТВО ПРОГРАММИРОВАНИЯ

ГЛАВА 31

# Форматирование и стиль

http://cc2e.com/3187

**Содержание**



**Связанные темы**

- Самодокументируемый код: глава 32
- Инструменты для форматирования кода: «Редактирование» в разделе 30.2

В этой главе рассматривается эстетический аспект программирования — форматирование исходного кода программы. Зрительное и интеллектуальное наслаждение, получаемое от хорошо отформатированного кода, — удовольствие, которое способны оценить лишь немногие непрограммисты. Но программисты, гордящиеся своей работой, получают огромное эстетическое удовлетворение от процесса шлифовки визуальной структуры кода.

Методики, описанные в этой главе, не влияют на скорость выполнения, объем памяти и другие внешние аспекты программы. Но от них зависит, насколько легко вы сможете понять, пересмотреть и исправить этот код спустя несколько месяцев после его создания. От них также зависит, насколько легко другие смогут его прочесть, понять и изменить в ваше отсутствие.

Эта глава полна мелких подробностей, которые обычно имеют в виду, говоря о «внимании к деталям». В течение жизни проекта внимание к таким деталям влияет на качество и итоговое удобство сопровождения создаваемого кода. Они слишком интегрированы в процесс кодирования, чтобы их можно было эффективно изменить потом. Если вы вообще собираетесь их учитывать, учтите их при пер-

воначальном конструировании. Перед началом работы над групповым проектом, дайте своей команде прочитать эту главу и согласуйте общий стиль кодирования.

Вы можете согласиться не со всем, что здесь прочтете, но я стремился не столько добиться полного одобрения моего мнения, сколько убедить вас рассмотреть вопросы, связанные со стилем форматирования. Если вы гипертоник, переходите к следующей главе — она вызовет меньше споров.

# 31.1. Основные принципы форматирования

В этом разделе мы обсудим теорию хорошего форматирования. Остальная часть главы посвящена практике.

## Экстремальные случаи форматирования

Посмотрите на метод, приведенный в листинге 31-1:

**Листинг 31-1.**
**Пример № 1 форматирования кода (Java)**

```
/* Используем способ сортировки методом вставок для сортировки массива «data» в
возрастающем порядке. Этот метод предполагает, что [ firstElement ] не является
первым элементом данных и элемент data[ firstElement-1 ] достижим. */ public void
InsertionSort( int[] data, int firstElement, int lastElement ) { /* Заменяем
элемент, расположенный на нижней границе интервала, элементом, который гаранти-
рованно будет первым в сортированном списке. */ int lowerBoundary = data
[ firstElement-1 ]; data[ firstElement-1 ] = SORT_MIN; /* Элементы в позициях от
firstElement до sortBoundary-1 всегда сортированы. При каждом проходе цикла sort-
Boundary увеличивается, и элемент, соответствующий новому sortBoundary, возможно,
будет неправильно отсортирован, поэтому он вставляется в надлежащую позицию массива
где-то между firstElement и sortBoundary. */ for ( int sortBoundary =
firstElement+1; sortBoundary <= lastElement; sortBoundary++  ) { int insertVal =
data[ sortBoundary ]; int insertPos = sortBoundary; while ( insertVal < data[
insertPos-1 ] ) { data[ insertPos ] = data[ insertPos-1 ]; insertPos = insertPos-1;
} data[ insertPos ] = insertVal; } /* Возвращаем исходное значение элементу,
расположенному на нижней границе интервала */ data[ firstElement-1 ] = lowerBoundary; }
```

Этот метод синтаксически корректен. Он прокомментирован и содержит хорошие имена переменных и понятную логику. Если не верите, прочтите его и найдите ошибку! Чего не хватает этому методу, так это хорошего форматирования. Это крайний случай, стремящийся к «минус бесконечности» на шкале способов форматирования с диапазоном от плохих до хороших. Листинг 31-2 показывает менее экстремальный вариант:

**Листинг 31-2.**
**Пример № 2 форматирования кода (Java)**

```
/* Используем способ сортировки методом вставок для сортировки массива «data» в
возрастающем порядке. Этот метод предполагает, что [ firstElement ] не является
первым элементом данных и элемент data[ firstElement-1 ] достижим. */
public void InsertionSort( int[] data, int firstElement, int lastElement ) {
/* Заменяем элемент, расположенный на нижней границе интервала, элементом,
```

```
который гарантированно будет первым в сортированном списке. */
int lowerBoundary = data[ firstElement-1 ];
data[ firstElement-1 ] = SORT_MIN;
/* Элементы в позициях от firstElement до sortBoundary-1 всегда отсортированы. При
каждом проходе цикла sortBoundary увеличивается, и элемент, соответствующий новому
каждом проходе цикла sortBoundary увеличивается, и элемент, соответствующий новому
sortBoundary, возможно, будет неправильно отсортирован, поэтому он вставляется
в надлежащую позицию массива между firstElement и sortBoundary. */
for (
int sortBoundary = firstElement+1;
sortBoundary <= lastElement;
sortBoundary++
) {
int insertVal = data[ sortBoundary ];
int insertPos = sortBoundary;
while ( insertVal < data[ insertPos-1 ] ) {
data[ insertPos ] = data[ insertPos-1 ];
insertPos = insertPos-1;
}
data[ insertPos ] = insertVal;
}
/* Возвращаем исходное значение элементу, расположенному на нижней границе интервала
*/
data[ firstElement-1 ] = lowerBoundary;
}
```

Это тот же код, что и в листинге 31-1. Хотя большинство согласится, что такое форматирование кода гораздо лучше первого примера, но код все еще не слишком читабелен. Текст еще расположен слишком кучно и не содержит подсказок о логической организации метода. Такое форматирование соответствует 0 на шкале вариантов форматирования. Первый пример был несколько надуман, но второй не так уж редко встречается в жизни. Я видел программы, состоящие из нескольких тысяч строк кода, форматированные столь же плохо, как и здесь. А при отсутствии документации и плохих именах переменных общая читабельность была еще хуже, чем в этом примере. Этот код отформатирован для компьютера, и нет никаких свидетельств, что автор думал о людях, которые будут читать его код. Листинг 31-3 содержит усовершенствованный вариант.

### Листинг 31-3. Пример № 3 форматирования кода (Java)

```
/* Используем способ сортировки методом вставок для сортировки массива «data» в
возрастающем порядке. Этот метод предполагает, что [ firstElement ] не является
первым элементом данных и элемент data[ firstElement-1 ] достижим.
*/

public void InsertionSort( int[] data, int firstElement, int lastElement ) {
   // Заменяем элемент, расположенный на нижней границе интервала, элементом,
   // который гарантированно будет первым в отсортированном списке.
   int lowerBoundary = data[ firstElement-1 ];
   data[ firstElement-1 ] = SORT_MIN;
```

```
   /* Элементы в позициях от firstElement до sortBoundary-1 всегда отсортированы.
   При каждом проходе цикла sortBoundary увеличивается,
   и элемент, соответствующий новому sortBoundary,
   возможно, будет неправильно отсортирован,
   поэтому он вставляется в надлежащую позицию массива
   где-то между firstElement и sortBoundary.
   */
   for ( int sortBoundary = firstElement + 1; sortBoundary <= lastElement;
      sortBoundary++ ) {
      int insertVal = data[ sortBoundary ];
      int insertPos = sortBoundary;
      while ( insertVal < data[ insertPos - 1 ] ) {
         data[ insertPos ] = data[ insertPos - 1 ];
         insertPos = insertPos - 1;
      }
      data[ insertPos ] = insertVal;
   }

   // Возвращаем исходное значение элементу,
   // расположенному на нижней границе интервала
   data[ firstElement - 1 ] = lowerBoundary;
}
```

Это форматирование соответствует строго положительному значению на шкале вариантов форматирования с диапазоном от плохих до хороших. Теперь этот метод организован в соответствии с принципами, объясняемыми на протяжении всей этой главы. Метод гораздо удобнее читать, а усилия, приложенные к документированию и выбору хороших имен переменных, теперь очевидны. Имена переменных и в прошлых примерах были столь же хороши, однако форматирование было настолько плохим, что от них не было пользы.

Единственное различие между этим и первыми двумя примерами заключается в использовании пробелов и разделителей — код и комментарии абсолютно одинаковы. Пробелы нужны исключительно людям — ваш компьютер смог бы прочесть все три фрагмента одинаково легко. Не расстраивайтесь, если не можете сделать это не хуже вашего компьютера!

## Основная теорема форматирования

Основная теорема форматирования гласит, что хорошее визуальное форматирование показывает логическую структуру программы.

Создание красивого кода — хорошо, а демонстрация структуры кода — лучше. Если одна методика лучше показывает структуру кода, а другая выглядит красивей, используйте ту, которая лучше демонстрирует структуру. Эта глава содержит массу примеров стилей форматирования, выглядящих хорошо, но искажающих логическую организацию кода. Отдавая на практике предпочтение логическому представлению, обычно нельзя получить уродливый код, если только сама логика этого кода не уродлива. Методики, позволяющие хорошему коду выглядеть хорошо, а плохому — плохо, полезней, чем методики, позволяющие любому коду выглядеть хорошо.

## Интерпретация программы человеком и компьютером

> Любой дурак может написать код, понятный компьютеру. Хорошие программисты пишут код, понятный людям.
>
> *Мартин Фаулер (Martin Fowler)*

Форматирование — это ключ к структуре программы. Компьютеру важна исключительно информация о скобках или операторах *begin* и *end*, а читатель-человек склонен делать выводы из визуального представления кода. Взгляните на фрагмент кода, приведенный в листинге 31-4, схема отступов в котором заставляет человека думать, что все три выражения выполняются при каждом проходе цикла.

**Листинг 31-4. Пример форматирования, рассказывающего разные истории человеку и компьютеру (Java)**

```
// меняем местами правые и левые элементы во всем массиве
for ( i = 0; i < MAX_ELEMENTS; i++ )
   leftElement = left[ i ];
   left[ i ]   = right[ i ];
   right[ i ]  = leftElement;
```

Если код не содержит обрамляющих скобок, компилятор будет выполнять первое выражение *MAX_ELEMENTS* раз, а второе и третье — по одному разу. Отступы делают очевидным и для вас, и для меня, что автор кода хотел выполнять все три выражения и собирался поместить скобки вокруг них. Компилятору это неочевидно. Листинг 31-5 содержит другой пример:

**Листинг 31-5. Другой пример форматирования, рассказывающего разные истории человеку и компьютеру (Java)**

```
x = 3+4 * 2+7;
```

Человек, читающий этот код, будет склонен интерпретировать это выражение так, что переменной *x* присваивается значение *(3+4) * (2+7)*, т. е. *63*. Компьютер проигнорирует пробелы и подчинится правилам приоритета, интерпретируя это выражение, как *3 + (4*2) + 7* или *18*. Идея в том, что хорошая схема форматирования приведет в соответствие визуальную и логическую структуру программы, т. е. расскажет одну и ту же историю и человеку, и компьютеру.

## Насколько важно хорошее форматирование?

*Знание схем программирования и правил изложения программы может существенно повлиять на возможность осмысления программы. В книге «[The] Elements of [Programming] Style» Керниган и Плоджер (Kernighan and Plauger) также определяют то, что мы назвали правилами изложения. Наши эмпирические результаты подтверждают эти правила: необходимость написания программ в определенном стиле вызвана не только эстетическими соображениями. Создание программ в определенной манере имеет скорее психологическую подоплеку: программисты возлагают большие надежды на то, что другие разработчики будут следовать их правилам изложения. Если эти правила нарушаются, то вся польза, на которую надеется программист, сводится к нулю. Эти выводы подтверждаются результатами экспериментов с начи-*

*нающими и подготовленными студентами-программистами, а также с опытными разработчиками, упоминаемыми в этой статье.*

*—Эллиот Соловей и Кейт Эрлих (Elliot Soloway and Kate Ehrlich)*

**Перекрестная ссылка** Хорошее форматирование — один из ключей к удобочитаемости (см. раздел 34.3).

В вопросах форматирования, возможно, сильнее, чем в других аспектах программирования, проявляется разница между взаимодействием с компьютером и взаимодействием с человеком. Меньшая задача программирования состоит в написании программы в понятном для компьютера виде, бóльшая — в том, чтобы написать ее так, чтобы другие люди могли ее прочесть.

В классической статье «Perception in Chess» Чейз и Саймон (Chase and Simon) сообщили об исследовании, в котором сравнивались способности экспертов и новичков запоминать позиции фигур в шахматах (1973). Когда фигуры были расставлены так, как это могло сложиться во время игры, память экспертов во много раз превосходила способности начинающих. Когда же фигуры были расставлены случайно, то результаты экспертов и новичков отличались не намного. Традиционно этот результат объясняется тем, что память эксперта несущественно отличается от памяти начинающих, но эксперты используют систему знаний, помогающую им запоминать определенные виды информации. Если новые данные соответствуют этой системе знаний (в данном случае осмысленному размещению шахматных фигур), эксперты легко могут их запомнить. Если же новая информация в эту систему не укладывается (шахматные фигуры расположены в случайном порядке) эксперт может запомнить ее ничуть не лучше новичка.

Несколько лет спустя Бен Шнейдерман (Ben Shneiderman) повторил результаты Чейза и Саймона в сфере компьютерного программирования и сообщил о них в статье «Exploratory Experiments in Programmer Behavior» (1976). Шнейдерман выяснил, что если программные операторы расположены в осмысленном порядке, то эксперты могли запомнить их лучше, чем новички. Когда же операторы перемешивались, преимущество экспертов снижалось. Результаты Шнейдермана были подтверждены и в других исследованиях (McKeithen et al., 1981; Soloway and Ehrlich, 1984). Основная концепция подтверждалась в играх го и бридже, а также в электронике, музыке и физике (McKeithen et al., 1981).

После публикации первого издания этой книги Хенк — один из программистов, рецензировавших рукопись, — сказал: «Я удивился, что ты недостаточно активно ратовал за использование такого стиля скобок:

```
for ( ...)
   {
   }
```

Странно, что ты вообще упомянул такой стиль скобок, как этот:

```
for ( ...) {
}
```

Я думал, что, поскольку и я, и Тони приводили доводы в пользу первого стиля, ты предпочтешь его».

Я ответил: «Ты, наверное, имел в виду, что ты ратовал за применение первого стиля, а Тони — второго, не так ли? Тони приводил доводы в пользу второго стиля, а не первого».

Хенк ответил: «Забавно. В последнем проекте, над которым Тони и я работали вместе, я предпочитал использовать стиль №2, а Тони — №1. Весь проект мы спорили о том, какой стиль лучше. Полагаю, мы уговорили друг друга предпочесть противоположный стиль!»

Этот случай, а также исследования, процитированные выше, позволяют предположить, что структура помогает экспертам воспринимать, осмысливать и запоминать важные свойства программ. Опытные программисты часто упорно цепляются за собственный стиль, даже если он очень сильно отличается от стиля, применяемого другими экспертами. Но практический результат показывает, что детали конкретного способа структурирования программы гораздо менее важны, чем сам факт единообразного структурирования программы.

## Форматирование как религия

Важное влияние, которое привычный способ структурирования среды оказывает на процесс понимания и запоминания, заставило исследователей выдвинуть такую гипотезу: если форматирование отличается от схемы, используемой экспертами, оно может повредить их способности читать программу (Sheil, 1981; Soloway and Ehrlich, 1984). Такая возможность вкупе с не только логическим, но и эстетическим значением форматирования привела к тому, что дебаты вокруг форматирования программ часто больше похожи на религиозные войны, а не на философские диспуты.


**Перекрестная ссылка** Если вы смешиваете вопросы разработки ПО и религии, прочтите раздел 34.9, прежде чем продолжить чтение остальной части этой главы.

Грубо говоря, очевидно, что некоторые виды форматирования лучше других. Последовательное улучшение фрагментов одного и того же кода, показанное в начале этой главы, делает это утверждение бесспорным. В этой книге я не буду избегать деликатных вопросов о форматировании только потому, что они спорны. Хорошие программисты должны непредвзято относиться к привычным для них способам форматирования и признавать другие варианты, доказавшие свое преимущество по отношению к использовавшимся ранее, даже если при переходе к новым методам возникнет некоторый начальный дискомфорт.

## Цели хорошего форматирования

Многие решения о том, как должно выглядеть хорошее форматирование, представляют собой субъективные эстетические оценки; часто можно достичь одной и той же цели поразному. Вы можете сделать споры о субъективных вопросах менее субъективными, если явно укажете критерии ваших предпочтений. Говоря объективно, хорошая схема форматирования должна делать следующие вещи.

> Эксперименты выявили хрупкость программистской квалификации: опытные программисты имеют *стойкие* представления о том, как должны выглядеть программы, и, если эти убеждения нарушаются — казалось бы, даже безобидными способами, — их производительность радикально ухудшается.
>
> *Эллиот Соловей и Кейт Эрлих (Elliot Soloway and Kate Ehrlich)*

***Точно представлять логическую структуру кода*** Снова повторим Основную теорему форматирования: главная цель хорошего форматирования — показать логическую структуру кода. Для демонстрации логической структуры программисты обычно применяют отступы и другие неотображаемые символы.

***Единообразно показывать логическую структуру кода*** Некоторые стили форматирования состоят из правил с таким количеством исключений, что последовательно их соблюдать практически невозможно. Действительно хороший стиль подходит в большинстве случаев.

***Улучшать читабельность*** Стратегия использования отступов, соответствующая логике, но усложняющая процесс чтения кода, бесполезна. Схема форматирования, использующая пробелы и разделители только там, где они требуются компилятору, логична, но читать такой код невозможно. Хорошая структура форматирования упрощает чтение кода.

***Выдерживать процедуру исправления*** Лучшие схемы форматирования хорошо переносят модификацию кода. Исправление одной строки не должно приводить к изменению нескольких других.

В дополнение к этим критериям иногда во внимание принимается и задача минимизации количества строк кода, необходимых для реализации простого выражения или блока.

### Как воспользоваться принципами хорошего форматирования на практике

Критерии хорошей схемы форматирования могут служить основой для обсуждения возможных вариантов форматов, помогая отличить субъективные причины в предпочтении одного стиля форматирования перед другим

Оценка критерия с нескольких точек зрения может привести к разным выводам. Так, если вы твердо убеждены, что минимизация количества строк кода очень важна (вероятно, потому, что у вас маленький монитор), вы можете критиковать один стиль только за то, что для списка параметров метода он использует на две строки больше, чем какой-то другой.

# 31.2. Способы форматирования

Вы можете получить хороший формат кода, по-разному используя несколько инструментов для форматирования.

## Неотображаемые символы

Используйтенеотображаемыесимволыдляулучшениячитаемости. Неотображаемые символы, к которым относятся пробелы, знаки табуляции, переводы строк и пустые строки, — это основное средство для демонстрации структуры программы.

**Перекрестная ссылка** Некоторые исследователи проводят аналогии между структурой книги и структурой программы (см. подраздел «Книжная парадигма документирования программ» раздела 32.5).

Вам вряд ли придет в голову писать книгу без пробелов между словами, разбиения на абзацы и деления на главы. Может, такую книгу и можно прочитать от начала до конца, но практически невозможно просматривать ее в поисках какой-то мысли или важного эпизода. Еще хуже, что такой формат книги не позволит показать читателю, как автор намеревался организовать информацию. Структура, предлагаемая автором, дает подсказку о логической организации темы.

Разбиение книги на главы, абзацы и предложения показывает читателю, как следует мысленно организовывать тему. Если эта организация неочевидна, читателю приходится самому ее домысливать, что налагает на него более тяжкое бремя и увеличивает вероятность никогда не узнать, как на самом деле организована данная тема.

Информация, содержащаяся в программе, сосредоточена еще плотней, чем информация в большинстве книг. Если страницу книги можно прочесть и понять за 1 или 2 минуты, то большинство программистов не могут читать и понимать листинг программы со скоростью, даже приблизительно сравнимой с этой. Программа должна давать гораздо больше подсказок о своей организации, чем книга.

***Группировка*** Еще один способ применения неотображаемых символов — группировка взаимосвязанных выражений

В литературе мысли группируются в абзацы. Хорошо написанный абзац содержит предложения, относящиеся только к определенной идее. Он не должен содержать посторонних предложений. Точно так же абзац кода должен содержать только взаимосвязанные операторы, выполняющие одно задание.

***Пустые строки*** Кроме необходимости группировать взаимосвязанные операторы, очень важно отделять несвязанные выражения друг от друга. Начало нового абзаца в книге обозначается отступом или пустой строкой. Начало нового абзаца в коде нужно указывать с помощью пустой строки.

Пустые строки позволяют продемонстрировать организацию программы. Вы можете использовать их для деления групп взаимосвязанных операторов на абзацы, отделения методов друг от друга и выделения комментариев.

Хотя эту статистику тяжело применить на практике, но одно исследование показало, что оптимальное число пустых строк в программе составляет от 8% до 16%. Если оно больше 16%, то время, затрачиваемое на отладку, заметно увеличивается (Gorla, Benander and Benander, 1990).

***Отступы*** Применяйте отступы для демонстрации логической структуры программы. Как правило, операторы выделяются отступами, когда они следуют после некоторого выражения, от которого они логически зависят.

Существуют данные, что отступы влияют на способность программиста понимать код. В статье «Program Indentation and Comprehensibility» сообщается, что некоторые исследования выявили корреляцию между наличием отступов и способностью к пониманию кода (Miaria et al., 1983). В тесте на понимание испытуемые показали результат на 20–30% лучше, когда программы использовали схему отступов из 2–4-х пробелов, чем когда программы вообще не содержали отступов.

То же исследование выявило, что нельзя не только недостаточно выделять, но и чрезмерно подчеркивать логическую структуру программы. Меньше всего баллов за понимание получили программы, совсем не содержащие отступов. Второй с конца результат принадлежал программам, использующим отступы из 6 пробелов. Авторы пришли к выводу, что оптимальными являются отступы из 2–4-х пробелов. Интересно, что многим испытуемым отступы из 6 пробелов показались удобнее, чем другие, даже несмотря на то, что окончательный результат оказался хуже. Возможно, это связано с тем, что отступы из 6 пробелов выглядят приятнее. Но независимо от того, насколько красиво они выглядят, отступы из 6 пробелов оказались хуже читаемыми. Это один из примеров коллизии между эстетикой и читабельностью.

### Скобки

Используйте скобки чаще, чем вам это кажется необходимым. Применяйте скобки для разъяснения выражений, состоящих из двух и более членов. Возможно, в скобках нет нужды, но они добавляют ясности и ничего вам не стоят. Например, скажите, как вычисляются следующие выражения?

```
Вариант на C++: 12 + 4% 3 * 7 / 8
Вариант на Microsoft Visual Basic: 12 + 4 mod 3 * 7 / 8
```

Пришлось ли вам задуматься о том, как эти выражения вычисляются, вот в чем вопрос? Можете ли вы быть уверенными в своем ответе без обращения к справочной информации? Даже опытные программисты не отвечают с полной уверенностью, и именно поэтому следует использовать скобки, если есть хоть малейшее сомнение в том, как вычисляется выражение.

## 31.3. Стили форматирования

Большинство вопросов форматирования касается размещения блоков — групп операторов, располагающихся под управляющими выражениями. Блок окружен скобками или ключевыми словами: *{* и *}* в C++ и Java, *if-then-endif* в Visual Basic и другими похожими структурами в других языках. Для простоты в большей части этого обсуждения используются общие обозначения *begin* и *end*. Я предполагаю, что вы поймете, как это можно соотнести со скобками в C++ и Java или аналогичными механизмами выделения блоков в других языках. Ниже описаны четыре основных стиля форматирования:

- явные блоки;
- эмуляция явных блоков;
- использование пар *begin-end* (скобок) для обозначения границ блока;
- форматирование в конце строки.

## Явные блоки

Большинство споров по поводу форматирования возникает из-за несовершенства большинства популярных языков программирования. Хорошо спроектированный язык имеет явную структуру блоков, которая приводит к естественному стилю отступов. Так, в Visual Basic у каждой управляющей структуры есть свой терминатор, и вы не сможете ее использовать без этого терминатора. Код разбивается на блоки естественным образом. Несколько примеров на Visual Basic приведено в листингах 31-6, 31-7 и 31-8:

**Листинг 31-6. Пример явного блока *if* (Visual Basic)**

```
If pixelColor = Color_Red Then
   statement1
   statement2
   ...
End If
```

**Листинг 31-7. Пример явного блока *while* (Visual Basic)**

```
While pixelColor = Color_Red
   statement1
   statement2
   ...
Wend
```

**Листинг 31-8. Пример явного блока *case* (Visual Basic)**

```
Select Case pixelColor
   Case Color_Red
      statement1
      statement2
      ...
   Case Color_Green
      statement1
      statement2
      ...
   Case Else
      statement1
      statement2
      ...
End Select
```

Управляющая структура на Visual Basic всегда состоит из начального выражения (в этих примерах — *If-Then*, *While* и *Select-Case)* и всегда содержит соответствую-

щий оператор *End*. Отступы внутри такой структуры не подвергаются сомнению, а варианты выравнивания других ключевых слов частично ограничены. Листинг 31-9 показывает абстрактное представление того, как выглядит такой вид форматирования:

**Листинг 31-9. Абстрактный пример стиля форматирования явного блока**


В этом примере управляющая конструкция начинается оператором A и кончается оператором D. Выравнивание между этими двумя операторами обеспечивает визуальное единство конструкции.

Дебаты по поводу форматирования управляющих структур частично возникают потому, что некоторые языки не *требуют* применения блоковых структур. Вы можете использовать оператор *if-then*, за которым следует единственное выражение, а не формальный блок. Для создания блока вам приходится добавлять пару *begin-end* или открывающую и закрывающую скобки вместо их автоматического получения с каждой управляющей структурой. Отделение *begin* и *end* от самой структуры — так, как языки, подобные C++ и Java, делают это со скобками *{* и *}*, — приводит к вопросу: куда поместить эти *begin* и *end*? Поэтому меньше проблемы с отступами становятся проблемами только потому, что вам приходится компенсировать недостатки плохо спроектированных языковых структур. Способы такой компенсации описаны ниже.

## Эмуляция явных блоков

Хорошим подходом в языках, не имеющих явных блоков, будет рассмотрение ключевых слов *begin* и *end* (или символов *{* и *}*) в виде расширений управляющих структур, с которыми они используются. Далее, имеет смысл попробовать сэмулировать форматирование языка Visual Basic на вашем языке. Листинг 31-10 содержит абстрактное представление визуальной структуры, которую вы пытаетесь эмулировать:

**Листинг 31-10. Абстрактный пример стиля форматирования явного блока**


При таком стиле управляющая конструкция открывает блок в операторе A и закрывает блок в операторе D. Это предполагает, что *begin* должен находиться в конце оператора A, а *end* должен быть оператором D. Говоря абстрактно, для эмуляции явных блоков вам надо сделать нечто, подобное листингу 31-11:

**Листинг 31-11. Абстрактный пример эмуляции стиля явного блока**

```
A   ▮▮▮▮▮▮▮▮▮▮▮▮▮▮{
B       ▮▮▮▮▮▮▮▮▮▮▮▮▮
C       ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
D   }
```

Несколько примеров такого стиля на C++ приведено в листингах 31-12, 31-13 и 31-14:

**Листинг 31-12. Пример эмуляции явного блока *if* (C++)**

```
if ( pixelColor == Color_Red ) {
   statement1;
   statement2;
   ...
}
```

**Листинг 31-13. Пример эмуляции явного блока *while* (C++)**

```
while ( pixelColor == Color_Red ) {
   statement1;
   statement2;
   ...
}
```

**Листинг 31-14. Пример эмуляции явного блока *switch/case* (C++)**

```
switch ( pixelColor ) {
   case Color_Red:
      statement1;
      statement2;
      ...
   break;
   case Color_Green:
      statement1;
      statement2;
      ...
   break;
   default:
      statement1;
      statement2;
      ...
   break;
}
```

Этот стиль выравнивания вполне функционален. Он хорошо выглядит, его можно применять единообразно, а также его удобно сопровождать. Он соответствует Основной теореме форматирования в том плане, что помогает показать логическую структуру кода. Это вполне разумный вариант стиля. Такой стиль является стандартом в Java и широко распространен в C++.

## Использование пар *begin-end* (скобок) для обозначения границ блока

Альтернативой структуре явного блока может служить использование пар *begin-end* в качестве границ блока. (В дальнейшем обсуждении пары *begin-end* применяются для общего обозначения операторов *begin-end*, скобок и других эквивалентных языковых структур.) Если вы берете за основу этот подход, то рассматриваете *begin* и *end* как операторы, следующие за управляющей структурой, а не как фрагменты, являющиеся ее частью. Графически это выглядит идеально, так же, как и при эмуляции явного блока, приводимой еще раз в листинге 31-15:

**Листинг 31-15. Абстрактный пример стиля форматирования явного блока**

```
A   ████████████████
B       ██████████
C       ███████████
D   ████
```

Но для того, чтобы в новом стиле *begin* и *end* трактовались как составные части структуры блока, а не управляющего выражения, надо поместить *begin* в начало блока (а не в конец управляющего выражения), а *end* — в конец блока (а не в качестве терминатора управляющего выражения). Говоря абстрактно, вам нужно сделать нечто, подобное структуре, изображенной в листинге 31-16:

**Листинг 31-16. Абстрактный пример использования *begin* и *end* в качестве границ блока**

```
A   ████████████████
        {
B       ████████████
C       ████████████
        }
```

Несколько примеров использования *begin* и *end* в качестве границ блоков, показано в листингах 31-17, 31-18 и 31-19:

**Листинг 31-17. Пример применения *begin* и *end* в качестве границ блока *if* (C++)**

```
if ( pixelColor == Color_Red )
   {
   statement1;
   statement2;
   ...
   }
```

**Листинг 31-18. Пример применения *begin* и *end* в качестве границ блока *while* (C++)**

```
while ( pixelColor == Color_Red )
   {
   statement1;
   statement2;
   ...
   }
```

**Листинг 31-19. Пример применения *begin* и *end* в качестве границ блока *switch/case* (C++)**

```
switch ( pixelColor )
   {
   case Color_Red:
      statement1;
      statement2;
      ...
      break;
   case Color_Green:
      statement1;
      statement2;
      ...
      break;
   default:
      statement1;
      statement2;
      ...
      break;
   }
```

Такой стиль выравнивания также вполне функционален и согласуется с Основной теоремой форматирования (напомним: он показывает логическую структуру, лежащую в основе данного кода). Единственное его ограничение в том, что его нельзя применять буквально в операторах *switch/case* в C++ и Java, что показано в листинге 31-19. (Ключевое слово *break* служит заменой закрывающей скобке, а у открывающей скобки нет эквивалента.)

## Форматирование в конце строки

Еще одна стратегия форматирования называется «форматированием в конце строки», и объединяет большую группу методик, в которых отступ в коде делается к середине или концу строки. Отступы в конце строки служат для выравнивания блока относительно ключевого слова, с которого он начинается, выравнивания следующих параметров метода под первым параметром, размещения вариантов в операторе *case* и подобных случаев. Листинг 31-20 представляет абстрактный пример:

**Листинг 31-20. Абстрактный пример стиля форматирования в конце строки**

```
A [redacted]
B [redacted]
C [redacted]
D [redacted]
```

В этом примере оператор A начинает управляющую конструкцию, а оператор D завершает. Операторы B, C и D выровнены под ключевым словом, с которого начинается блок в операторе A. Одинаковые отступы B, C и D показывают, что эти операторы сгруппированы вместе. Листинг 31-21 содержит менее абстрактный пример кода, отформатированного в соответствии с этой стратегией:

**Листинг 31-21. Пример форматирования в конце строки в блоке *while* (Visual Basic)**

```
While ( pixelColor = Color_Red )
        statement1;
        statement2;
        ...
        Wend
```

В этом примере *end* помещен в конец строки, а не под соответствующим ключевым словом. Некоторые предпочитают располагать *end* под ключевым словом, но выбор между этими двумя вполне приемлемыми вариантами — наименьшая из проблем этого стиля.

Стиль форматирования в конце строки иногда работает вполне удовлетворительно. Листинг 31-22 демонстрирует пример, в котором этот стиль работает:

**Листинг 31-22. Редкий пример, в котором форматирование в конце строки выглядит привлекательно (Visual Basic)**

```
If ( soldCount > 1000 ) Then
                              markdown = 0.10
                              profit = 0.05
```

Ключевое слово *else* выровнено относительно слова *then*, расположенного над ним.

```
                         Else
                              markdown = 0.05
                         End If
```

В этом случае ключевые слова *Then*, *Else* и *End If* выровнены, и код, следующий за ними, также выровнен. Визуальный эффект соответствует ясной логической структуре.

Критически взглянув на приводимый ранее пример *case*-оператора, вы, вероятно, сможете указать на недостаток данного стиля. По мере усложнения условного выражения этот стиль начнет давать бесполезные или обманчивые подсказки о логической структуре кода. В листинге 31-23 приведен пример ухудшения этого стиля при его применении в более сложном условном выражении:

**Листинг 31-23. Более типичный пример, в котором форматирование в конце строки является неудачным решением (Visual Basic)**

```
If ( soldCount > 10 And prevMonthSales > 10 ) Then
   If ( soldCount > 100 And prevMonthSales > 10 ) Then
      If ( soldCount > 1000 ) Then
                                 markdown = 0.1
                                 profit = 0.05
                            Else
                                 markdown = 0.05
                            End If
                                                Else
                                                   markdown = 0.025
                                                End If
                                           Else
                                              markdown = 0.0
                                           End If
```

В чем причина причудливого форматирования выражений *Else* в конце примера? Они последовательно выровнены под соответствующими ключевыми словами, но вряд ли можно утверждать, что такие отступы проясняют логическую структуру. И если при модификации кода изменится длина первой строки, такой стиль форматирования потребует изменения отступов во всех соответствующих выражениях. Так что возникает проблема сопровождения, которой не существует для стилей явных блоков, их эмуляции и использования пар *begin-end* для обозначения границ блоков.

Вы можете решить, что эти примеры придуманы лишь в демонстрационных целях, но такой стиль применяется очень упорно, несмотря на все его недостатки. Масса учебников и справочников по программированию рекомендует этот стиль. Самая первая увиденная мной книга, содержащая эту рекомендацию, была опубликована в середине 1970-х, последняя — в 2003 году.

Вообще форматирование в конце строки неаккуратно, его сложно применять единообразно и тяжело сопровождать. Далее я приведу другие проблемы такого стиля.

### Какой стиль наилучший?

При работе в Visual Basic используйте отступы для явных блоков (тем более что в среде разработки Visual Basic затруднительно не придерживаться этого стиля).

В Java стандартной практикой является применение формата явных блоков.

В C++ можно просто выбрать тот стиль, который вам больше нравится или которому отдают предпочтение большинство разработчиков вашей команды. Как эмуляция явных блоков, так и обозначение границ с помощью *begin-end* работает одинаково хорошо. Единственное исследование, сравнивавшее эти два стиля, не обнаружило статистически значимых различий между ними с точки зрения понятности кода (Hansen and Yim, 1987).

Ни один из этих стилей не обеспечивает защиты от дурака, и оба время от времени требуют «разумного и очевидного» компромисса. Вы можете предпочесть тот или иной стиль по эстетическим причинам. В этой книге в примерах кода применяется стиль явных блоков, так что вы можете увидеть массу иллюстраций этого стиля, просто просмотрев листинги. Выбрав однажды стиль, вы получите наибольшую выгоду от хорошего форматирования, применяя его единообразно.

## 31.4. Форматирование управляющих структур

**Перекрестная ссылка** О документировании управляющих структур см. подраздел «Комментирование управляющих структур» раздела 32.5. О других аспектах управляющих структур см. главы 14–19.

Форматирование некоторых программных элементов часто является только эстетическим вопросом. Однако форматирование управляющих структур влияет на удобство чтения и понимания и поэтому имеет практическое значение.

### Тонкие моменты форматирования блоков управляющих структур

Работа с блоками управляющих структур требует внимания к деталям. Вот некоторые советы.

***Избегайте отсутствия отступов в парах begin-end*** В стиле форматирования, проиллюстрированном в листинге 31-24, пара *begin-end* выровнена по границе управляющей структуры, а в выражениях, охватываемых операторами *begin* и *end*, сделаны отступы относительно *begin*.

**Листинг 31-24. Пример пары *begin-end*, не выделенной отступами (Java)**

Ключевое слово *begin* выровнено по границе *for*.

```
for ( int i = 0; i < MAX_LINES; i++ )
{
```

В выражениях сделан отступ относительно *begin*.

```
   ReadLine( i );
   ProcessLine( i );
```

Слово *end* также выровнено по границе структуры *for*.

```
}
```

Хотя такой подход выглядит хорошо, он нарушает Основную теорему форматирования, так как не показывает логическую структуру кода. При таком расположении *begin* и *end* не являются частью управляющей структуры, но в то же время, они не являются и частью блока выражений, расположенного далее.

Листинг 31-25 демонстрирует абстрактное представление этого подхода:

**Листинг 31-25. Абстрактный пример вводящего в заблуждение выравнивания**


Можно ли сказать, что оператор B подчиняется оператору A? Он не выглядит частью оператора A, и нет оснований считать, что он ему подчиняется. Если вы используете такой подход, смените его на один из двух вариантов стиля, описанных ранее, и ваше форматирование будет более целостным.

***Избегайте двойных отступов при использовании* begin *и* end** Следствием правила относительно использования пары *begin-end* без отступов является случай, касающийся дополнительных отступов после *begin-end*. Этот стиль, продемонстрированный в листинге 31-26, содержит отступы как перед *begin* и *end*, так и перед выражениями, которые они охватывают:

**Листинг 31-26. Пример неуместного двойного отступа в блоке *begin-end* (Java)**

```
for ( int i = 0; i < MAX_LINES; i++ )
   {
```

В выражениях после *begin* сделан лишний отступ.

```
        ReadLine( i );
        ProcessLine( i );
    }
```

Еще один пример стиля, внешне привлекательного, но нарушающего Основную теорему форматирования. Исследования показали, что с точки зрения понимания программы, использующие одинарные и двойные отступы, не отличаются друг от друга (Miaria et al., 1983), но приведенный стиль неточно отображает логическую структуру программы. *ReadLine()* и *ProcessLine()* показаны так, будто они подчинены паре *begin-end*, что не соответствует действительности.

Этот подход также преувеличивает сложность логической структуры программы. Какая из структур, приведенных в листингах 31-27 и 31-28, выглядит сложнее?

**Листинг 31-27. Абстрактная структура 1**

**Листинг 31-28. Абстрактная структура 2**

Обе являются абстрактными представлениями структуры цикла *for*. Структура 1 выглядит сложней, хотя и представляет тот же код, что и Структура 2. Если бы вам понадобилось создать 2 или 3 уровня вложенности операторов, то из-за двойных отступов вы получили бы 4 или 6 уровней отступов. Такое форматирование выглядело бы сложнее, чем на самом деле. Избавьтесь от этой проблемы, эмулировав явные блоки или применив *begin* и *end* в качестве границ блока, выравнивая *begin* и *end* так же, как и выражения, которые они охватывают.

## Другие соглашения

Хотя отступы в блоках являются главным вопросом форматирования управляющих структур, вы также можете столкнуться с другими проблемами, поэтому далее я привожу еще несколько советов.

***Используйте пустые строки между абзацами*** Некоторые блоки не разграничиваются с помощью пар *begin-end*. Логический блок — группа подходящих друг к другу операторов — должны рассматриваться так же, как абзацы в обычной книге. Отделяйте их друг от друга с помощью пустых строк. Листинг 31-29 представляет пример абзацев, которые следует разделить:

### Листинг 31-29. Пример кода, который следует сгруппировать и разбить на абзацы (C++)

```
cursor.start = startingScanLine;
cursor.end   = endingScanLine;
window.title = editWindow.title;
window.dimensions       = editWindow.dimensions;
window.foregroundColor  = userPreferences.foregroundColor;
cursor.blinkRate        = editMode.blinkRate;
window.backgroundColor  = userPreferences.backgroundColor;
SaveCursor( cursor );
SetCursor( cursor );
```

**Перекрестная ссылка** Если вы придерживаетесь Процесса программирования с псевдокодом, ваш код автоматически разбивается на абзацы (см. главу 9).

Этот код выглядит неплохо, но пустые строки позволят улучшить его с двух точек зрения. Во-первых, если у вас есть группа операторов, не требующих выполнения в определенном порядке, заманчиво объединить их именно так. Вам не надо усовершенствовать порядок выражений для помощи компьютеру, но читатели-люди оценят дополнительные подсказки о том, какие операторы выполнять в определенном порядке, а какие — просто расположены рядом друг с другом. Дисциплина добавления пустых строк в коде программы заставляет вас тщательней обдумывать вопрос, какие операторы на самом деле подходят друг другу. Исправленный фрагмент кода, представленный в листинге 31-30, показывает, как организовать данный набор операторов.

### Листинг 31-30. Пример кода, который правильно сгруппирован и разбит на абзацы (C++)

```
Эти строки настраивают текстовое окно.

window.dimensions = editWindow.dimensions;
window.title = editWindow.title;
window.backgroundColor = userPreferences.backgroundColor;
window.foregroundColor = userPreferences.foregroundColor;

Эти строки устанавливают параметры курсора и должны быть отделены от предыдущих строк.

cursor.start = startingScanLine;
cursor.end = endingScanLine;
cursor.blinkRate = editMode.blinkRate;
SaveCursor( cursor );
SetCursor( cursor );
```

Реорганизованный код подчеркивает выполнение двух разных действий. В первом примере недостатки организации операторов, отсутствие пустых строк, а также старый трюк с выравниванием знаков равенства приводили к тому, что выражения выглядели связанными сильнее, чем на самом деле.

Второе преимущество использования пустых строк, приводящее к улучшению кода, заключается в появлении естественного промежутка для добавления комментариев. В листинге 31-30 комментарии над каждым блоком станут отличным дополнением к улучшенному формату.

***Форматируйте блоки из одного оператора единообразно*** Блоком из одного оператора называется единственный оператор, следующий за управляющей структурой, например, оператор, который следует за проверкой условия *if*. В этом случае для корректной компиляции пара *begin* и *end* необязательна, и вы можете выбрать один из трех вариантов стиля, показанных в листинге 31-31:

**Листинг 31-31. Пример вариантов стиля для блоков из одного оператора (Java)**

```
Стиль 1
if ( expression )
    один-оператор;

Стиль 2а
if ( expression ) {
    один-оператор;
}

Стиль 2б
if ( expression )
    {
    один-оператор;
    }

Стиль 3
if ( expression ) один-оператор;
```

Существуют аргументы в защиту каждого из этих трех подходов. Стиль 1 следует схеме отступов, используемой для блоков, поэтому он согласуется с остальными подходами. Стиль 2 (как 2а, так и 2б) также не противоречит общим правилам, а пары *begin-end* уменьшают вероятность добавления операторов после условия *if*, не указав *begin* и *end*. Это станет особенно трудноуловимой ошибкой, потому что отступы будут подсказывать вам, что все в порядке, однако компилятор не интерпретирует отступы. Основное преимущества Стиля 3 над Стилем 2 в том, что его легче набирать. Его преимущество над стилем 1 в том, что при копировании в другую часть программы он с большей вероятностью будет скопирован правильно. А недостаток в том, что при использовании построчного отладчика такая строка будет рассматриваться как одно целое и вы не узнаете, выполняется ли выражение после проверки условия *if*.

Я использовал Стиль 1 и много раз становился жертвой неправильной модификации. Мне не нравится нарушение стратегии отступов в Стиле 3, поэтому его я тоже избегаю. В групповых проектах я предпочитаю вариации Стиля 2 из-за получаемого единообразия и возможности безопасной модификации. Независимо от избранного стиля применяйте его последовательно и используйте один и тот же стиль в условиях *if* и во всех циклах.

***В сложных выражениях размещайте каждое условие на отдельной строке*** Размещайте каждую часть сложного выражения на отдельной строке. Листинг 31-32 содержит пример выражения, форматированного без учета удобочитаемости:

### Листинг 31-32. Пример совершенно неформатированного (и нечитаемого) сложного выражения (Java)

```
if ((('0' <= inChar) && (inChar <= '9')) || (('a' <= inChar) &&
   (inChar <= 'z')) || (('A' <= inChar) && (inChar <= 'Z')))
   ...
```

Это пример форматирования для машины, а не для человека. Разбив выражение на несколько строк, как показано в листинге 31-33, вы можете обеспечить более удобное чтение.

### Листинг 31-33. Пример вполне читаемого сложного выражения (Java)

```
if ( ( ( '0' <= inChar ) && ( inChar <= '9' ) ) ||
   ( ( 'a' <= inChar ) && ( inChar <= 'z' ) ) ||
   ( ( 'A' <= inChar ) && ( inChar <= 'Z' ) ) )
   ...
```

Второй фрагмент использует несколько методов форматирования (отступы, пробелы, размещение значений в соответствии с числовой прямой и выделение незавершенных строк) — в результате выражение вполне можно прочесть. Более того, назначение проверки понятно. Если выражение содержит небольшую ошибку, скажем, указана *z* вместо *Z*, то при таком форматировании в отличие от менее аккуратного варианта это сразу бросится в глаза.

**Перекрестная ссылка** Другая методика повышения удобочитаемости сложных выражений — перенести их в логические функции (см. раздел 19.1).

***Избегайте операторов* goto** Первоначальная причина отказа от *goto* состояла в том, что они затрудняли доказательство корректности программы. Это хороший аргумент для тех, кто хочет доказать, что их программы корректны, т. е. практически ни для кого. Для большинства программистов более насущная проблема состоит в том, что операторы *goto* затрудняют форматирование кода. Делаете ли вы отступ в коде между *goto* и меткой, на которую он переходит? Что, если у вас есть несколько *goto*, использующих одну и ту же метку? Размещаете ли вы их один под другим? Вот несколько советов по форматированию *goto*.

**Перекрестная ссылка** Об операторе *goto* см. раздел 17.3.

- Избегайте *goto*. Это также позволит обойти проблемы форматирования.
- Для меток перехода используйте только заглавные буквы. Это делает метки очевидными.
- Помещайте оператор, содержащий *goto*, на отдельной строке. Это делает использование *goto* очевидным.
- Помещайте метку перехода *goto* на отдельной строке. Окружите ее пустыми строками. Это подчеркнет наличие метки. Строку, содержащую метку, выравнивайте по левому краю, чтобы сделать метку настолько очевидной, насколько это возможно.

> Метки *goto* должны быть выровнены влево, состоять из заглавных букв и содержать имя программиста, его домашний телефон и номер кредитной карты.
>
> *Абдул Низар (Abdul Nizar)*

Листинг 31-34 показывает эти соглашения по форматированию *goto* в действии.

**Перекрестная ссылка** О других способах решения этой проблемы см. подраздел «Обработка ошибок и операторы *goto*» раздела 17.3.

**Листинг 31-34. Пример наилучшего выхода из плохой ситуации (использование *goto*) C++**

```
void PurgeFiles( ErrorCode & errorCode ) {
   FileList fileList;
   int numFilesToPurge = 0;
   MakePurgeFileList( fileList, numFilesToPurge );

   errorCode = FileError_Success;
   int fileIndex = 0;
   while ( fileIndex < numFilesToPurge ) {
      DataFile fileToPurge;
      if ( !FindFile( fileList[ fileIndex ], fileToPurge ) ) {
         errorCode = FileError_NotFound;
```

Здесь используется *goto*.

```
         goto END_PROC;
      }

      if ( !OpenFile( fileToPurge ) ) {
         errorCode = FileError_NotOpen;
```

Здесь используется *goto*.

```
         goto END_PROC;
      }

      if ( !OverwriteFile( fileToPurge ) ) {
         errorCode = FileError_CantOverwrite;
```

Здесь используется *goto*.

```
         goto END_PROC;
      }

      if ( !Erase( fileToPurge ) ) {
         errorCode = FileError_CantErase;
```

Здесь используется *goto*.

```
         goto END_PROC;
      }
      fileIndex++;
   }
```

Здесь находится метка *goto*. Заглавные буквы и специальное форматирование служат для того, чтобы метку было сложно не заметить.

```
END_PROC:

   DeletePurgeFileList( fileList, numFilesToPurge );
}
```

Пример кода на C++, приведенный в листинге 31-34 достаточно длинный, и вы можете представить себе обстоятельства, когда опытный программист может добросовестно решить, что применение *goto* будет наилучшим решением. В таком случае приведенный пример форматирования — лучшее, что вы можете сделать.

***Не используйте форматирование в конце строки в виде исключения для операторов* case** Одна из опасностей форматирования в конце строки проявляется при работе с операторами *case*. Популярный стиль форматирования *case* состоит в их выравнивании справа от описания каждого варианта (см. листинг 31-35). Проблема в том, что такой стиль приносит при сопровождении лишь головную боль.

**Перекрестная ссылка** Об операторах *case* см. раздел 15.2.

**Листинг 31-35. Пример сложного в сопровождении форматирования в конце строки в операторе *case* (C++)**

```
switch ( ballColor ) {
   case BallColor_Blue:              Rollout();
                                     break;
   case BallColor_Orange:            SpinOnFinger();
                                     break;
    case BallColor_FluorescentGreen: Spike();
                                     break;
   case BallColor_White:             KnockCoverOff();
                                     break;
   case BallColor_WhiteAndBlue:      if ( mainColor == BallColor_White ) {
                                        KnockCoverOff();
                                     }
                                     else if ( mainColor == BallColor_Blue ) {
                                        RollOut();
                                     }
                                     break;
   default:                          FatalError( "Unrecognized kind of ball." );
                                     break;
}
```

Если вы добавите вариант с более длинным, чем существующие, именем, вам придется сдвигать все варианты и код, к ним относящийся. Изначально большие отступы затрудняют размещение какой-то дополнительной логики, как показано в варианте *WhiteAndBlue*. Решением этой проблемы будет переход к стандартному виду отступов. Так, если в циклах вы делаете отступ величиной в три пробела, используйте в вариантах *case* такое же число пробелов, как показано в листинге 31-36:

**Листинг 31-36. Пример хороших стандартных отступов в операторе *case* (C++)**

```
switch ( ballColor ) {
   case BallColor_Blue:
      Rollout();
      break;
   case BallColor_Orange:
      SpinOnFinger();
      break;
```

```
    case BallColor_FluorescentGreen:
       Spike();
       break;
    case BallColor_White:
       KnockCoverOff();
       break;
    case BallColor_WhiteAndBlue:
       if ( mainColor = BallColor_White ) {
          KnockCoverOff();
       }
       else if ( mainColor = BallColor_Blue ) {
          RollOut();
       }
       break;
    default:
       FatalError( "Unrecognized kind of ball." );
       break;
}
```

Это пример ситуации, в которой многие могут предпочесть внешний вид первого варианта. Однако с точки зрения размещения длинных строк, единообразия и удобства сопровождения второй подход имеет гораздо больше преимуществ.

Если у вас есть оператор *case*, все варианты которого выглядят абсолютно одинаково, а все действия называются кратко, можно рассмотреть возможность размещения и варианта, и действия на одной и той же строке. Однако в большинстве случаев вы впоследствии пожалеете об этом. Это форматирование изначально неудобно и портится при модификации. Кроме того, тяжело поддерживать одинаковую структуру всех вариантов в случае удлинения кратких имен действий.

# 31.5. Форматирование отдельных операторов

Ниже описаны способы улучшения отдельных выражений в программе.

## Длина выражения

**Перекрестная ссылка** О документировании отдельных выражений см. подраздел «Комментирование отдельных строк» раздела 32.5.

Общее и в какой-то мере устаревшее правило гласит, что длина строки выражения не должна превышать 80 символов, так как строки длиннее 80 символов:

- тяжело читать;
- препятствуют созданию глубокой вложенности;
- часто не помещаются на стандартный лист бумаги, особенно если печатается по 2 страницы кода на каждой физической странице распечатки.

С появлением больших экранов, узких шрифтов и альбомной ориентации бумаги 80-символьное ограничение выглядит необоснованным. Одна строка в 90 символов часто удобнее для чтения, чем другая, разбитая на две только с целью избежать выхода за 80-символьную границу. При современном уровне технологий,

вероятно, будет правильным изредка допускать превышение ограничения в 80 колонок.

## Использование пробелов для ясности

Добавляйте в выражение пробелы в целях повышения удобства чтения:

***Используйте пробелы, чтобы сделать читаемыми логические выражения***
Выражение:

```
while(pathName[startPath+position]<>';')  and
    ((startPath+position)<length(pathName)) do
```

примерно так же читабельно, как Попробуйтепрочитайтеэто.

Как правило, следует отделять идентификаторы друг от друга с помощью пробелов. Следуя этой рекомендации, выражение *while* можно переписать так:

```
while ( pathName[ startPath+position ] <> ';' ) and
   (( startPath + position ) < length( pathName )) do
```

Некоторые художники от программирования могут предложить дальнейшее улучшение данного выражения с помощью дополнительных пробелов для подчеркивания его логической структуры:

```
while ( pathName[ startPath + position ] <> ';' ) and
   ( ( startPath + position ) < length( pathName ) ) do
```

Это хорошо, но и предыдущий вариант неплохо улучшал читабельность. Однако дополнительные пробелы вряд ли повредят, так что будьте на них щедрее.

***Используйте пробелы, чтобы сделать читаемыми обращения к массиву***
Выражение:

```
grossRate[census[groupId].gender,census[groupId].ageGroup]
```

не легче читать, чем приводимое ранее сжатое выражение *while*. Добавляйте пробелы вокруг каждого индекса массива для упрощения их чтения. После применения этого правила выражение будет выглядеть так:

```
grossRate[ census[ groupId ].gender, census[ groupId ].ageGroup ]
```

***Используйте пробелы, чтобы сделать читаемыми аргументы методов***
Назовите четвертый аргумент следующего метода:

```
ReadEmployeeData(maxEmps,empData,inputFile,empCount,inputError);
```

А теперь назовите четвертый аргумент этого метода:

```
GetCensus( inputFile, empCount, empData, maxEmps, inputError );
```

Какой из них было легче найти? Вопрос вполне закономерный, поскольку порядок аргументов имеет большое значение во всех основных процедурных языках. Довольно часто на одной половине экрана приходится располагать определение метода, на другой — его вызов и сравнивать каждый формальный параметр с соответствующим фактическим параметром.

## Форматирование строк с продолжением

Одна из наиболее неприятных проблем форматирования — решение о том, что делать с частью выражения, переносимой на следующую строку. Делать ли нормальный отступ? Или выровнять его по ключевому слову? А что делать с присваиваниями?

Далее приведен здравый, последовательный подход, особенно полезный в Java, C, C++, Visual Basic и других языках, поощряющих длинные имена переменных.

***Сделайте так, чтобы незавершенность выражения была очевидна*** Иногда выражение должно разбиваться на несколько строк либо потому, что оно длинней, чем это позволяют стандарты программирования, либо потому, что оно слишком длинное, чтобы поместиться в одной строке. Сделайте очевидным факт, что часть выражения на первой строке является всего лишь частью. Самый простой способ добиться этого — разбить выражение так, чтобы его часть на первой строке стала вопиюще некорректной, если рассматривать ее отдельно. В листинге 31-37 приведены некоторые примеры:

**Листинг 31-37. Примеры очевидно незавершенных выражений (Java)**

Оператор *&&* сигнализирует, что выражение не завершено.

```
while ( pathName[ startPath + position ] != ';' ) &&
    ( ( startPath + position ) <= pathName.length() )
...
```

Знак плюс сигнализирует, что выражение не завершено.

```
totalBill = totalBill + customerPurchases[ customerID ] +
    SalesTax( customerPurchases[ customerID ] );
...
```

Запятая сигнализирует, что выражение не завершено.

```
DrawLine( window.north, window.south, window.east, window.west,
    currentWidth, currentAttribute );
...
```

Кроме сообщения о незавершенности выражения в первой строке, такое разбиение предотвращает неправильную модификацию. Если удалить продолжение выражения, первая строка не будет выглядеть так, будто вы просто забыли скобку или точку с запятой — оно явно требует чего-то еще.

Альтернативный подход, также хорошо работающий, — размещение знака продолжения в начале перенесенной строки, как показано в листинге 31-38.

**Листинг 31-38. Примеры очевидно незавершенных выражений — альтернативный стиль (Java)**

```
while ( pathName[ startPath + position ] != ';' )
   && ( ( startPath + position ) <= pathName.length() )
...
```

```
totalBill = totalBill + customerPurchases[ customerID ]
   + SalesTax( customerPurchases[ customerID ] );
```

Хотя такой стиль не приведет к появлению синтаксической ошибки при повисании операторов && или +, он сильно упрощает поиск операторов по левой границе столбца — там, где текст выровнен, а не по правой — где край текста неровный. Его дополнительное преимущество в том, что он позволяет пояснить структуру операций, как показано в листинге 31-39.

**Листинг 31-39. Пример стиля, поясняющего смысл сложной операции (Java)**

```
totalBill = totalBill
   + customerPurchases[ customerID ]
   + CitySalesTax( customerPurchases[ customerID ] )
   + StateSalesTax( customerPurchases[ customerID ] )
   + FootballStadiumTax()
   - SalesTaxExemption( customerPurchases[ customerID ] );
```

***Располагайте сильно связанные элементы вместе*** Разбивая строку на части, оставляйте рядом взаимосвязанные элементы: обращения к массиву, аргументы метода и т. д. Пример, приведенный в листинге 31-40, демонстрирует плохой вариант:

**Листинг 31-40. Пример неправильного разбиения строки (Java)**

```
customerBill = PreviousBalance( paymentHistory[ customerID ] ) + LateCharge(
   paymentHistory[ customerID ] );
```

Надо признать, что такой разрыв строки соответствует принципу подчеркивания очевидности незавершенных выражений, но это сделано так, что надобности затрудняет чтение выражения. Можно придумать ситуации, когда такое разбиение будет оправданным, но это не тот случай. Лучше оставить все обращения к массиву на одной строке. Листинг 31-41 показывает лучшее форматирование:

**Листинг 31-41. Пример правильного разбиения строки (Java)**

```
customerBill = PreviousBalance( paymentHistory[ customerID ] ) +
   LateCharge( paymentHistory[ customerID ] );
```

***При переносе строк в вызове метода используйте отступ стандартного размера*** Если в циклах и условных выражениях вы обычно делаете отступ в три пробела, то при переносе строки в вызове метода тоже сделайте отступ в три пробела. Листинг 31-42 содержит несколько примеров:

**Листинг 31-42. Примеры переноса строк в вызовах методов с применением стандартного размера отступов (Java)**

```
DrawLine( window.north, window.south, window.east, window.west,
   currentWidth, currentAttribute );
SetFontAttributes( faceName[ fontId ], size[ fontId ], bold[ fontId ],
   italic[ fontId ], syntheticAttribute[ fontId ].underline,
   syntheticAttribute[ fontId ].strikeout );
```

Одной из альтернатив такому подходу служит выравнивание перенесенных строк под первым аргументом, как показано в листинге 31-43:

**Листинг 31-43. Пример отступов при переносе строк в вызове методов, позволяющих выделить имена методов (Java)**

```
DrawLine( window.north, window.south, window.east, window.west,
          currentWidth, currentAttribute );
SetFontAttributes( faceName[ fontId ], size[ fontId ], bold[ fontId ],
                   italic[ fontId ], syntheticAttribute[ fontId ].underline,
                   syntheticAttribute[ fontId ].strikeout );
```

С эстетической точки зрения, в сравнении с первым вариантом этот код выглядит несколько неровно. Кроме того, этот подход тяжело сопровождать, так как имена методов и аргументов могут изменяться. Большинство программистов со временем склоняются к использованию первого стиля.

***Упростите поиск конца строки с продолжением*** Одна из проблем показанного выше подхода — сложность поиска конца каждой строки. Альтернативой может служить размещение каждого элемента на отдельной строке и выделение окончания всей группы с помощью закрывающей скобки (листинг 31-44).

**Листинг 31-44. Пример форматирования переноса строк в вызовах методов, в котором каждый аргумент размещается на отдельной строке (Java)**

```
DrawLine(
   window.north,
   window.south,
   window.east,
   window.west,
   currentWidth,
   currentAttribute
);

SetFontAttributes(
   faceName[ fontId ],
   size[ fontId ],
   bold[ fontId ],
   italic[ fontId ],
   syntheticAttribute[ fontId ].underline,
   syntheticAttribute[ fontId ].strikeout
);
```

Ясно, что такой подход требует много места. Однако если аргументами функции являются длинные названия полей объектов или имена указателей, например, такие, как два последних в приведенном примере, размещение одного аргумента в строке существенно улучшает читаемость. Знаки *);* в конце блока делают заметным его окончание. Вам также не придется переформатировать код при добавлении параметра — вы просто добавите новую строку.

На практике только небольшая часть методов нуждается в разбиении на строки. Остальные можно располагать на одной строке. Любой из трех вариантов фор-

матирования многострочных вызовов методов работает хорошо при последовательном применении.

***При переносе строк в управляющем выражении делайте отступ стандартного размера*** При нехватке места для цикла *for*, цикла *while* или оператора *if*, перенося строку, сделайте такой же отступ, как и в выражениях внутри цикла или после условия *if*. В листинге 31-45 приведены два примера:

**Листинг 31-45. Примеры отступов при переносе строк в управляющих выражениях (Java)**

```
while ( ( pathName[ startPath + position ] != ';' ) &&
В продолжении строки отступ содержит стандартное количество пробелов...
    ( ( startPath + position ) <= pathName.length() ) ) {
    ...
}

for ( int employeeNum = employee.first + employee.offset;
...так же, как и в этом случае.
    employeeNum < employee.first + employee.offset + employee.total;
    employeeNum++ ) {
    ...
}
```

Приведенный код соответствует критериям, установленным ранее в этой главе. Перенос строки выполняется логично — в нем всегда есть отступ относительно выражения, которое он продолжает. Дальнейшее выравнивание может выполняться как обычно — новая строка занимает всего на несколько пробелов больше, чем исходный вариант. Он так же читаем, как и любой другой код, и его так же просто сопровождать. В некоторых случаях вы могли бы повысить удобочитаемость, настроив отступы или пробелы, но, рассматривая вопросы тонкой настройки, не забывайте о компромиссе с точки зрения удобства сопровождения.

**Перекрестная ссылка** Иногда для сложных условий лучше всего поместить их в логические функции (см. подраздел «Упрощение сложных выражений» раздела 19.1).

***Не выравнивайте правые части выражений присваивания*** В первом издании этой книги я рекомендовал выравнивать правые части выражений, содержащие присваивания (листинг 31-46):

**Листинг 31-46. Примеры форматирования в конце строки, используемого при выравнивании выражений присваивания, — плохая практика (Java)**

```
customerPurchases = customerPurchases + CustomerSales( CustomerID );
customerBill      = customerBill + customerPurchases;
totalCustomerBill = customerBill + PreviousBalance( customerID ) +
                    LateCharge( customerID );
customerRating    = Rating( customerID, totalCustomerBill );
```

С высоты 10-летнего опыта могу сказать, что, хотя этот стиль может выглядеть привлекательно, он превращается в настоящую головную боль, когда приходится поддерживать выравнивание знаков равенства при изменении имен переменных и прогоне кода через утилиты, заменяющие пробелы знаками табуляции, а знаки табуляции — пробелами. Кроме того, его тяжело сопровождать при перемещении строк между частями программы, в которых применяются разные размеры отступов.

Для обеспечения единообразия с другими принципами отступов, а также учитывая вопросы удобства сопровождения, обращайтесь с группами выражений, содержащими операции присваивания, так же, как вы бы обращались с любыми другими выражениями (листинг 31-47):

**Листинг 31-47. Пример использования стандартных отступов для выравнивания выражений присваивания — хорошая практика (Java)**

```
customerPurchases = customerPurchases + CustomerSales( CustomerID );
customerBill = customerBill + customerPurchases;
totalCustomerBill = customerBill + PreviousBalance( customerID ) +
   LateCharge( customerID );
customerRating = Rating( customerID, totalCustomerBill );
```

***При переносе строк в выражениях присваивания применяйте отступы стандартного размера*** В листинге 31-47 при переносе строки в третьем присваивании используется стандартный размер отступа. Это сделано с той же целью, что и отказ от специального форматирования выражений присваивания: из общих соображений читаемости и удобства сопровождения.

## Размещение одного оператора на строке

Современные языки, такие как C++ и Java, позволяют располагать несколько операторов на одной строке. Однако когда дело касается этого вопроса, мощь свободного форматирования оборачивается палкой о двух концах. Следующая строка содержит несколько выражений, которые, с точки зрения логики, вполне могут располагаться в отдельных строках:

```
i = 0; j = 0; k = 0; DestroyBadLoopNames( i, j, k );
```

Аргументом в защиту размещения нескольких выражений на одной строке может служить факт, что в этом случае требуется меньшее число строк экранного пространства или бумаги для распечатки, что позволяет одновременно видеть больший объем кода. Это также позволяет сгруппировать взаимосвязанные выражения, а некоторые программисты даже полагают, что так они подсказывают компилятору, как можно оптимизировать код.

Все так, но основания для самоограничения, требующие оставлять не более одного оператора в строке, гораздо серьезней.

- Размещение каждого оператора на отдельной строке дает точное представление о сложности программы. При этом не скрывается сложность из-за того, что сложные операторы выглядят тривиальными. Сложные операторы и выглядят сложными, простые — простыми.

- Размещение нескольких операторов на одной строке не помогает современным компиляторам в оптимизации. Сегодняшние оптимизирующие компиляторы не нуждаются в подсказках, сделанных с помощью форматирования (см. ниже).

**Перекрестная ссылка** Об оптимизации производительности на уровне кода см. главы 25 и 26.

- Если операторы расположены на отдельных строках, чтение кода происходит сверху вниз, а не сверху вниз и слева направо. При поиске определенной строки у взгляда должна быть возможность придерживаться левого края кода. Он не должен просматривать каждую строку целиком только потому, что в одной строке может быть два оператора.
- При размещении операторов на отдельных строках легко найти синтаксические ошибки, если компилятор сообщает только номера строк, где они произошли. При расположении нескольких операторов на одной строке ее номер ничего не скажет о том, какой оператор содержит ошибку.
- При размещении операторов на отдельных строках легко выполнять пошаговую отладку кода, используя построчные отладчики. Если строка содержит несколько операторов, отладчик выполнит их все одновременно, и вам придется переключиться на ассемблерный листинг для выполнения пошаговой отладки отдельных выражений.
- Когда строка содержит только один оператор, его легко редактировать — можно удалить или временно закомментировать всю строку. Если же на одной строке вы разместили несколько операторов, вам придется выполнять редактирование между остальными операторами.

***В C++ избегайте выполнения нескольких операций в одной строке (побочные эффекты)*** Побочные эффекты — это последствия выполнения некоторого выражения, проявляющиеся в дополнение к основным результатам выполнения этого выражения. Так, в C++ оператор ++, расположенный на одной строке с другими операторами, приводит к проявлению побочного эффекта. Присваивание значения переменной и применение левой части этого присваивания в условном операторе также является примером побочного эффекта.

Побочные эффекты снижают читаемость кода. Например, если *n* равно *4*, что напечатает выражение, приведенное в листинге 31-48?

**Листинг 31-48. Пример непредсказуемого побочного эффекта (C++)**

```
PrintMessage( ++n, n + 2 );
```

*4* и *6*? Или *5* и *7*? А может, *5* и *6*? Правильный ответ: «Ни то, ни другое и не третье». Первый аргумент —*++n* — равен *5*. Но язык C++ не определяет порядок вычисления условий выражения или аргументов функции. Поэтому компилятор может вычислить второй аргумент, *n + 2*, либо до, либо после первого аргумента, и результат может быть равен *6* или *7* в зависимости от компилятора. В листинге 31-49 показано, как переписать это выражение, чтобы прояснить свои намерения:

**Листинг 31-49. Пример избавления от непредсказуемого побочного эффекта (C++)**

```
++n;
PrintMessage( n, n + 2 );
```

Если вы все еще не совсем уверены в том, что побочные эффекты надо выносить в отдельные строки, попробуйте понять, что делает функция, приводимая в листинге 31-50:

**Листинг 31-50. Пример слишком большого количества операции в строке (C)**

```
strcpy( char * t, char * s ) {
   while ( *++t = *++s )
      ;
}
```

Некоторые опытные программисты не видят сложности в этом примере, потому что эта функция им знакома. Они смотрят на нее и говорят: «Это функция *strcpy()*». Однако в нашем случае это не совсем *strcpy()*. Она содержит ошибку. Если вы сказали «Это *strcpy()*», увидев данный код, вы узнали код, а не прочитали его. В такую же ситуацию вы попадаете при отладке программы: код, на который вы не обратили внимания, потому что «узнали», а не прочли его, может содержать ошибку, поиск которой займет гораздо больше времени, чем она этого заслуживает.

Фрагмент, показанный в листинге 31-51, функционально идентичен первому варианту и гораздо удобней для чтения:

**Листинг 31-51. Пример читаемого количества операций в каждой строке (C)**

```
strcpy( char * t, char * s ) {
   do {
      ++t;
      ++s;
      *t = *s;
    }
   while ( *t != '\0' );
}
```

В этом переформатированном коде ошибка очевидна. Конечно, *t* и *s* инкрементируются до того, как **s* будет скопирована в **t*. Первый символ пропускается.

Второй пример выглядит продуманней первого, хотя операции, выполняемые во втором примере, идентичны первому. Причина такого впечатления в том, что во втором варианте не скрывается сложность выполняемых действий.

**Перекрестная ссылка** О настройке кода см. главы 25 и 26.

Рост производительности также не оправдывает размещения нескольких операций на одной строке. Поскольку обе функции *strcpy()* логически эквивалентны, можно ожидать, что компилятор сгенерирует для них идентичный код. Однако при профилировании обеих функций выяснилось, что для копирования 5 000 000 строк первой функции понадобилось 4,81 секунды, а второй —4,35.

В нашем случае «умная» версия показала снижение скорости на 11%, что делает ее гораздо менее умной. Результаты могут изменяться от компилятора к компиля-

тору, но в целом они свидетельствуют о том, что пока вы не измерили прирост производительности, следует сначала стремиться к ясности и корректности, а уж затем — к производительности.

Даже если вы легко читаете выражения с побочными эффектами, пожалейте тех, кому придется разбираться с вашим кодом. Большинству программистов нужно дважды подумать, чтобы понять выражения с побочными эффектами. Позвольте им использовать мозговые клетки для осмысления более общих вопросов работы вашего кода, а не синтаксических деталей конкретного выражения.

## Размещение объявлений данных

***Располагайте каждое объявление данных в отдельной строке*** Как показали предыдущие примеры, каждому объявлению данных надо выделять отдельную строку. Тогда проще дописывать комментарии к каждому объявлению — ведь каждое расположено в отдельной строке. Проще исправлять объявления, поскольку все они изолированы друг от друга. Проще находить конкретные переменные, так как можно просканировать одну колонку, а не читать каждую строчку. Проще искать и исправлять синтаксические ошибки — строка, указанная компилятором, содержит только одно объявление.

**Перекрестная ссылка** О документировании объявлений данных см. подраздел «Комментирование объявлений данных» раздела 32.5. Об использовании данных см. в главах 10-13.

А ну-ка, скажите быстренько: какой тип имеет переменная *currentBottom* в объявлении данных, приведенном в листинге 31-52?

**Листинг 31-52. Пример скопления нескольких объявлений переменных в одной строке (C++)**

```
int rowIndex, columnIdx; Color previousColor, currentColor, nextColor; Point
previousTop, previousBottom, currentTop, currentBottom, nextTop, nextBottom; Font
previousTypeface, currentTypeface, nextTypeface; Color choices[ NUM_COLORS ];
```

Это, конечно, крайний случай, однако он не так уж далек от гораздо более распространенного стиля, показанного в листинге 31-53:

**Листинг 31-53. Пример скопления нескольких объявлений переменных в одной строке (C++)**

```
int rowIndex, columnIdx;
Color previousColor, currentColor, nextColor;
Point previousTop, previousBottom, currentTop, currentBottom, nextTop, nextBottom;
Font previousTypeface, currentTypeface, nextTypeface;
Color choices[ NUM_COLORS ];
```

Это не такой уж и редкий стиль объявления переменных, а конкретную переменную все так же тяжело найти, поскольку все объявления свалены в кучу. Тип переменной тоже тяжело выяснить. А теперь скажите, какой тип имеет *nextColor* в листинге 31-54?

**Листинг 31-54. Пример читаемого кода, достигнутого благодаря размещению только одной переменной в каждой строке (C++)**

```
int rowIndex;
int columnIdx;
Color previousColor;
Color currentColor;
Color nextColor;
Point previousTop;
Point previousBottom;
Point currentTop;
Point currentBottom;
Point nextTop;
Point nextBottom;
Font previousTypeface;
Font currentTypeface;
Font nextTypeface;
Color choices[ NUM_COLORS ];
```

Вероятно, переменную *nextColor* было проще найти, чем *nextTypeface* в листинге 31-53. Такой стиль характеризуется наличием в каждой строке одного полного объявления, включающего тип переменной.

Надо признать, что такой стиль съедает больше экранного пространства — 20 строк, а не 3, как в первом примере, хотя те три строки и выглядели довольно безобразно. Я не могу процитировать ни одного исследования, показывающего, что этот стиль приводит к меньшему количеству ошибок или к лучшему пониманию программы. Однако если Салли попросит меня посмотреть ее код, а объявления данных будут выглядеть, как в первом примере, я отвечу: «Ни за что — это читать невозможно». Если они будут выглядеть, как во втором примере, я скажу: «Гм... Может, попозже». А если они будут выглядеть, как в третьем примере, я скажу: «Конечно, с удовольствием!»

***Объявляйте переменные рядом с местом их первого использования*** Еще более предпочтительный вариант, чем объявление всех переменных в одном большом блоке, — это стиль, при котором переменная объявляется рядом с местом ее первого использования. Это уменьшает срок службы и время жизни переменной и позволяет проще выполнить рефакторинг кода на меньшие методы (см. подраздел «Делайте время жизни переменных как можно короче» раздела 10.4).

***Разумно упорядочивайте объявления*** В листинге 31-54 объявления сгруппированы по типам. Такая группировка обычно имеет смысл, поскольку переменные одинаковых типов часто используются в аналогичных операциях. В других случаях можно упорядочивать их по алфавиту в соответствии с именами переменных. Хотя у алфавитной сортировки много сторонников, мне кажется, затрачиваемых не нее усилий она не стоит. Если ваш список переменных настолько длинен, что ему помогает алфавитное упорядочение, ваш метод, вероятно, слишком велик. Разбейте его на части, чтобы создать меньшие по размеру методы с небольшим количеством переменных.

***В C++ при объявлении указателей располагайте звездочку рядом с именем переменной или объявляйте типы-указатели*** Очень часто приходится ви-

деть объявления указателей, в которых звездочка расположена рядом с типом, как показано в листинге 31-55:

**Листинг 31-55. Примеры расположения звездочек в объявлениях указателей (C++)**

```
EmployeeList* employees;
File* inputFile;
```

При размещении звездочки рядом с именем типа, а не переменной возникает опасность, что в случае добавления нового объявления в той же строке звездочка будет относиться только к первой переменной, хотя визуальное форматирование наводит на мысль, что она применяется ко всем переменным в строке. Вы можете решить эту проблему, поместив звездочку рядом с именем переменной, а не с именем типа (листинг 31-56):

**Листинг 31-56. Пример расположения звездочек в объявлениях указателей (C++)**

```
EmployeeList *employees;
File *inputFile;
```

Недостаток такого подхода в том, что в этом случае звездочка может казаться частью имени переменной, а это не соответствует действительности. Переменную можно применять как со звездочкой, так и без нее.

Наилучший вариант — объявление и использование типа-указателя (листинг 31-57):

**Листинг 31-57. Пример правильного применения типа-указателя в объявлениях (C++)**

```
EmployeeListPointer employees;
FilePointer inputFile;
```

Указанную проблему можно решить, либо требуя объявления всех указателей с помощью типов-указателей (листинг 31-57), либо запрещая размещение более одной переменной в строке. Убедитесь, что применяете хотя бы одно из этих решений!

## 31.6. Размещение комментариев

Хорошо оформленные комментарии могут значительно улучшить читаемость программы; плохо — сильно ей повредить.

**Перекрестная ссылка** О других аспектах комментариев см. главу 32.

***Делайте в комментарии такой же отступ, как и в соответствующем ему коде*** Визуальные отступы вносят важный вклад в понимание логической структуры программы, и хорошие комментарии не мешают визуальному выравниванию. Например, что можно сказать о логической структуре метода, приведенного в листинге 31-58?

**Листинг 31-58. Пример комментариев с неправильными отступами (Visual Basic)**

```
For transactionId = 1 To totalTransactions
' получаем информацию о транзакции
```

```
    GetTransactionType( transactionType )
    GetTransactionAmount( transactionAmount )

' обрабатываем транзакцию в зависимости от ее типа
    If transactionType = Transaction_Sale Then
        AcceptCustomerSale( transactionAmount )

    Else
        If transactionType = Transaction_CustomerReturn Then

' процесс либо оформляет автоматический возврат, либо в случае необходимости
' ждет подтверждения от менеджера
            If transactionAmount >= MANAGER_APPROVAL_LEVEL Then

' пытаемся получить подтверждение от менеджера, а затем оформляем или отменяем
' возврат средств в зависимости от полученного подтверждения
                GetMgrApproval( isTransactionApproved )
                If ( isTransactionApproved ) Then
                    AcceptCustomerReturn( transactionAmount )
                Else
                    RejectCustomerReturn( transactionAmount )
                End If
            Else

' подтверждение менеджера не требуется, поэтому оформляем возврат
                AcceptCustomerReturn( transactionAmount )
            End If
        End If
    End If
Next
```

В этом примере вам не удастся сходу разобраться в логической структуре кода, так как комментарии полностью скрывают его визуальный формат. Вероятно, тяжело поверить, что кто-то в здравом уме решит использовать такой стиль отступов, но я неоднократно встречал его в профессиональных программах и знаю минимум один учебник, рекомендующий его.

Код, приведенный в листинге 31-59, абсолютно идентичен предыдущему примеру из листинга 31-58 за исключением отступов в комментариях.

**Листинг 31-59. Пример комментариев с правильными отступами (Visual Basic)**

```
For transactionId = 1 To totalTransactions
    ' получаем информацию о транзакции
    GetTransactionType( transactionType )
    GetTransactionAmount( transactionAmount )

    ' обрабатываем транзакцию в зависимости от ее типа
    If transactionType = Transaction_Sale Then
        AcceptCustomerSale( transactionAmount )
```

```
   Else
      If transactionType = Transaction_CustomerReturn Then

         ' процесс либо оформляет автоматический возврат, либо
         ' в случае необходимости ждет подтверждения от менеджера
         If transactionAmount >= MANAGER_APPROVAL_LEVEL Then

            ' пытаемся получить подтверждение от менеджера, а затем оформляем
            ' или отменяем возврат средств в зависимости
            ' от полученного подтверждения
            GetMgrApproval( isTransactionApproved )
            If ( isTransactionApproved ) Then
               AcceptCustomerReturn( transactionAmount )
            Else
               RejectCustomerReturn( transactionAmount )
            End If
         Else
            ' подтверждение менеджера не требуется, поэтому оформляем возврат
            AcceptCustomerReturn( transactionAmount )
         End If
      End If
   End If
Next
```

Логическая структура кода в листинге 31-59 более прозрачна. В ходе одного исследования эффективности комментариев выяснилось, что их применение не всегда является абсолютным достоинством. Автор объяснил это тем, что они «нарушают процесс визуального просмотра программы» (Shneiderman, 1980). Из этих примеров становится понятно, что *стиль* комментариев очень сильно влияет на их «разрушающую» способность.

***Отделяйте каждый комментарий хотя бы одной пустой строкой*** Наиболее эффективный способ беглого просмотра программы — прочитать комментарии, не читая при этом код. Выделение комментариев с помощью пустых строк помогает читателю просматривать код. В листинге 31-60 приведен пример:

**Листинг 31-60. Пример выделения комментария с помощью пустой строки (Java)**

```
// комментарий 0
CodeStatementZero;
CodeStatementOne;

// комментарий 1
CodeStatementTwo;
CodeStatementThree;
```

Некоторые добавляют пустую строку и до, и после комментария. Две пустых строки занимают больше экранного пространства, но кто-то считает, что так код выглядит лучше, чем с одной строкой. Пример таких комментариев приведен в листинге 31-61:

**Листинг 31-61. Пример выделения комментария с помощью двух пустых строк (Java)**

```
// комментарий 0

CodeStatementZero;
CodeStatementOne;

// комментарий 1

CodeStatementTwo;
CodeStatementThree;
```

За исключением случаев, когда экранное пространство ограничено, это чисто эстетический вопрос, и вы можете решать его по своему усмотрению. В этой, как и во многих других ситуациях, сам факт наличия соглашения гораздо важней конкретных деталей этого соглашения.

## 31.7. Размещение методов

**Перекрестная ссылка** О документировании методов см. подраздел «Комментирование методов» раздела 32.5. О процессе создания метода см. раздел 9.3. О разнице между хорошими и плохими методами см. главу 7.

Методы состоят из отдельных операторов, данных, управляющих структур, комментариев — всех тех элементов, которые обсуждались в остальных частях этой главы. В данном разделе рассмотрены принципы форматирования, относящиеся исключительно к методам.

***Используйте пустые строки для разделения составных частей метода*** Оставьте пустые строки между заголовком метода, объявлениями данных и именованных констант (если они есть) и телом метода.

***Используйте стандартный отступ для аргументов метода*** Для форматирования заголовка метода можно задействовать те же, что и раньше, варианты форматирования: отсутствие определенного форматирования, форматирование в конце строки или стандартные размеры отступов. Как и в большинстве других случаев, стандартные отступы становятся наилучшим решением с точки зрения аккуратности, единообразия, удобства для чтения и исправления. Листинг 31-62 содержит два примера заголовков методов без определенного форматирования:

**Листинг 31-62. Примеры заголовков методов, не использующих определенного форматирования (C++)**

```
bool  ReadEmployeeData(int  maxEmployees,EmployeeList  *employees,
    EmployeeFile *inputFile,int *employeeCount,bool  *isInputError)
...

void  InsertionSort(SortArray data,int  firstElement,int  lastElement)
```

Эти заголовки методов исключительно утилитарны. Компьютер может их прочитать, как, впрочем, он может прочитать и любой другой формат заголовков, однако у людей они вызовут затруднения. Нужно очень постараться, чтобы придумать еще более нечитаемый вариант.

В качестве другого подхода к форматированию заголовка метода можно предложить форматирование в конце строки, обычно выглядящее неплохо. В листинге 31-63 приведены те же заголовки, отформатированные по-другому:

**Листинг 31-63. Примеры заголовков методов с посредственным форматированием в конце строки (C++)**

```
bool ReadEmployeeData( int                maxEmployees,
                       EmployeeList       *employees,
                       EmployeeFile       *inputFile,
                       int                *employeeCount,
                       bool               *isInputError )

...
void InsertionSort( SortArray   data,
                    int         firstElement,
                    int         lastElement )
```

Этот подход аккуратен и эстетически привлекателен. Его проблема в том, что он требует больших усилий при сопровождении, а сложные в сопровождении стили в результате вовсе не сопровождаются. Допустим, имя функции *ReadEmployeeData()* изменится на *ReadNewEmployeeData()*. Это нарушит выравнивание первой строки относительно остальных 4-х строк. Вам придется переформатировать 4 строки списка параметров, выравнивая их относительно новой позиции *maxEmployees*, вызванной более длинным именем функции. Вам может даже не хватить места с правой стороны экрана, поскольку элементы и так уже далеко сдвинуты вправо.

**Перекрестная ссылка** О применении параметров методов см. раздел 7.5.

Примеры, приведенные в листинге 31-64, отформатированы с применением стандартных отступов. Они выглядят также эстетически привлекательно, но их сопровождение требует гораздо меньше усилий.

**Листинг 31-64. Примеры заголовков методов, содержащих отступы стандартного размера (C++)**

```
public bool ReadEmployeeData(
   int maxEmployees,
   EmployeeList *employees,
   EmployeeFile *inputFile,
   int *employeeCount,
   bool *isInputError
)
...

public void InsertionSort(
   SortArray data,
   int firstElement,
   int lastElement
)
```

Такой стиль лучше выдерживает модификацию. Изменение имени метода не влияет на остальные параметры. При добавлении/удалении параметров необходимо изменить только одну строку — плюс-минус запятую в предыдущей. Визуальные сигналы аналогичны схемам отступов, применяемым в циклах или операторах *if*. Поиск значимой информации о методах не требует просмотра разных частей страницы: всегда известно, где будет находиться эта информация.

Этот стиль можно использовать в Visual Basic, хотя при этом потребуются символы продолжения строки (листинг 31-65):

**Листинг 31-65. Пример заголовков методов, содержащих стандартные отступы, удобные для чтения и сопровождения (Visual Basic)**

```
Символ «_» используется как признак переноса строки.

Public Sub ReadEmployeeData ( _
    ByVal maxEmployees As Integer, _
    ByRef employees As EmployeeList, _
    ByRef inputFile As EmployeeFile, _
    ByRef employeeCount As Integer, _
    ByRef isInputError As Boolean _
)
```

# 31.8. Форматирование классов

В этом разделе рассказано об основных принципах размещения кода в классах. В первом подразделе вы узнаете, как отформатировать интерфейс класса, во втором — как оформлять реализацию класса, в последнем — мы обсудим организацию файлов и программ.

## Форматирование интерфейсов классов

**Перекрестная ссылка** О документировании классов см. подраздел «Комментирование классов, файлов и программ» раздела 32.5. О разнице между хорошими и плохими классами см. главу 6.

Соглашение о размещении интерфейсов классов предусматривает перечисление членов класса в следующем порядке:

1. шапка с комментарием, содержащая описание класса и любые примечания, касающиеся общих вопросов его использования;
2. конструкторы и деструкторы;
3. открытые методы;
4. защищенные методы;
5. закрытые методы и члены-данные.

## Форматирование реализаций классов

Реализации классов в общем случае могут размещаться в следующем порядке:

1. шапка с комментарием, описывающая содержимое файла с классом;
2. данные класса;
3. открытые методы;
4. защищенные методы;
5. закрытые методы.

***Если файл содержит более одного класса, четко определяйте границы каждого класса*** Взаимосвязанные методы должны быть сгруппированы в классы. Читатель, просматривающий код, должен иметь возможность легко определить, где какой класс. Четко выделяйте каждый класс с помощью нескольких пустых строк между концом одного и началом следующего класса. Класс — как глава в книге. В книге вы начинаете каждую главу с новой страницы и выделяете ее заголовок крупным шрифтом. Подчеркивайте начало каждого класса подобным образом. Пример разделения классов показан в листинге 31-66:

**Листинг 31-66. Пример форматирования разделения между классами на C++**

Это последний метод в классе.

```
// Создаем строку, идентичную sourceString, за исключением пробелов,
// заменяемых подчеркиваниями.
void  EditString::ConvertBlanks(
   char *sourceString,
   char *targetString
   ) {
   Assert( strlen( sourceString ) <= MAX_STRING_LENGTH );
   Assert( sourceString != NULL );
   Assert( targetString != NULL );
   int charIndex = 0;
   do {
      if ( sourceString[ charIndex ] == " " ) {
         targetString[ charIndex ] = '_';
      }
      else {
         targetString[ charIndex ] = sourceString[ charIndex ];
      }
      charIndex++;
   } while sourceString[ charIndex ] != '\0';
}
```

Начало нового класса отмечается несколькими пустыми строками и именем этого класса.

```
//---------------------------------------------------------------------------
// МАТЕМАТИЧЕСКИЕ ФУНКЦИИ
//
// Этот класс содержит математические функции программы.
//---------------------------------------------------------------------------
```

Это первый метод в новом классе.

```
// Ищем арифметический максимум аргументов arg1 и arg2.
int Math::Max( int arg1, int arg2 ) {
   if ( arg1 > arg2 ) {
      return arg1;
   }
   else {
      return arg2;
   }
}
```

Этот метод отделяется от предыдущего только пустыми строками.

```
// Ищем арифметический минимум аргументов arg1 и arg2.
int Math::Min( int arg1, int arg2 ) {
   if ( arg1 < arg2 ) {
      return arg1;
   }
   else {
      return arg2;
   }
}
```

Избегайте излишне заметных комментариев в классах. Если вы выделяете каждый метод, комментируя его с помощью ряда звездочек, а не просто используя пустые строки, у вас будут трудности с эффективным выделением начала нового класса (листинг 31-67):

**Листинг 31-67. Пример излишнего форматирования класса (C++)**

```
//*************************************************************************
//*************************************************************************
// МАТЕМАТИЧЕСКИЕ ФУНКЦИИ
//
// Этот класс содержит математические функции программы.
//*************************************************************************
//*************************************************************************

//*************************************************************************
// Ищем арифметический максимум аргументов arg1 и arg2.
//*************************************************************************
int Math::Max( int arg1, int arg2 ) {
//*************************************************************************
   if ( arg1 > arg2 ) {
      return arg1;
   }
   else {
      return arg2;
   }
}

//*************************************************************************
// Ищем арифметический минимум аргументов arg1 и arg2.
//*************************************************************************
int Math::Min( int arg1, int arg2 ) {
//*************************************************************************
   if ( arg1 < arg2 ) {
      return arg1;
   }
   else {
      return arg2;
   }
}
```

В этом примере столь многое выделено звездочками, что в результате ни на чем нельзя сделать акцент. Программа превращается в звездное небо. И хотя это больше эстетический, а не технический вопрос, но в форматировании действует принцип «чем меньше, тем лучше».

Если вам нужно разделять части программы длинными строками специальных символов, разработайте их иерархию (от самых плотных к самым прозрачным) и не рассчитывайте исключительно на звездочки. Так, звездочки можно применять для разделения классов, пунктирную линию — для методов, а пустые строки — для важных комментариев. Воздерживайтесь от размещения по соседству двух строк звездочек или пунктира (листинг 31-68):

**Листинг 31-68. Пример хорошего форматирования с ограничениями (C++)**

```
//*************************************************************************
// МАТЕМАТИЧЕСКИЕ ФУНКЦИИ
//
// Этот класс содержит математические функции программы.
//*************************************************************************
```

Легковесность этой строки по сравнению со строкой звездочек визуально подчеркивает тот факт, что метод подчиняется классу.

```
//-------------------------------------------------------------------------
// Ищем арифметический максимум аргументов arg1 и arg2.
//-------------------------------------------------------------------------
int Math::Max( int arg1, int arg2 ) {
   if ( arg1 > arg2 ) {
      return arg1;
   }
   else {
      return arg2;
   }
}

//-------------------------------------------------------------------------
// Ищем арифметический минимум аргументов arg1 и arg2.
//-------------------------------------------------------------------------
int Math::Min( int arg1, int arg2 ) {
   if ( arg1 < arg2 ) {
      return arg1;
   }
   else {
      return arg2;
   }
}
```

Этот совет о разграничении нескольких классов в файле относится только к случаям, когда ваш язык ограничивает число файлов, используемых в программе. Если вы пишете на C++, Java, Visual Basic или других языках, поддерживающих большое количество исходных файлов, помещайте в каждый файл по одному классу,

если только у вас нет уважительных причин сделать иначе (например, объединить нескольких небольших классов, составляющих единый шаблон). Один класс, однако, может содержать группы методов, и эти группы можно выделять с помощью технологий, продемонстрированных выше.

## Организация файлов и программ

**Перекрестная ссылка** О документации см. подраздел «Комментирование классов, файлов и программ» раздела 32.5.

Методики форматирования классов являются частным случаем более общего вопроса форматирования: как организовывать классы и методы внутри файла и какие классы помещать в отдельный файл в первую очередь?

***Помещайте каждый класс в отдельный файл*** Файл — не просто место для хранения кода. Если ваш язык позволяет, файл должен содержать набор методов, имеющих одно общее назначение. Файл подчеркивает факт принадлежности этих методов к одному классу.

**Перекрестная ссылка** О разнице между классами и методами и о том, как сделать класс из набора методов, см. главу 6.

Все методы внутри файла составляют класс. Под классом понимается как фактический класс программы, так и некоторая логическая сущность, являющаяся частью вашего проекта.

Классы — это семантическая языковая концепция. Файлы — это физическая концепция операционной системы. Соответствие между классами и файлами случайно, и со временем оно продолжает ослабевать, поскольку все больше сред поддерживает помещение кода в базы данных или иными способами скрывают взаимосвязь методов, классов и файлов.

***Называйте файл в соответствии с именем класса*** Большинство проектов поддерживает однозначное соответствие между названиями классов и именами файлов. Так, для класса *CustomerAccount* будут созданы файлы *CustomerAccount.cpp* и *CustomerAccount.h*.

***Четко разделяйте методы внутри файла*** Отделяйте методы друг от друга с помощью хотя бы двух пустых строк. Пустые строки так же эффективны, как и длинные ряды звездочек или пунктира, но их гораздо проще набирать и сопровождать. Две или три строки нужны для визуального различия между пустыми строками внутри метода и строками, разделяющими методы (листинг 31-69:

**Листинг 31-69. Пример применения пустых строк между методами (Visual Basic)**

```
' ищем арифметический максимум аргументов arg1 и arg2
Function Max( arg1 As Integer, arg2 As Integer ) As Integer
   If ( arg1 > arg2 ) Then
      Max = arg1
   Else
      Max = arg2
   End If
End Function

Методы разделяют минимум две пустых строки.
```

```
' ищем арифметический минимум аргументов arg1 и arg2
Function Min( arg1 As Integer, arg2 As Integer ) As Integer
   If ( arg1 < arg2 ) Then
      Min = arg1
   Else
      Min = arg2
   End If
end Function
```

Пустые строки набирать легче любых других сепараторов, а выглядят они не хуже. Этот пример содержит три пустых строки, и поэтому разделение между методами выглядит заметней, чем пустые строки внутри каждого метода.

***Упорядочивайте методы по алфавиту*** Альтернативой группировке взаимосвязанных функций может служить их размещение в алфавитном порядке. Если вы не можете разбить программу на классы или ваш редактор не позволяет легко находить функции, алфавитный подход может ускорить поиск.

***Аккуратно упорядочивайте исходный файл на C++*** Далее приведена типичная организация файла с исходным кодом на C++:

1. комментарий с описанием файла;
2. файлы, включаемые директивой #include;
3. определения констант, относящиеся к нескольким классам (если файл содержит несколько классов);
4. перечисления, относящиеся к нескольким классам (если файл содержит несколько классов);
5. определения макросов;
6. определения типов, относящиеся к нескольким классам (если файл содержит несколько классов);
7. импортируемые глобальные переменные и функции;
8. экспортируемые глобальные переменные и функции;
9. переменные и функции, видимые только в этом файле;
10. классы, вместе с определениями констант, перечислений и типов, относящимися к конкретному классу.

### Контрольный список: форматирование

http://cc2e.com/3194

**Общие вопросы**

- ❑ Применяется ли форматирование в основном для логического структурирования кода?
- ❑ Может ли схема форматирования применяться единообразно?
- ❑ Позволяет ли форматирование получить код, который легко сопровождать?
- ❑ Улучшает ли схема форматирования читаемость кода?

**Управляющие структуры**

- ❑ Содержит ли код двойные отступы в парах *begin-end* или {}?
- ❑ Отделяются ли последовательные блоки друг от друга пустыми строками?
- ❑ Форматируются ли сложные выражения с учетом удобочитаемости?
- ❑ Форматируются ли блоки из одного оператора единообразно?

- ❑ Согласуется ли стиль форматирования операторов *case* с форматированием других управляющих структур?
- ❑ Отформатированы ли операторы *goto* так, что их применение очевидно?

**Отдельные операторы**

- ❑ Применяются ли пробелы для повышения удобочитаемости логических выражений, обращений к массивам и аргументов методов?
- ❑ Выглядит ли конец строки с незавершенным оператором очевидно некорректным?
- ❑ Сделан ли при переносе строк отступ стандартного размера?
- ❑ Не содержит ли каждая строка более одного оператора?
- ❑ Не содержат ли операторы побочных эффектов?
- ❑ Не содержит ли каждая строка более одного определения данных?

**Комментарии**

- ❑ Сделаны ли в комментариях отступы такого же размера, как и в коде, который они комментируют?
- ❑ Легко ли сопровождать принятый стиль комментариев?

**Методы**

- ❑ Отформатированы ли аргументы всех методов так, что каждый из них легко читать, исправлять и комментировать?
- ❑ Используются ли пустые строки для разделения составных частей метода?

**Классы, файлы и программы**

- ❑ Существует ли однозначное соответствие между классами и файлами для большинства классов и файлов?
- ❑ Если файл содержит несколько классов, сгруппированы ли методы каждого класса вместе, и можно ли четко выделить каждый класс?
- ❑ Разделяются ли методы в файле пустыми строками?
- ❑ Вместо более строгого организационного принципа не стоит ли упорядочить все методы по алфавиту?

# Дополнительные ресурсы

http://cc2e.com/3101

Большинство учебников по программированию упоминает о форматировании и стиле вскользь, углубленное обсуждение стиля программирования встречается редко, а проблемы форматирования прорабатываются еще реже. Вопросам форматирования и стиля посвящены следующие публикации.

Kernighan, Brian W. and Rob Pike. *The Practice of Programming.* Reading, MA: Addison-Wesley, 1999. В главе 1 обсуждается стиль программирования на языках C и C++.

Vermeulen, Allan, et al. *The Elements of Java Style*. Cambridge University Press, 2000.

Misfeldt, Trevor, Greg Bumgardner, and Andrew Gray. *The Elements of C++ Style*. Cambridge University Press, 2004.

Kernighan, Brian W., and P. J. Plauger. *The Elements of Programming Style*, 2d ed. New York, NY: McGraw-Hill, 1978. Этот классический труд по стилю программирования — первый в этом жанре.

Абсолютно другой подход к удобочитаемости можно найти в следующей книге.

Knuth, Donald E. *Literate Programming*. Cambridge University Press, 2001. В этой подборке статей описывается концепция «грамотного программирования», объединяющего языки программирования и документирования. Кнут пишет о достоинствах грамотного программирования уже на протяжении 20 лет, но, хотя сам он может претендовать на титул Лучшего программиста планеты, грамотное программирование никак не войдет в моду. Прочтите примеры его кода, чтобы сформировать собственное мнение о причинах этой непопулярности.

## Ключевые моменты

- Главная цель визуального форматирования — это подчеркивание логической структуры кода. В критерии оценки достижения этой цели входят аккуратность, единообразие, удобство чтения и сопровождения кода.
- Критерий хорошего внешнего вида имеет вторичное, далеко не основное значение. Однако если другие критерии соблюдены, а лежащий в основе код написан хорошо, то форматирование будет выглядеть привлекательно.
- Visual Basic поддерживает явные блоки, а стандартное соглашение в Java предписывает их использование, поэтому, программируя на этих языках, вы можете применять явные блоки. В C++ хорошо смотрится эмуляция явных блоков или обозначение границ блоков с помощью пар *begin-end*.
- Структурирование кода само по себе имеет большое значение. Конкретные соглашения менее важны, чем сам факт, что вы последовательно применяете определенные соглашения. Договоренности по форматированию, соблюдаемые лишь от случая к случаю, могут сильно ухудшить читаемость кода.
- Многие аспекты форматирования сродни религиозным вопросам. Пытайтесь разделять объективные и субъективные предпочтения. Используйте явные критерии для обоснования вашей точки зрения при обсуждении стилевых предпочтений.

ГЛАВА 32

# Самодокументирующийся код

http://cc2e.com/3245

**Содержание**



**Связанные темы**

- Форматирование: глава 31
- Процесс программирования с псевдокодом: глава 9
- Классы: глава 6
- Высококачественные методы: глава 7
- Программирование как общение: разделы 33.5 и 34.3

Пишите код, исходя из того, что все программисты, которые будут сопровождать вашу программу, — склонные к насилию психопаты, знающие, где вы живете.

*Аноним*

Если стандарты документации разумны, большинству программистов нравится писать документацию. Как и форматирование, хорошая документация — свидетельство профессиональной гордости, выражаемой программистом в программе. Документация имеет много вариантов, и после ее общего обзора в этой главе мы рассмотрим специфический вид документации, известный как «комментарии».

## 32.1. Внешняя документация

**Перекрестная ссылка** У внешней документации имеются стандарты (см. раздел 32.6).

Документация программного проекта складывается из информации, содержащейся в листингах исходного кода, и внешней информации, которая обычно имеет форму отдельных документов или папок разработки блоков. В крупных

формальных проектах основная часть документации является внешней (Jones, 1998). Внешняя документация этапа конструирования обычно относится к более высокому уровню, чем код, и к более низкому, чем документация этапов определения проблемы, выработки требований и проектирования архитектуры.

***Папки разработки блоков*** Папка разработки блока (unit-development folder, UDF), или папка разработки ПО (software-development folder, SDF), — это неформальный документ, содержащий записи, используемые разработчиком во время конструирования. Термин «блок» здесь не имеет точного определения: обычно под ним понимается класс, а иногда — пакет или компонент. В первую очередь UDF служит для регистрации решений проектирования, которые нигде больше не документируются. Во многих проектах устанавливаются стандарты, определяющие минимальное содержание UDF — например, копии релевантных требований, высокоуровневые аспекты проектирования, реализуемые в блоке, копии стандартов разработки, листинги кода и заметки разработчика блока. Иногда заказчик требует, чтобы ему предоставлялись папки UDF; часто они предназначены только для внутреннего использования.

**Дополнительные сведения** О папках разработки блоков см. в «The Unit Development Folder (UDF): An Effective Management Tool for Software Development» (Ingrassia, 1976) и «The Unit Development Folder (UDF): A Ten-Year Perspective» (Ingrassia, 1987).

***Документ детального проектирования*** Документ детального проектирования описывает низкоуровневую конструкцию. Он очерчивает решения, принятые при проектировании классов или методов, рассмотренные варианты и мотивы выбора окончательных подходов. Иногда эта информация включается в формальный документ; при этом детальное проектирование обычно рассматривается как этап, отдельный от конструирования. Иногда этот документ состоит преимущественно из заметок разработчиков, собранных в UDF, но чаще эта документация присутствует только в самом коде.

## 32.2. Стиль программирования как вид документации

В отличие от внешней внутренняя документация включается в сам код. Это наиболее подробный вид документации, относящийся к уровню команд исходного кода. Внутренняя документация сильнее всего связана с кодом, поэтому при изменении кода она чаще остается корректной, чем документация иных видов.

Главный вклад в документацию уровня кода вносится не комментариями, а хорошим стилем программирования. Грамотное структурирование программы, использование простых и понятных подходов, удачный выбор имен переменных и методов, использование именованных констант вместо литералов, ясное форматирование, минимизация сложности потока управления и структур данных — все это аспекты стиля программирования.

Вот пример плохого стиля программирования:

**Пример плохого документирования кода из-за плохого стиля программирования (Java)**

```
for ( i = 2; i <= num; i++ ) {
meetsCriteria[ i ] = true;
}
for ( i = 2; i <= num / 2; i++ ) {
j = i + i;
while ( j <= num ) {
meetsCriteria[ j ] = false;
j = j + i;
}
}
for ( i = 1; i <= num; i++ ) {
if ( meetsCriteria[ i ] ) {
System.out.println ( i + " meets criteria." );
}
}
```

**Перекрестная ссылка** Здесь переменная *factorableNumber* используется исключительно ради пояснения операции. О добавлении переменных с целью пояснения операций см. подраздел «Упрощение сложных выражений» раздела 19.1.

Что, по-вашему, делает этот метод? Его непонятность не имеет никакого обоснования. Он плохо документирован, но дело не в отсутствии комментариев, а в плохом стиле программирования. Имена переменных неинформативны, а способ форматирования груб. Вот улучшенный вариант того же кода — простое улучшение стиля программирования делает его смысл гораздо яснее:

**Пример документирования кода без использования комментариев — за счет хорошего стиля (Java)**

```
for ( primeCandidate = 2; primeCandidate <= num; primeCandidate++ ) {
   isPrime[ primeCandidate ] = true;
}

for ( int factor = 2; factor < ( num / 2 ); factor++ ) {
   int factorableNumber = factor + factor;
   while ( factorableNumber <= num ) {
      isPrime[ factorableNumber ] = false;
      factorableNumber = factorableNumber + factor;
   }
}

for ( primeCandidate = 1; primeCandidate <= num; primeCandidate++ ) {
   if ( isPrime[ primeCandidate ] ) {
      System.out.println( primeCandidate + " is prime." );
   }
}
```

Одного взгляда на этот код достаточно, чтобы понять, что он имеет какое-то отношение к простым числам (prime numbers). Второй взгляд показывает, что он

находит простые числа от *1* до *Num*. Что касается первого фрагмента, то, взглянув на него пару раз, вы даже не поймете, где завершаются циклы.

Различие между двумя фрагментами с комментариями не связано — их вообще нет. Однако второй фрагмент читать гораздо лучше, приближаясь к Святому Граалю понятности — самодокументированию. Задача документирования такого кода во многом решается за счет хорошего стиля программирования. Если код написан хорошо, комментарии — всего лишь глазурь на пирожном читабельности.

### Контрольный список: самодокументирующийся код

http://cc2e.com/3252

**Классы**

- ❑ Формирует ли интерфейс класса согласованную абстракцию?
- ❑ Удачное ли имя присвоено классу? Описывает ли оно главную цель класса?
- ❑ Ясно ли интерфейс описывает применение класса?
- ❑ Достаточно ли абстрактен интерфейс класса, чтобы можно было не думать о том, как реализованы его сервисы? Можете ли вы рассматривать класс как «черный ящик»?

**Методы**

- ❑ Точно ли имя каждого метода описывает выполняемые в нем действия?
- ❑ Выполняет ли каждый метод одну и только одну хорошо определенную задачу?
- ❑ Все ли части метода, которые целесообразно поместить в отдельные методы, сделаны отдельными методами?
- ❑ Очевиден ли и ясен ли интерфейс каждого метода?

**Имена данных**

- ❑ Достаточно ли описательны имена типов, чтобы помогать документировать объявления данных?
- ❑ Удачно ли названы переменные?
- ❑ Переменные используются только с той целью, в соответствии с которой они названы?
- ❑ Присвоены ли счетчикам циклов более информативные имена, чем *i*, *j* и *k*?
- ❑ Используете ли вы грамотно названные перечисления вместо самодельных флагов или булевых переменных?
- ❑ Используете ли вы именованные константы вместо магических чисел или магических строк?
- ❑ Проводят ли конвенции именования различия между именами типов, перечислений, именованных констант, локальных переменных, переменных класса и глобальных переменных?

**Организация данных**

- ❑ Используете ли вы дополнительные переменные для пояснения кода?
- ❑ Сгруппированы ли обращения к переменным?
- ❑ Просты ли типы данных? Способствуют ли они минимизации сложности?
- ❑ Обращаетесь ли вы к сложным данным при помощи абстрактных методов доступа (абстрактных типов данных)?

**Управление**

- ❑ Очевиден ли номинальный путь выполнения кода?
- ❑ Сгруппированы ли связанные операторы?
- ❑ Вынесены ли относительно независимые группы операторов в отдельные методы?
- ❑ Следует ли нормальный вариант после *if*, а не после *else*?
- ❑ Просты ли управляющие структуры? Способствуют ли они минимизации сложности?
- ❑ Выполняет ли цикл одну и только одну функцию, как это делает хорошо спроектированный метод?
- ❑ Минимизировали ли вы вложенность?
- ❑ Упростили ли вы булевы выражения путем использования дополнительных булевых переменных, булевых функций и таблиц принятия решений?

**Форматирование**

- ❑ Характеризует ли форматирование программы ее логическую структуру?

**Проектирование**

- ❑ Прост ли код? Избегаете ли вы «хитрых» решений?
- ❑ Скрыты ли детали реализации настолько, насколько возможно?
- ❑ Стараетесь ли вы писать программу в терминах проблемной области, а не структур вычислительной техники или языка программирования?

## 32.3. Комментировать или не комментировать?

Комментарии легче написать плохо, а не хорошо, и комментирование иногда бывает скорее вредным, чем полезным. Жаркие дискуссии по поводу достоинств комментирования часто напоминают философские дебаты о моральных достоинствах, и это наводит меня на мысль о том, что, будь Сократ программистом, между ним и его учениками могла бы произойти следующая беседа.

### Комменто

*Действующие лица:*

| | |
|---|---|
| ФРАСИМАХ | Неопытный пурист теории, который верит всему, что читает. |
| КАЛЛИКЛ | Закаленный в боях представитель старой школы — «настоящий» программист. |
| ГЛАВКОН | Молодой, самоуверенный, энергичный программист. |
| ИСМЕНА | Опытная разработчица, уставшая от громких обещаний и просто желающая найти несколько работающих методик. |
| СОКРАТ | Мудрый опытный программист. |

*Мизансцена:*

Завершение ежедневного собрания группы

— Желает ли кто-то обсудить еще что-нибудь, прежде чем мы вернемся к работе? — спрашивает Сократ.

— Я хочу предложить стандарт комментирования для наших проектов, — говорит Фрасимах. — Некоторые наши программисты почти не комментируют свой код, а всем известно, что код без комментариев нечитаем.

— Ты, должно быть, еще менее опытен, чем я думал, — отвечает Калликл. — Комментарии — это академическая панацея, и любому, кто писал реальные программы, известно, что комментарии затрудняют чтение кода, а не облегчают. Естественный язык менее точен, чем Java или Visual Basic, и страдает от избыточности, тогда как операторы языков программирования лаконичны и попадают в самое яблочко. Если кто-то не может написать ясный код, разве ему удастся написать ясные комментарии? Кроме того, комментарии устаревают при изменениях кода. Доверяя устаревшим комментариям, ты сам себе роешь яму.

— Полностью согласен, — вступает в разговор Главкон. — Щедро прокомментированный код читать труднее, потому что в этом случае читать приходится больше. Мало того, что я должен читать код, так мне нужно читать еще и кучу комментариев!

— Подождите минутку, — вставляет свои две драхмы Исмена, ставя чашку кофе. — Конечно, злоупотребление комментариями возможно, но хорошие комментарии ценятся на вес золота. Мне приходилось сопровождать код как с комментариями, так и без них, и, будь у меня выбор, я бы предпочла первый вариант. Не думаю, что нам нужен стандарт, заставляющий писать один комментарий на каждые *x* строк кода, но побудить всех программистов комментировать код не помешает.

— Если комментарии — пустая трата времени, почему все их используют, Калликл? — спрашивает Сократ.

— Или потому, что таково требование, или потому, что человек прочитал где-то о пользе комментариев. Ни один программист, когда-либо задумавшийся об этом, не пришел к выводу, что комментарии полезны.

— Исмена считает, что они полезны. Она уже три года сопровождает твой код без комментариев и чужой код с комментариями. Ее мнение ты уже слышал. Что ты скажешь на это?

— Комментарии бесполезны, потому что они просто повторяют код в более многословной...

— Подожди, — прерывает Калликла Фрасимах. — Хорошие комментарии не повторяют код и не объясняют его. Они поясняют его цель. Комментарии должны объяснять намерения программиста на более высоком уровне абстракции, чем код.

— Верно, — соглашается Исмена. — Если я ищу фрагмент, который мне нужно изменить или исправить, я просматриваю комментарии. Комментарии, повторяющие код, на самом деле бесполезны, потому что все уже сказано в самом коде. Когда я читаю комментарии, я хочу, чтобы они напоминали оглавление книги. Комментарии должны помочь мне найти нужный раздел, а после этого я начну читать код. Гораздо быстрее прочитать одно предложение на обычном языке, чем анализировать 20 строк кода на языке программирования.

Исмена наливает себе вторую чашку кофе.

— Мне кажется, что люди, отказывающиеся писать комментарии, (1) думают, что их код понятнее, чем мог бы быть, (2) считают, что другие программисты гораздо сильнее интересуются их кодом, чем есть на самом деле, (3) думают, что другие программисты умнее, чем есть на самом деле, (4) ленятся или (5) боятся, что кто-то другой узнает, как работает их код.

— В этом смысле очень помогли бы обзоры кода, Сократ, — продолжает Исмена. — Если кто-то утверждает, что комментарии писать не нужно, и получает во время обзора кучу вопросов — если сразу несколько коллег спрашивают его: „Что происходит в этом фрагменте твоего кода?“ — он начинает писать комментарии. Если программист не дойдет до этого сам, его руководитель сможет заставить его писать комментарии.

— Я не утверждаю, Калликл, что ты ленишься или боишься, что кто-то другой поймет твой код. Я работала с твоим кодом и могу сказать, что ты — один из лучших программистов компании. Но будь чуточку добрее, а? Мне было бы легче сопровождать твой код, если бы ты писал комментарии.

— Но это пустая трата времени, — не сдается Калликл. — Код хорошего программиста должен быть самодокументирующимся: все, что нужно знать другим программистам, должно быть в самом коде.

— Нет! — Фрасимах вскакивает со стула. — В коде и так уже есть все, что нужно знать компилятору! С таким же успехом можно было бы сказать, что все, что нужно знать другим программистам, уже есть в двоичном исполняемом файле! Если бы мы только были достаточно умны, чтобы уметь читать его! Информации о том, что программист *собирался* сделать, в коде нет.

Фрасимах замечает, что вскочил с места, и садится.

— Сократ, это немыслимо. Почему мы спорим о важности комментариев? Во всех книгах, которые я читал, говорится, что комментарии полезны и что на них не стоит экономить. Мы зря теряем время.

— Успокойся, Фрасимах. Спроси у Калликла, как давно он программирует.

— Действительно, как давно, Калликл?

— Ну, я начал с системы Акрополь IV где-то 15 лет назад. Думаю, что я стал свидетелем рождения и гибели примерно десятка крупных приложений. А еще в десятке проектов я работал над крупными компонентами. Две из этих систем включали более полумиллиона строк кода, так что я знаю, о чем говорю. Комментарии совершенно бесполезны.

Ясно, что на некотором уровне комментарии *должны* быть полезны. Думать иначе означало бы полагать, что понятность программы не зависит от того, сколько информации о ней уже известно читающему программу человеку.

*Б. Шейл (B. A. Sheil)*

Сократ бросает взгляд на более молодого программиста.

— Как говорит Калликл, с комментариями действительно связано много проблем, и ты не поймешь это, пока не приобретешь больше опыта. Если комментировать код неграмотно, комментарии не просто бесполезны — они вредны.

— Они бесполезны, даже если комментировать код грамотно, — заявляет Калликл. — Комментарии менее точны, чем язык программирования. Лично я думаю, что лучше вообще их не писать.

— Постой, — говорит Сократ. — Исмена согласна с тем, что комментарии менее точны, но она утверждает, что комментарии поднимают программиста на более высокий уровень абстракции, а все мы знаем, что абстракция — один из самых эффективных инструментов, имеющихся в нашем распоряжении.

— Я с этим не согласен, — заявляет Главкон. — Нам следует концентрироваться не на комментариях, а на читабельности кода. При рефакторинге большинство

моих комментариев исчезает. После рефакторинга мой код может включать 20 или 30 вызовов методов, не нуждающихся в каких бы то ни было комментариях. Хороший программист способен определять цель автора по самому коду; к тому же какой толк в чтении о чьей-то цели, если известно, что код содержит ошибку?

Главкон умолкает, удовлетворенный своим вкладом в беседу. Калликл кивает.

— Похоже, вам никогда не приходилось изменять чужой код, — говорит Исмена. Калликл делает вид, что его очень заинтересовали плитки на потолке.

— Почему бы вам не попробовать прочитать собственный код через шесть месяцев или год после его написания? Вы можете развивать и навык чтения кода, и навык его комментирования. Никто не заставляет вас выбирать что-то одно. Если вы читаете роман, вам, может, и не нужны названия глав. Но при чтении технической книги вам, наверное, хотелось бы иметь возможность быстро найти то, что вы ищете. Тогда мне не пришлось бы переходить в режим сверхсосредоточенности и читать сотни строк кода для нахождения двух строк, которые я хочу изменить.

— Хорошо, я понимаю, что возможность быстрого просмотра кода была бы удобной, — говорит Главкон. Он видел некоторые из программ Исмены, и они не оставили его равнодушным. — Но как быть с другим утверждением Калликла — что комментарии устаревают по мере изменений кода? Я программирую лишь пару лет, но даже я знаю, что никто не обновляет свои комментарии.

— Ну, и да, и нет, — говорит Исмена. — Если вы считаете комментарии неприкосновенными, а код подозрительным, у вас серьезные проблемы. На самом деле несоответствие между комментарием и кодом обычно говорит о том, что неверно и то, и другое. Если какие-то комментарии плохи, это не означает, что само комментирование плохо. Пойду налью себе еще чашку кофе.

Исмена выходит из комнаты.

— Мое главное возражение против комментариев, — заявляет Калликл, — в том, что они тратят ресурсы.

— Можете ли вы предложить способ минимизации времени написания комментариев? — спрашивает Сократ.

— Проектирование методов на псевдокоде, преобразование псевдокода в комментарии и написание соответствующего им кода, — говорит Главкон.

— OK, это сработает, если комментарии не будут повторять код, — утверждает Калликл.

— Написание комментария заставляет вас лучше подумать над тем, что делает ваш код, — говорит Исмена, возвращаясь с новой чашкой кофе. — Если комментарии писать трудно, это означает, что код плох или вы недостаточно хорошо его понимаете. В обоих случаях вы должны поработать над кодом еще, так что время, потраченное на его комментирование, не пропадает — оно указывает вам на необходимую работу.

— Хорошо, — подводит итоги Сократ. — Не думаю, что у нас остались еще вопросы. Похоже, Исмена сегодня была самой убедительной. Мы будем поощрять комментирование, но не будем относиться к нему простодушно. Мы будем выполнять обзоры кода, чтобы все поняли, какие комментарии на самом деле полезны. Если у вас возникнут проблемы с пониманием кода другого программиста, подскажите ему, как он может его улучшить.

# 32.4. Советы по эффективному комментированию

Пока существуют плохо определенные цели, странные ошибки и нереалистичные сроки, будут существовать и Настоящие Программисты, желающие немедленно приступить к Решению Проблемы и оставляющие документацию на потом. Да здравствует Фортран!

*Эд Пост (Ed Post)*

Что делает следующий метод?

### Загадочный метод номер один (Java)

```
// вывод сумм чисел 1..n для всех n от 1 до num
current = 1;
previous = 0;
sum = 1;
for ( int i = 0; i < num; i++ ) {
   System.out.println( "Sum = " + sum );
   sum = current + previous;
   previous = current;
   current = sum;
}
```

Этот метод вычисляет первые *num* чисел Фибоначчи. Стиль кодирования этого метода чуть лучше, чем стиль метода, указанного в начале главы, но комментарий неверен, и если вы слепо доверяете комментариям, то, пребывая в блаженном неведении, пойдете в неверном направлении.

А что скажете по поводу этого метода?

### Загадочный метод номер два (Java)

```
// присваивание переменной "product" значения переменной "base"
product = base;

// цикл от 2 до "num"
for ( int i = 2; i <= num; i++ ) {
   // умножение "base" на "product"
   product = product * base;
}
System.out.println( "Product = " + product );
```

Этот метод возводит целое число *base* в целую степень *num*. Комментарии в этом методе верны, но они не говорят о коде ничего нового. Это просто более многословная версия самого кода.

Наконец, еще один метод:

### Загадочный метод номер три (Java)

```
// вычисление квадратного корня из Num с помощью аппроксимации Ньютона-Рафсона
r = num / 2;
while ( abs( r - (num/r) ) > TOLERANCE ) {
   r = 0.5 * ( r + (num/r) );
}
System.out.println( "r = " + r );
```

Этот метод вычисляет квадратный корень из *num*. Код неидеален, но комментарий верен.

Какой метод понять было легче всего? Все они написаны довольно плохо — особенно неудачны имена переменных. По сути эти методы иллюстрируют достоинства и недостатки внутренних комментариев. Комментарий Метода 1 неверен. Комментарии Метода 2 просто повторяют код и потому бесполезны. Только комментарий Метода 3 оправдывает свое существование. Наличие плохих комментариев хуже, чем их отсутствие. Методы 1 и 2 лучше было бы оставить вообще без комментариев.

В следующих подразделах я приведу советы по написанию эффективных комментариев.

## Виды комментариев

Комментарии можно разделить на шесть категорий, описанных ниже.

### Повторение кода

Повторяющий комментарий переформулирует код иными словами и только заставляет больше читать, не предоставляя дополнительной информации.

### Объяснение кода

Такие комментарии обычно служат для объяснения сложных, хитрых или нестабильных фрагментов кода. В этих ситуациях они полезны, но обычно только потому, что код непонятен. Если код настолько сложен, что требует объяснения, почти всегда разумнее улучшить код, а не добавлять комментарии. Сделайте сам код более ясным, а потом добавьте в него резюмирующие комментарии или комментарии цели.

### Маркер в коде

Этот тип предназначен не для того, чтобы оставаться в коде. Такая пометка указывает программисту на то, что работа еще не выполнена. Некоторые разработчики делают маркеры синтаксически некорректными (например, `******`), чтобы о невыполненной работе напомнил компилятор. Другие включают в комментарии определенный ряд символов, не приводящий к ошибке компиляции и служащий для поиска нужного фрагмента.

Мало что может сравниться с чувствами программиста, получившего от заказчика сообщение о проблеме в коде и увидевшего во время отладки что-нибудь вроде:

```
return NULL; // ****** НЕ СДЕЛАНО! ИСПРАВИТЬ ДО ВЫПУСКА ПРОГРАММЫ!!!
```

Если вы поставили заказчикам дефектную программу, это плохо; если же вы *знали*, что она дефектна, это просто ужасно.

Стандартизуйте стиль комментариев-маркеров, иначе одни программисты будут использовать для этого символы `*******`, другие — *!!!!!!*, третьи — *TBD*, а четвертые — что-то еще. Применение разных нотаций повышает вероятность ошибок при механическом поиске незавершенных фрагментов кода или вообще делает его невозможным. Стандартизация стиля маркирования позволяет сделать механичес-

кий поиск незавершенных фрагментов одним из действий контрольного списка перед выпуском программы и предотвратить проблему *ИСПРАВИТЬ ДО ВЫПУСКА!!!*. Некоторые редакторы поддерживают теги «to do» и позволяют легко искать их.

### Резюме кода

Резюмирующие комментарии выражают суть нескольких строк кода в одном-двух предложениях. Эти комментарии более полезны, чем повторяющие, потому что программист, читающий программу, может просматривать их быстрее, чем код. Резюмирующие комментарии особенно полезны, если программу будет изменять не только ее автор.

### Описание цели кода

Комментарий уровня цели объясняет намерения автора кода. Такие комментарии относятся скорее к уровню проблемы, чем к уровню ее решения. Например, комментарий:

```
-- получение информации о текущем сотруднике
```

является комментарием цели, тогда как:

```
-- обновление объекта employeeRecord
```

является резюмирующим комментарием, сформулированным в терминах решения.

Шестимесячное исследование, проведенное в IBM, показало, что программисты, отвечавшие за сопровождение программы, «чаще всего говорили, что труднее всего было понять цель автора кода» (Fjelstad and Hamlen, 1979). Различие между комментариями цели и резюмирующими не всегда очевидно и обычно не играет особой роли. В этой главе вы найдете массу примеров комментариев цели.

### Информация, которую невозможно выразить в форме кода

Кое-какую информацию, не выразимую в форме кода, все-таки нужно включить в программу. Эта категория комментариев включает уведомления об авторских правах и конфиденциальности данных, номера версий и другие детали, замечания о структуре кода, ссылки на релевантные документы требований или архитектуры, ссылки на Интернет-ресурсы, соображения по оптимизации, комментарии, нужные таким инструментам редактирования, как Javadoc, Doxygen и т. д.

К трем видам комментариев, приемлемых в готовом коде, относятся резюмирующие комментарии, комментарии цели и информация, не выразимая в форме кода.

## Эффективное комментирование

Эффективное комментирование отнимает не так много времени. Избыток комментариев не менее плох, чем недостаток, а оптимальной середины можно достичь без особых проблем.

Написание комментариев может отнимать много времени по двум причинам. Во-первых, стиль комментирования может быть трудоемким или нудным. Если это

так, измените стиль. Стиль комментирования, требующий значительных усилий, очень трудно сопровождать. Если комментарии трудно изменять, их просто не будут изменять, из-за чего они станут неверными и обманчивыми, что хуже, чем полное отсутствие комментариев.

Во-вторых, комментирование может быть сложным из-за трудностей с формулированием действий программы в словах. Обычно это говорит о том, что вы не понимаете работу программы. Время, уходящее на «комментирование», на самом деле тратится на лучшее понимание программы, и его все равно придется потратить, комментируете вы код или нет.

Вот некоторые рекомендации по эффективному комментированию.

***Используйте стили, не препятствующие изменению комментариев*** Поддерживать слишком причудливый стиль надоедает. Попробуйте, например, определить часть следующего комментария, которую никто не будет поддерживать:

**Пример стиля комментирования, который трудно поддерживать (Java)**

```
//  Переменная         Смысл
//  -----           -----
//  xPos .......... Координата X (в метрах)
//  yPos .......... Координата Y (в метрах)
//  ndsCmptng...... Требуется ли вычисление? (= 0, если вычисление требуется;
//                                            = 1, если вычисление не требуется)
//  ptGrdTtl....... Общее число точек
//  ptValMax....... Максимальное значение в точке
//  psblScrMax..... Максимально возможная сумма
```

Если вы подумали про точки (.....), вы совершенно правы! Красиво, конечно, но список хорош и без них. Точки затрудняют изменение комментариев, и, если есть выбор (а он обычно есть), комментарии лучше сделать правильными, чем красивыми.

Вот пример другого популярного стиля, трудного в сопровождении:

**Пример стиля комментирования, который трудно поддерживать (C++)**

```
/*************************************************************************
 * класс:  GigaTron (GIGATRON.CPP)                                       *
 *                                                                       *
 * автор:  Дуайт К. Кодер                                                *
 * дата:   4 июля 2014 года                                              *
 *                                                                       *
 * Методы, управляющие самым передовым инструментом оценки кода.         *
 * Точкой входа в эти методы является метод EvaluateCode(),              *
 * расположенный в конце файла.                                          *
*************************************************************************/
```

Это симпатичный блочный комментарий. По оформлению ясно, что весь блок — единое целое; начало и окончание блока также бросаются в глаза. Что неясно, так это легкость изменения этого блока. При его изменении почти наверняка придется заново выстраивать аккуратные столбцы звездочек. На практике это означает, что из-за слишком большого объема работы такой комментарий поддержи-

вать не будут. Если можно выровнять столбцы звездочек одним нажатием клавиши, прекрасно — можете использовать этот стиль. Проблема не в звездочках, а в том, что их трудно поддерживать. Следующий комментарий выглядит почти так же хорошо, а сопровождать его куда легче:

**Пример стиля комментирования, который легко поддерживать (C++)**

```
/************************************************************************
   класс:  GigaTron (GIGATRON.CPP)

   автор:  Дуайт К. Кодер
   дата:   4 июля 2014 года

   Методы, управляющие самым передовым инструментом оценки кода.
   Точкой входа в эти метода является метод EvaluateCode(),
   расположенный в конце файла.
  ************************************************************************/
```

Вот стиль, особенно трудный в сопровождении:

**Пример стиля комментирования, который трудно поддерживать (Visual Basic)**

```
‘   настройка перечисления Color
’    +-------------------------+
    ...

’   настройка перечисления Vegetable
’    +-----------------------------+
    ...
```

Трудно сказать, какой смысл имеют плюсы в начале и в конце каждой пунктирной линии, зато легко догадаться, что при каждом изменении комментария эту линию придется адаптировать, чтобы конечный плюс находился на своем месте. А если комментарий перейдет на вторую строку? Что тогда вы сделаете с плюсом? Выбросите из комментария какие-то слова, чтобы он занимал только одну строку? Сделаете длину обеих строк одинаковой? При попытке согласованного применения этой методики проблемы быстро множатся.

На подобных рассуждениях основан и популярный совет использовать в Java и C++ синтаксис *//* для однострочных и синтаксис */* … */* — для более длинных комментариев:

**Пример использования разного синтаксиса комментирования с разными целями (Java)**

```
// Это короткий комментарий.
...
/* Это гораздо более длинный комментарий. Восемьдесят семь лет назад наши отцы
основали на этом континенте новую нацию, взращенную в условиях свободы, преданную
принципу, согласно которому все люди созданы равными. Сейчас мы ведем великую
Гражданскую войну, в которой проверяется, может ли эта нация или любая другая,
воспитанная в таком же духе и преданная таким же идеалам, существовать дальше.
```

```
Мы встретились сейчас на поле одной из величайших битв этой войны. Мы пришли сюда
для того, чтобы отвести часть этого поля для последнего места успокоения тех, кто
отдал здесь свои жизни ради того, чтобы эта нация могла жить. Очень правильно, что
мы делаем это.
*/
```

Первый комментарий легко сопровождать, пока он короток. Если комментарий длиннее, создание столбцов двойных слэшей, ручное разбиение текста на строки и другие подобные действия начинают утомлять, поэтому для многострочных комментариев больше подходит синтаксис /* … */.

Суть сказанного в том, что вам следует обращать внимание на то, как вы тратите свое время. Если вы вводите или удаляете дефисы для выравнивания плюсов, вы не программируете — вы занимаетесь ерундой. Найдите более эффективный стиль. В одном из предыдущих примеров дефисы и плюсы можно было бы удалить без ущерба для комментариев. Если вы хотите подчеркнуть какие-то комментарии, найдите иной способ, но не дефисы с плюсами. Например, вы могли бы подчеркивать комментарии строками, состоящими из одинакового числа дефисов. Такие строки не требовали бы сопровождения, а для их ввода можно было бы использовать макрос текстового редактора.

***Используйте для сокращения времени, уходящего на комментирование, процесс программирования с псевдокодом*** Формулирование кода в виде комментариев перед его написанием обеспечивает целый ряд преимуществ. Когда вы завершите работу над кодом, комментарии уже будут готовы. Вам не нужно будет выделять время на комментирование. Создавая высокоуровневый псевдокод перед написанием соответствующего низкоуровневого кода на языке программирования, вы также получите все выгоды проектирования.

**Перекрестная ссылка** О процессе программирования с псевдокодом см. главу 9.

***Интегрируйте комментирование в свой стиль разработки*** Альтернативой интеграции комментирования в стиль разработки является откладывание комментирования на конец проекта, но у этого подхода слишком много недостатков: комментирование становится самостоятельной задачей и кажется более объемной работой, чем написание комментариев небольшими порциями. Комментирование, выполняемое позднее, требует больше времени, потому что вы должны вспомнить или определить, что код делает, вместо того чтобы просто написать, о чем вы думаете прямо сейчас. Кроме того, оно менее точно, поскольку со временем вы забываете предположения или тонкости, имевшие место на этапе проектирования.

Противники комментирования по мере написания кода часто утверждают: «если вы сосредоточены на коде, не следует отвлекаться на написание комментариев». Однако если написание кода требует столь сильной концентрации, что комментирование нарушает ход мыслей, вам на самом деле нужно спроектировать фрагмент сначала на псевдокоде и потом преобразовать псевдокод в комментарии. Код, требующий такой сильной концентрации, — тревожный симптом.

Если проект трудно закодировать, упростите его до начала работы над комментариями или кодом. Если для организации своего мышления вы будете использовать псевдокод, кодирование станет легким, а комментирование автоматическим.

***Производительность не является разумной причиной отказа от комментирования*** Одним периодически повторяющимся атрибутом технологических волн, обсуждавшихся в разделе 4.3, являются интерпретируемые среды, в которых комментарии вызывают чувствительное снижение производительности. В 1980-х это относилось к программам, написанным на Basic. В 1990-х — к asp-страницам. Сейчас похожая проблема связана с кодом JavaScript и другим кодом, который нужно передавать по сети.

В каждом из этих случаев решением является не отказ от комментариев, а создание готовой версии кода, отличающейся от разрабатываемой. Как правило, для этого код «пропускают» через утилиту, удаляющую комментарии.

### Оптимальная плотность комментариев

По данным Кейперса Джонса, исследования, проведенные в IBM, показали, что понятность кода достигала пика при одном комментарии примерно на 10 операторов. Как меньшая, так и более высокая плотность комментариев осложняла понимание кода (Jones, 2000).

Подобные данные можно использовать неверно, и в проектах иногда принимаются стандарты типа «комментарии должны встречаться в программе не реже, чем каждые пять строк кода». Этот стандарт устраняет симптом написания недостаточно ясного кода, но не причину.

Если вы эффективно используете процесс программирования с псевдокодом, вы, вероятно, тоже в итоге получите один комментарий на каждые несколько строк кода. Однако плотность комментариев будет побочным эффектом самого процесса. Вместо концентрации на плотности комментариев сосредоточьтесь на их эффективности. Если комментарии будут характеризовать цель написания кода и соответствовать другим критериям, указанным в этой главе, комментариев будет достаточно.

## 32.5. Методики комментирования

Существует несколько методик комментирования, различающихся по тому, к какому уровню относится комментарий: к программе, файлу, методу, абзацу кода или отдельной строке.

### Комментирование отдельных строк

Если код хорош, необходимость комментирования отдельных строк возникает редко. Есть две возможных причины комментирования строки кода:

- строка довольно сложна и потому требует объяснения;
- строка когда-то содержала ошибку, и вы хотите отметить это.

Вот некоторые советы по комментированию отдельных строк кода.

***Избегайте комментариев, высосанных из пальца*** Много лет назад я слышал историю о программисте, которого подняли с постели исправить неработавшую программу. С автором программы, который к тому времени уже оставил компанию, связаться было невозможно. Программист раньше не работал с этой про-

граммой, и после тщательного изучения документации он обнаружил только такой комментарий:

```
MOV AX, 723h     ; R. I. P. L. V. B.
```

Поломав над ним голову всю ночь, программист в итоге все исправил и пошел домой спать. Несколько месяцев спустя он встретился с автором программы на конференции и узнал, что комментарий означал «Rest in peace, Ludwig van Beethoven» (Покойся в мире, Людвиг ван Бетховен). Бетховен умер в 1827 году, которому соответствует шестнадцатеричное значение 723. Необходимость использования значения 723h не имела никакого отношения к комментарию. &%@*?#%~!!!

### Комментарии в концах строк и связанные с ними проблемы

#### Пример кода с комментариями в концах строк (Visual Basic)

```
For employeeId = 1 To employeeCount
   GetBonus( employeeId, employeeType, bonusAmount )
   If employeeType = EmployeeType_Manager Then
      PayManagerBonus( employeeId, bonusAmount ) ' полная оплата
   Else
      If employeeType = EmployeeType_Programmer Then
         If bonusAmount >= MANAGER_APPROVAL_LEVEL Then
            PayProgrammerBonus( employeeId, StdAmt() ) ' стандартная оплата
         Else
            PayProgrammerBonus( employeeId, bonusAmount ) ' полная оплата
         End If
      End If
   End If
Next
```

Иногда комментарии в концах строк полезны, но вообще они создают несколько проблем. Чтобы они не нарушали восприятие структуры кода, их нужно выравнивать по правому краю кода. Если вы не будете их выравнивать, листинг будет выглядеть так, будто его пропустили через мясорубку. Как правило, комментарии в концах строк трудно форматировать. Если их много, выравнивание занимает массу времени. Это время тратится не на изучение кода, а исключительно на нудный ввод пробелов или символов табуляции.

В то же время комментарии в концах строк трудно поддерживать. При удлинении какой-либо строки с таким комментарием он сдвигается, что заставляет приводить в соответствие все остальные комментарии в концах строк. Стили, которые трудно поддерживать, никто не поддерживает, и после изменений кода комментарии начинают ухудшать его, а не улучшать.

Кроме того, комментарии в концах строк часто страдают от таинственности. Справа от строки обычно остается не так много места, и, если вы хотите написать комментарий в той же строке, он должен быть кратким. В результате целью комментирования становится формулирование комментариев в максимально краткой, а не максимально ясной форме.

***Не используйте комментарии в концах строк, относящиеся к одиночным строкам*** С комментариями в концах строк связаны не только практические, но и концептуальные проблемы. Вот пример набора комментариев в концах строк:

**Пример бесполезных комментариев в концах строк (C++)**

Здесь комментарии просто повторяют код.

```
memoryToInitialize = MemoryAvailable();     // получение объема доступной памяти
pointer = GetMemory( memoryToInitialize ); // получение указателя на память
ZeroMemory( pointer, memoryToInitialize ); // обнуление памяти
...
FreeMemory( pointer );                      // освобождение памяти
```

Общий недостаток комментариев в концах строк в том, что трудно придумать выразительный комментарий для одной строки кода. Большинство таких комментариев просто повторяют строку кода, что скорее вредит, чем помогает.

***Не используйте комментарии в концах строк, относящиеся к нескольким строкам кода*** Если комментарий в конце строки относится к нескольким строкам, форматирование не позволяет узнать, к каким именно:

**Пример непонятного комментария в конце строки (Visual Basic)**

```
For rateIdx = 1 to rateCount                    ' Вычисление цен со скидкой
   LookupRegularRate( rateIdx, regularRate )
   rate( rateIdx ) = regularRate * discount( rateIdx )
Next
```

Суть данного комментария ясна, но что́ он комментирует? Чтобы узнать, относится ли комментарий к отдельной команде или целому циклу, придется прочитать и комментарий, и код.

### Когда использовать комментарии в концах строк?

Ниже описаны три разумных причины использования комментариев в концах строк.

**Перекрестная ссылка** О других аспектах использования комментариев в концах строк для пояснения объявлений данных см. ниже подраздел «Комментирование объявлений данных».

***Используйте комментарии в концах строк для пояснения объявлений данных*** Комментарии в концах строк полезны для аннотирования объявлений данных, потому что с ними не связаны проблемы, характерные для аналогичных комментариев кода, если, конечно, строки имеют достаточную длину. При 132 символах в строке вы обычно можете написать выразительный комментарий около каждого объявления данных:

**Пример объявлений данных с хорошими комментариями в концах строк (Java)**

```
int boundary = 0;          // верхний индекс отсортированной части массива
String insertVal = BLANK; // элемент, вставляемый в отсортированную часть массива
int insertPos = 0;         // индекс вставки элемента в отсортиров. часть массива
```

***Не используйте комментарии в концах строк для вставки пометок во время сопровождения ПО*** Комментарии в концах строк иногда используют для регистрации изменений, вносимых в код после выпуска первоначальной версии программы. Эти комментарии обычно включают дату изменения и инициалы программиста; иногда — номер отчета об ошибке и т. д. Вот пример:

```
for i = 1 to maxElmts - 1   - исправлена ошибка #A423 01.10.05 (scm)
```

Наверное, написать такой комментарий после ночного сеанса отладки программы приятно, но в итоговом коде им не место. Их лучше включать в системы управления версиями. Комментарии должны объяснять, почему код работает *сейчас*, а не почему он когда-то не работал.

***Используйте комментарии в концах строк для обозначения концов блоков*** В комментариях в концах строк полезно отмечать окончание объемных блоков кода — например, циклов *while* или операторов *if*. Подробнее об этом см. ниже.

**Перекрестная ссылка** О комментариях в концах строк, служащих для обозначения концов блоков см. ниже подраздел «Комментирование управляющих структур».

За исключением нескольких специальных случаев комментарии в концах строк сопряжены с концептуальными проблемами и обычно служат для объяснения слишком сложного кода. Кроме того, их сложно форматировать и поддерживать. В общем, их лучше избегать.

## Комментирование абзацев кода

Большинство комментариев в хорошо документированной программе состоят из одного-двух предложений и описывают абзацы кода:

### Пример хорошего комментария абзаца кода (Java)

```
// перемена корней местами
oldRoot = root[0];
root[0] = root[1];
root[1] = oldRoot;
```

Этот комментарий не повторяет код — он описывает цель кода. Такие комментарии относительно легко поддерживать. Даже если вы найдете ошибку в коде, меняющем корни местами, изменять комментарий не придется. Комментарии, написанные не на уровне цели, поддерживать труднее.

***Пишите комментарии на уровне цели кода*** Описывайте цель блока кода, следующего за комментарием. Вот пример комментария, неэффективного из-за того, что он не относится к уровню цели:

### Пример неэффективного комментария (Java)

```
/* проверка символов строки "inputString", пока не будет
обнаружен
знак доллара или пока не будут проверены все символы
*/
done = false;
maxLen = inputString.length();
i = 0;
```

**Перекрестная ссылка** Этот код, выполняющий простой поиск символа в строке, служит только в качестве примера. В реальном коде вы использовали бы для этого встроенные библиотечные функции Java. О важности понимания возможностей языка см. подраздел «Читайте!» раздела 33.3.

```
while ( !done && ( i < maxLen ) ) {
   if ( inputString[ i ] == '$' ) {
      done = true;
   }
   else {
      i++;
   }
}
```

Прочитав код, можно понять, что цикл ищет символ $, и это полезно отразить в комментарии. Недостаток этого комментария в том, что он просто повторяет код и ничего не говорит о его цели. Следующий комментарий чуть лучше:

```
// поиск символа '$' в строке inputString
```

Этот комментарий лучше потому, что он говорит, что целью цикла является поиск символа *$*. Но мы все еще не знаем причину поиска символа *$* — иначе говоря, более глубокую цель цикла. Вот комментарий, который еще лучше:

```
// поиск терминального символа слова команды ($)
```

Этот комментарий на самом деле сообщает то, чего нет в листинге: что символ *$* завершает слово команды. Просто прочитав код, этого никак не понять, так что данный комментарий действительно важен.

При комментировании на уровне цели полезно подумать о том, как бы вы назвали метод, выполняющий то же самое, что и код, который вы хотите прокомментировать. Если каждый абзац кода имеет одну цель, это несложно: примером может служить комментарий в предыдущем фрагменте кода. Имя *FindCommandWordTerminator()* было бы вполне приемлемым именем метода. Другие варианты — *Find$InInputString()* и *CheckEachCharacterInInputStrUntilADollarSignIsFoundOrAllCharactersHaveBeenChecked()* — по очевидным причинам являются плохими (или некорректными) именами. Опишите суть кода как имя соответствующего метода, не используя сокращений и аббревиатур. Это описание и будет комментарием, и скорее всего он будет относиться к уровню цели.

***Во время документирования сосредоточьтесь на самом коде*** Строго говоря, сам код — это документация, которую всегда следует проверять в первую очередь. В предыдущем примере литерал *$* следовало бы заменить на именованную константу, а имена переменных должны были бы предоставлять больше информации о том, что происходит. Если бы вы хотели добиться максимальной читабельности, вы добавили бы переменную, содержащую результат поиска. Это провело бы ясное различие между индексом цикла и результатом выполнения цикла. Вот предыдущий пример, переписанный в хорошем стиле и дополненный хорошим комментарием:

#### Пример хорошего кода с хорошим комментарием (Java)

```
// поиск терминального символа слова команды
foundTheTerminator = false;
commandStringLength = inputString.length();
testCharPosition = 0;
```

```
while ( !foundTheTerminator && ( testCharPosition < commandStringLength ) ) {
   if ( inputString[ testCharPosition ] == COMMAND_WORD_TERMINATOR ) {
      foundTheTerminator = true;
```

Эта переменная содержит результат поиска.

```
      terminatorPosition = testCharPosition;
   }
   else {
      testCharPosition = testCharPosition + 1;
   }
}
```

Если код достаточно хорош, его цель можно понять при чтении и без комментариев. Фактически они даже могут стать избыточными, но от этого страдают очень немногие программы.

Наконец, этот код следовало бы выделить в метод с именем вроде *FindCommandWordTerminator()*. Аналогичный комментарий также полезен, но при изменении ПО комментарии устаревают и становятся неверными чаще, чем имена методов.

**Перекрестная ссылка** О вынесении фрагмента кода в отдельный метод см. также подраздел «Извлечение метода из другого метода» раздела 24.3.

***Придумывая комментарий абзаца, стремитесь ответить на вопрос «почему?», а не «как?»*** Комментарии, объясняющие, «как» что-то выполняется, обычно относятся к уровню языка программирования, а не проблемы. Цель операции почти невозможно выразить в таком комментарии, поэтому он часто оказывается избыточным. Разве следующий комментарий сообщает что-то такое, чего не ясно из самого кода?

**Пример комментария, отвечающего на вопрос «как?» (Java)**

```
// если флаг счета равен нулю
if ( accountFlag == 0 ) ...
```

Нет, не сообщает. А этот?

**Пример комментария , отвечающего на вопрос «почему?» (Java)**

```
// если создается новый счет
if ( accountFlag == 0 ) ...
```

Этот комментарий гораздо лучше, так как он говорит что-то, чего нельзя понять, исходя из самого кода. Код также можно улучшить, заменив *0* на элемент перечисления с выразительным именем и уточнив имя переменной. Вот самая лучшая версия этого кода и этого комментария:

**Пример дополнения хорошего комментария хорошим стилем программирования (Java)**

```
// если создается новый счет
if ( accountType == AccountType.NewAccount ) ...
```

Когда код достигает такого уровня читабельности, ценность комментариев становится сомнительной. В нашем случае после улучшения кода комментарий стал

избыточным, и его вполне можно удалить. Можно поступить и иначе, чуть изменив роль комментария:

**Пример комментария, играющего роль «заголовка раздела» (Java)**

```
// создание нового счета
if ( accountType == AccountType.NewAccount ) {
   ...
}
```

Если этот комментарий относится ко всему блоку кода после проверки условия *if*, он является резюмирующим комментарием, и не будет лишним сохранить его в качестве заголовка абзаца кода.

***Используйте комментарии для подготовки читателя кода к последующей информации*** Хорошие комментарии говорят человеку, читающему код, чего ожидать. Просмотра комментариев должно быть достаточно для получения хорошего представления о том, что делает код и где искать места выполнения отдельных операций. Из этого следует, что комментарии должны всегда предшествовать описываемому ими коду. На занятиях по программированию об этом говорят не всегда, но в отрасли разработки коммерческого ПО эта конвенция стала почти стандартом.

***Не размножайте комментарии без необходимости*** В избыточном комментировании ничего хорошего: если комментариев слишком много, они только скрывают смысл кода, который должны пояснять. Вместо того чтобы писать дополнительные комментарии, постарайтесь сделать читабельнее сам код.

***Документируйте сюрпризы*** Если что-то не следует из самого кода, отразите это в комментарии. Если вы использовали хитрую методику вместо простой для повышения производительности, укажите в комментариях, каким был бы простой подход, и оцените прирост производительности, достигаемый благодаря хитрости, например:

**Пример документирования сюрприза (C++)**

```
for ( element = 0; element < elementCount; element++ ) {
   // Для деления на 2 используется операция сдвига вправо.
   // Это сокращает время выполнения цикла на 75%.
   elementList[ element ] = elementList[ element ] >> 1;
}
```

Операция сдвига вправо в этом примере выбрана намеренно. Почти всем программистам известно, что в случае целых чисел сдвиг вправо функционально эквивалентен делению на 2.

Если это известно почти всем, зачем это документировать? Потому что целью данного кода является не сдвиг вправо, а деление на 2. Важно то, что программист использовал не ту методику, которая лучше всего соответствует цели. Кроме того, большинство компиляторов в любом случае заменяют целочисленное деление на 2 операцией сдвига вправо, из чего следует, что ухудшение ясности кода обычно не требуется. В данной ситуации комментарий говорит, что компилятор

не оптимизирует деление на 2, поэтому замена деления на сдвиг вправо существенно ускоряет выполнение кода. Благодаря комментарию программист, читающий код, тут же поймет мотивацию использования неочевидного подхода. Если бы комментария не было, тот же программист мог бы ворчать, что код усложнен без причины. Обычно такое ворчание оправданно, поэтому документировать исключения из правил важно.

***Избегайте сокращений*** Комментарии должны быть однозначны и не заставлять программистов расшифровывать сокращения. Никаких сокращений, кроме самых распространенных, в комментариях быть не должно. Обычно такой соблазн возникает только при использовании комментариев в концах строк. Это еще один аргумент против данного вида комментариев.

***Проведите различие между общими и детальными комментариями*** Иногда желательно провести различие между комментариями разных уровней, указав, что детальный комментарий является частью предыдущего, более общего комментария. Эту задачу можно решить несколькими способами. Вы можете подчеркивать общие комментарии и не подчеркивать детальные:

**Пример проведения различия между общим и детальными комментариями при помощи подчеркивания — не рекомендуется (C++)**

```
Общий комментарий подчеркивается.
// копирование всех строк таблицы,
// кроме строк, подлежащих удалению
//---------------------------------
Детальный комментарий, являющийся частью действия, описываемого общим комментарием, не
подчеркивается ни здесь...
// определение числа строк в таблице
...
...ни здесь.
// маркирование строк, подлежащих удалению
...
```

Слабость этого подхода в том, что он заставляет подчеркивать больше комментариев, чем хотелось бы. Подчеркивание какого-то комментария предполагает, что все следующие неподчеркнутые комментарии являются подчиненными по отношению к нему. Соответственно первый комментарий, не относящийся к подчеркнутому комментарию, также должен быть подчеркнут, и все начинается сызнова. Результат? Избыток подчеркивания или его несогласованность.

Эта тема имеет несколько вариаций, с которыми связана та же проблема. Если вы используете в общем комментарии только заглавные буквы, а в детальных — только строчные, вы просто заменяете избыток подчеркивания на избыток заглавных букв. Некоторые программисты проводят различие между общим и детальными комментариями при помощи регистра только первой буквы, но это различие слишком мало, и его легко не заметить.

Лучше предварять детальные комментарии многоточием:

**Пример проведения различия между общим и детальными комментариями при помощи многоточий (C++)**

Общий комментарий форматируется как обычно.

```
// копирование всех строк таблицы,
// кроме строк, подлежащих удалению
```

Детальный комментарий, являющийся частью действия, описываемого общим комментарием, предваряется многоточием здесь...

```
// ... определение числа строк в таблице
...
```

...и здесь.

```
// ... маркирование строк, подлежащих удалению
...
```

Другой подход, который часто оказывается самым лучшим, — вынесение кода, соответствующего общему комментарию, в отдельный метод. В идеале методы должны быть логически «плоскими»: все выполняемые в них действия должны относиться примерно к одному логическому уровню. Если в методе выполняются и общие, и детальные действия, метод не является плоским. В результате вынесения сложной группы действий в отдельный метод вы получите два логически ясных метода вместо одного логически скомканного.

Данное обсуждение общих и детальных комментариев не относится к коду с отступами, содержащемуся внутри циклов и условных операторов. В этих случаях часто имеется общий комментарий перед циклом и более детальные комментарии в коде с отступами. Логическая организация комментариев характеризуется отступами. Таким образом, вышесказанное относилось только к последовательным абзацам кода, когда полная операция охватывает несколько абзацев, а некоторые абзацы подчинены другим.

***Комментируйте все, что имеет отношение к ошибкам или недокументированным возможностям языка или среды*** Если в языке или среде есть ошибка, она, вероятно, недокументирована. Даже если она где-то описана, не помешает сделать это еще раз в коде. Недокументированная возможность по определению не описана нигде, и ее следует задокументировать в коде.

Допустим, вы обнаружили, что библиотечный метод *WriteData( data, numItems, blockSize )* работает неверно, если *blockSize* имеет значение *500*. Метод прекрасно обрабатывает значения *499*, *501* и все остальные, которые вы когда-либо пробовали, но имеет дефект, проявляющийся, только когда параметр *blockSize* равен *500*. Напишите перед вызовом *WriteData()*, почему вы создали специальный случай для этого значения параметра. Вот как это могло бы выглядеть:

**Пример документирования кода, предотвращающего ошибку (Java)**

```
blockSize = optimalBlockSize( numItems, sizePerItem );
```

```
/* Следующий код нужен потому, что метод WriteData() содержит ошибку,
проявляющуюся, только когда третий параметр равен 500.
Значение '500' ради ясности заменено на именованную константу.
*/
if ( blockSize == WRITEDATA_BROKEN_SIZE ) {
    blockSize = WRITEDATA_WORKAROUND_SIZE;
}
WriteData ( file, data, blockSize );
```

***Обосновывайте нарушения хорошего стиля программирования*** Если вы вынуждены нарушить хороший стиль программирования, объясните причину. Благодаря этому программисты, исполненные благих намерений, узнают, что попытка улучшения вашего кода может привести к нарушению его работы, и не станут изменять его. Объяснение ясно скажет, что вы знали, что делаете, а не допустили небрежность — улучшите свою репутацию, если есть такая возможность!

***Не комментируйте хитрый код — перепишите его*** Вот один комментарий из проекта, в котором я принимал участие:

**Пример комментирования хитрого кода (C++)**

```
// ОЧЕНЬ ВАЖНОЕ ЗАМЕЧАНИЕ:
// Конструктор этого класса принимает ссылку на объект UiPublication.
// Объект UiPublication НЕЛЬЗЯ УНИЧТОЖАТЬ раньше объекта DatabasePublication,
// иначе программу ожидает мученическая смерть.
```

Это хороший пример одного из самых распространенных и опасных заблуждений, согласно которому комментарии следует использовать для документирования особенно «хитрых» или «нестабильных» фрагментов кода. Данная идея обосновывается тем, что люди должны знать, когда им следует быть осторожными.

Это плохая идея.

Комментирование хитрого кода — как раз то, чего делать не следует. Комментарии не могут спасти сложный код. Как призывают Керниган и Плоджер, «не документируйте плохой код — перепишите его» (Kernighan and Plauger, 1978).

Исследования показали, что фрагменты исходного кода с большим числом комментариев обычно включали максимальное число дефектов и отнимали бóльшую долю ресурсов, уходивших на разработку ПО (Lind and Vairavan, 1989). Ученые предположили, что программисты склонны щедро комментировать сложный код.

Когда кто-то говорит: «Это по-настоящему *хитрый* код,» — я слышу: «Это по-настоящему *плохой* код». Если вам что-то кажется хитрым, для кого-то другого это окажется непонятным. Даже если что-то не кажется вам очень уж хитрым, другой человек, который не сталкивался с этим трюком, сочтет его очень замысловатым. Если вы спрашиваете себя: «Хитро ли это?» — это хитро. Всегда можно найти несложный вариант решения проблемы, поэтому перепишите код. Сделайте его несколько хорошим, чтобы нужда в комментариях вообще отпала, а затем прокомментируйте код, чтобы сделать его еще лучше.

Этот совет относится преимущественно к коду, который вы пишете впервые. Если вы сопровождаете программу и не имеете возможности переписывать плохой код, комментирование хитрых фрагментов — хороший подход.

## Комментирование объявлений данных

**Перекрестная ссылка** О форматировании данных см. подраздел «Размещение объявлений данных» раздела 31.5. Об эффективном использовании данных см. главы 10–13.

Комментарий объявления переменной описывает аспекты переменной, которые невозможно выразить в ее имени. Тщательно документировать данные важно: по крайней мере одна компания, изучавшая собственные методики, пришла к выводу, что комментировать данные даже важнее, чем процессы, в которых эти данные используются (SDC в Glass, 1982).

***Указывайте в комментариях единицы измерения численных величин*** Если число представляет длину, укажите единицы представления: дюймы, футы, метры или километры. Если речь идет о времени, поясните, в чем оно выражено: в секундах, прошедших с 1 января 1980 года, миллисекундах, прошедших с момента запуска программы, или как-то иначе. Если это координаты, напишите, что они представляют (ширину, долготу и высоту) и в чем они выражены (в радианах или градусах), укажите систему координат и т. д. Не предполагайте, что единицы измерения очевидны — для нового программиста они такими не будут. Для кого-то, кто работает над другой частью системы, они такими не будут. После значительного изменения программы они тоже очевидными не будут.

Единицы измерения часто следует указывать в именах переменных, а не в комментариях. Например, выражение вида *distanceToSurface = marsLanderAltitude* выглядит корректным, тогда как *distanceToSurfaceInMeters = marsLanderAltitudeInFeet* ясно указывает на ошибку.

**Перекрестная ссылка** Более эффективный способ документирования диапазонов допустимых значений переменных — использование утверждений в начале и в конце метода (см. раздел 8.2).

***Указывайте в комментариях диапазоны допустимых значений численных величин*** Если предполагается, что значение переменной должно попадать в некоторый диапазон, задокументируйте это. Одной из мощных возможностей языка Ada была возможность указания диапазона допустимых значений численной переменной. Если ваш язык не поддерживает такую возможность (а большинство языков ее не поддерживает), используйте для документирования диапазона ожидаемых значений комментарии. Например, если переменная представляет денежную сумму в долларах, укажите, что в вашем случае она должна находиться в пределах от 1 до 100 долларов. Если переменная представляет напряжение, напишите, что оно должно находиться в пределах от 105 В до 125 В.

***Комментируйте смысл закодированных значений*** Если ваш язык поддерживает перечисления (как C++ и Visual Basic), используйте их для выражения смысла закодированных значений. Если нет, указывайте смысл каждого значения в комментариях и представляйте каждое значение в форме именованной константы, а не литерала. Так, если переменная представляет виды электрического тока, закомментируйте тот факт, что *1* представляет переменный ток, *2* — постоянный, а *3* — неопределенный вид.

Вот пример, иллюстрирующий три предыдущих рекомендации: вся информация о диапазонах значений указана в комментариях:

### Пример грамотного документирования объявлений переменных (Visual Basic)

```
Dim cursorX As Integer  ' горизонтальная позиция курсора; диапазон: 1..MaxCols
Dim cursorY As Integer  ' вертикальная позиция курсора; диапазон: 1..MaxRows

Dim antennaLength As Long       ' длина антенны в метрах; диапазон: >= 2
Dim signalStrength As Integer   ' мощность сигнала в кВт; диапазон: >= 1

Dim characterCode As Integer       ' код символа ASCII; диапазон: 0..255
Dim characterAttribute As Integer ' 0=Обычный; 1=Курсив; 2=Жирный; 3=Жирный курсив
Dim characterSize As Integer       ' размер символа в точках; диапазон: 4..127
```

***Комментируйте ограничения входных данных*** Входные данные могут быть получены в виде входного параметра, прочитаны из файла или введены пользователем. Предыдущие советы относятся к входным параметров методов в той же степени, что и к другим видам данных. Убедитесь, что вы документируете ожидаемые и неожиданные значения. Комментарии — это один из способов документирования того, что метод никогда не должен принимать некоторые данные. Задокументировать диапазоны допустимых значений можно также, использовав утверждения, и тогда эффективность обнаружения ввода неверных данных заметно повысится.

***Документируйте флаги до уровня отдельных битов*** Если переменная используется как битовое поле, укажите смысл каждого бита:

**Перекрестная ссылка** Об именовании переменных-флагов см. подраздел «Именование переменных статуса» раздела 11.2.

### Пример документирования флагов до уровня битов (Visual Basic)

```
' Значения битов переменной statusFlags в порядке от самого старшего
' бита (MSB) до самого младшего бита (LSB):
' MSB   0     обнаружена ли ошибка?: 1=да, 0=нет
'       1-2   тип ошибки: 0=синтаксич., 1=предупреждение, 2=тяжелая, 3=фатальная
'       3     зарезервировано (следует обнулить)
'       4     статус принтера: 1=готов, 0=не готов
'       ...
'       14    не используется (следует обнулить)
' LSB   15-32 не используются (следует обнулить)
Dim statusFlags As Integer
```

Если бы мы писали этот пример на C++, следовало бы использовать синтаксис битовых полей — тогда значения полей были бы самодокументирующимися.

***Включайте в комментарии, относящиеся к переменной, имя переменной*** Если какие-то комментарии относятся к конкретной переменной, убедитесь, что вы обновляете их вместе с переменной. Помочь в этом может использование в комментариях имени переменной. Благодаря этому при поиске переменной в коде вы найдете не только ее, но и связанные с ней комментарии.

**Перекрестная ссылка** О глобальных данных см. раздел 13.3.

***Документируйте глобальные данные*** Если вы используете глобальные данные, комментируйте их в местах объявления. В этом комментарии следует указать роль данных и причину, по которой они должны быть глобальными. Используя их, каждый раз поясняйте, что данные глобальны. Первое средство подчеркивания глобального статуса переменной — конвенция именования. Если такую конвенцию именования вы не приняли, комментарии могут восполнить этот пробел.

## Комментирование управляющих структур

**Перекрестная ссылка** Об управляющих структурах см. также разделы 31.3 и 31.4 и главы с 14 по 19.

Обычно самое подходящее место для комментирования управляющей структуры — предшествующие ей строки. Если это оператор *if* или блок *case*, вы можете пояснить в комментарии условие и результаты. Если это цикл, можно указать его цель.

**Пример комментирования цели управляющей структуры (C++)**

Цель цикла.

```
// копирование символов входного поля до запятой
while ( ( *inputString != ',' ) && ( *inputString != END_OF_STRING ) ) {
   *field = *inputString;
   field++;
   inputString++;
```

Комментарий конца цикла (полезен в случае длинных вложенных циклов, хотя необходимость такого комментария указывает на чрезмерную сложность кода).

```
} // while - копирование символов входного поля

*field = END_OF_STRING;

if ( *inputString != END_OF_STRING ) {
```

Цель цикла. Положение комментария ясно говорит, что переменная *inputString* устанавливается с целью использования в цикле.

```
   // пропуск запятой и пробелов для нахождения следующего входного поля
   inputString++;
   while ( ( *inputString == ' ' ) && ( *inputString != END_OF_STRING ) ) {
      inputString++;
   }
} // if - конец строки
```

Опираясь на этот пример, можно дать несколько советов по комментированию управляющих структур.

***Пишите комментарий перед каждым оператором* if, *блоком* case, *циклом или группой операторов*** Эти конструкции часто требуют объяснения, а место перед ними лучше всего подходит для этого. Используйте комментарии для пояснения цели управляющих структур.

***Комментируйте завершение каждой управляющей структуры*** Используйте комментарий для объяснения того, что именно завершилось, например:

```
} // for clientIndex – обработка записей всех клиентов
```

Комментарии особенно полезно применять для обозначения концов длинных циклов и для пояснения их вложенности. Вот пример комментариев, поясняющих концы циклов:

**Пример использования комментариев, иллюстрирующих вложенность (Java)**

```
for ( tableIndex = 0; tableIndex < tableCount; tableIndex++ ) {
   while ( recordIndex < recordCount ) {
      if ( !IllegalRecordNumber( recordIndex ) ) {
         ...
```

Эти комментарии сообщают, какая управляющая структура завершается.

```
      } // if
   } // while
} // for
```

Эта методика комментирования дополняет визуальную информацию о логической структуре кода, предоставляемую отступами кода. Если циклы коротки, а вложенности нет, эта методика не нужна, однако в случае глубокой вложенности или длинных циклов она окупается.

***Рассматривайте комментарии в концах циклов как предупреждения о сложности кода*** Если цикл настолько сложен, что в его конце нужен комментарий, подумайте, не упростить ли цикл. Это же правило относится к сложным операторам *if* и блокам *case*.

Комментарии в концах циклов сообщают полезную информацию о логической структуре кода, но писать и поддерживать их иногда утомительно. Зачастую лучший способ предотвратить эту нудную работу — переписать код, который в силу своей сложности требует подобной документации.

## Комментирование методов

С комментариями методов связан одни из самых худших советов, который дается в типичных учебниках по программированию. Многие авторы советуют независимо от размера или сложности метода нагромождать перед его началом целые информационные баррикады:

**Перекрестная ссылка** О форматировании методов см. раздел 31.7. О создании высококачественных методов см. главу 7.

**Пример монолитного натуралистичного пролога метода (Visual Basic)**

```
'*********************************************************************
' Имя: CopyString
'
' Цель:             Этот метод копирует строку-источник (источник)
'                   в строку-приемник (приемник).
'
```

```
' Алгоритм:          Метод получает длину "источника", после чего поочередно
'                    копирует каждый символ в "приемник". В качестве индекса
'                    массивов "источника" и "приемника" используется индекс
'                    цикла. Индекс цикла/массивов увеличивается после
'                    копирования каждого символа.
'
' Входные данные:  input    Копируемая строка
'
' Выходные данные: output   Строка, содержащая копию строки "input"
'
' Предположения об интерфейсе: нет
'
' История изменений: нет
'
' Автор:            Дуайт К. Кодер
' Дата создания:    01.10.04
' Телефон:          (555) 222-2255
' SSN:              111-22-3333
' Цвет глаз:        Зеленый
' Девичья фамилия: -
' Группа крови:     AB-
' Девичья фамилия матери: -
' Любимый автомобиль: "Понтиак Ацтек"
' Персонализированный номер автомобиля: "Tek-ie"
'*****************************************************************************
```

Это глупо. Метод *CopyString* тривиален и скорее всего включает не более пяти строк кода. Комментарий совершенно не соответствует объему метода. Цель и алгоритм метода высосаны из пальца, потому что трудно описать что-то настолько простое, как *CopyString*, на уровне детальности между «копированием строки» и самим кодом. Предположения об интерфейсе и история изменений также бесполезны — эти комментарии только занимают место в листинге. Фамилия автора дополнена избыточными данными, которые можно легко найти в системе управления ревизиями. Заставлять указывать всю эту информацию перед каждым методом — значит подталкивать программистов к написанию неточных комментариев и затруднять сопровождение программы. Эти лишние усилия не окупятся никогда.

Другая проблема с тяжеловесными заголовками методов состоит в том, что они мешают факторизовать код: затраты, связанные с созданием нового метода, так велики, что программисты будут стремиться создавать меньше методов. Конвенции кодирования должны поощрять применение хороших методик — тяжеловесные заголовки методов поощряют их игнорировать.

А теперь несколько советов по комментированию методов.

***Располагайте комментарии близко к описываемому ими коду*** Одна из причин того, что пролог метода не должен содержать объемной документации, в том, что при этом комментарии далеки от описываемых ими частей метода. Если комментарии далеки от кода, вероятность того, что их не будут изменять вместе с кодом при сопровождении, повышается. Смысл комментариев и кода начинает расходиться, и внезапно комментарии становятся никчемными. Поэтому соблюдайте

Принцип Близости и располагайте комментарии как можно ближе к описываемому ими коду. Тогда их будут поддерживать, а они сохранят свою полезность.

Несколько компонентов, которые по мере необходимости следует включать в прологи методов, описаны ниже. Ради удобства создавайте стандартизованные прологи. Не думайте, что перед каждым методом нужно указывать всю информацию. Включайте действительно важные элементы и опускайте остальные.

***Описывайте каждый метод одним-двумя предложениями перед началом метода*** Если вы не можете описать метод одним или двумя краткими предложениями, вам, вероятно, следует лучше обдумать роль метода. Если краткое описание придумать трудно, значит, проект метода не так хорош. Попробуйте перепроектировать метод. Краткое резюмирующее предложение должно присутствовать почти во всех методах, кроме простых методов доступа *Get* и *Set*.

**Перекрестная ссылка** Удачный выбор имени метода — важнейший аспект документирования методов (см. раздел 7.3).

***Документируйте параметры в местах их объявления*** Самый простой способ документирования входных и выходных переменных — написать комментарии после их объявления:

**Пример документирования входных и выходных данных в местах их объявления — хороший подход (Java)**

```
public void InsertionSort(
   int[] dataToSort, // массив элементов, подлежащих сортировке
   int firstElement, // индекс первого сортируемого элемента (>=0)
   int lastElement   // индекс последнего сортируемого элемента (<= MAX_ELEMENTS)
)
```

Этот совет — уместное исключение из правила, предписывающего избегать комментариев в концах строк; такие комментарии крайне полезны при документировании входных и выходных параметров. Кроме того, данный случай хорошо иллюстрирует полезность выравнивания параметров методов с помощью стандартных отступов, а не отступов в конце строк — при отступах в конце строк у вас просто не останется места для выразительных комментариев. В этом примере комментариям тесно даже при стандартных отступах. Этот пример также показывает, что комментарии — не единственный способ документирования. Если имена переменных достаточно хороши, их можно не комментировать. Наконец, необходимость документирования входных и выходных переменных — хорошая причина избегать глобальных данных. Где вы будете их документировать? Вероятно, документировать глобальные данные следует в огромном прологе. Для этого нужно выполнить большую работу, что на практике, увы, обычно означает, что глобальные данные не документируются. Это очень плохо, так как глобальные данные нужно документировать не менее тщательно, чем все остальное.

**Перекрестная ссылка** О комментариях в концах строк см. выше подраздел «Комментарии в концах строк и связанные с ними проблемы» этого раздела.

***Используйте утилиты документирования кода, такие как Javadoc*** Если бы предыдущий пример нужно было на самом деле написать на Java, вы могли бы адаптировать код к Javadoc — утилите извлечения документации Java. Тогда

смысл совета «документируйте параметры в местах их объявления» несколько изменился бы, что привело бы к получению такого кода:

**Пример документирования входных и выходных параметров для использования Javadoc (Java)**

```
/**
 * ... <описание метода> ...
 *
 * @param dataToSort  массив элементов, подлежащих сортировке
 * @param firstElement индекс первого сортируемого элемента (>=0)
 * @param lastElement  индекс последнего сортируемого элемента (<= MAX_ELEMENTS)
 */
public void InsertionSort(
   int[] dataToSort,
   int firstElement,
   int lastElement
)
```

При использовании инструмента вроде Javadoc выгода от специфической адаптации кода к последующему извлечению документации из него перевешивает риск, связанный с отделением описания параметров от их объявлений. Если среда не поддерживает извлечение документации, комментарии обычно лучше располагать ближе к именам параметров во избежание несогласованного редактирования кода и комментариев, а также дублирования самих имен.

***Проведите различие между входными и выходными данными*** Знать, какие данные являются входными, а какие выходными, полезно. При работе с Visual Basic определить это относительно легко, потому что выходным данным предшествует ключевое слово *ByRef*, а входным — *ByVal*. Если ваш язык не поддерживает такую дифференциацию автоматически, выразите это при помощи комментариев. Вот пример:

**Перекрестная ссылка** Порядок этих параметров соответствует стандартному порядку, принятому для методов C++, но конфликтует с более общими методиками (см. подраздел «Передавайте параметры в порядке „входные значения — изменяемые значения — выходные значения“» раздела 7.5). Об использовании конвенции именования для проведения различия между входными и выходными данными см. раздел 11.4.

**Пример проведения различия между входными и выходными данными (C++)**

```
void StringCopy(
   char *target,        // out: строка-приемник
   const char *source   // in: строка-источник
)
...
```

Объявления методов C++ немного хитры, потому что иногда звездочка (*) говорит о том, что аргумент является выходным параметром, но очень часто это просто означает, что с переменной легче работать как с указателем. Как правило, лучше идентифицировать входные и выходные параметры явно.

Если ваши методы достаточно коротки и вы поддерживаете ясное различие между входными и выходными данными, документировать статус данных (входной или выходной), наверное, не нужно. Однако если метод более объемен, указание статуса поможет всем, кто будет читать код метода.

***Документируйте выраженные в интерфейсе предположения*** Этот совет можно рассматривать как подмножество других рекомендаций по поводу комментирования. Если вы сделали какие-либо предположения о состоянии получаемых вами переменных (о допустимых и недопустимых значениях, о том, что массивы должны быть отсортированы, что данные-члены должны быть инициализированы или должны иметь только допустимые значения и т. д.), укажите это или в прологе метода, или в месте объявления данных. Этот вид документации должен присутствовать почти в каждом методе.

**Перекрестная ссылка** Другие соображения относительно интерфейсов методов см. в разделе 7.5. О документировании предположений с помощью утверждений см. подраздел «Используйте утверждения для документирования и проверки предусловий и постусловий» раздела 8.2.

Убедитесь, что вы задокументировали используемые глобальные данные. Глобальная переменная — такая же часть интерфейса метода, как и все остальное, но при этом она более опасна, так как иногда не кажется частью интерфейса.

Если вы пишете метод и понимаете, что делаете предположение об интерфейсе, немедленно задокументируйте его.

***Комментируйте ограничения методов*** Если метод возвращает число, укажите его точность. Если в некоторых условиях результаты вычислений неопределенны, задокументируйте эти условия. Если на случай проблем вы реализовали в методе поведение по умолчанию, задокументируйте это поведение. Если предполагается, что метод будет работать только с массивами или таблицами определенных размеров, укажите это. Если вам известны изменения кода, которые могут привести к нарушению работы метода, задокументируйте их. Если при разработке метода возникают сбои, задокументируйте и их.

***Документируйте глобальные результаты выполнения метода*** Если метод изменяет глобальные данные, опишите, что именно он делает с ними. Как было сказано в разделе 13.3, изменение глобальных данных минимум на порядок опаснее, чем их простое чтение, поэтому изменять их нужно осторожно, и частью этой осторожности должна быть ясная документация. Если документирование становится слишком обременительным, следуйте обычному подходу — перепишите код, чтобы глобальных данных стало меньше.

***Документируйте источники используемых алгоритмов*** Если вы использовали в коде алгоритм из книги или журнала, укажите источник и номер страницы. Если вы разработали алгоритм сами, укажите, где читатель может найти его описание.

***Используйте комментарии для маркирования частей программы*** Некоторые программисты отмечают комментариями те или иные части кода, чтобы их было легче искать. Одна такая методика, принятая в C++ и Java, предполагает, что начало каждого метода отмечается комментарием, начинающимся с символов:

```
/**
```

Это позволяет перескакивать с метода на метод или путем поиска символов /**, или автоматически, если такую возможность поддерживает редактор.

Похожая методика заключается в использовании разных маркеров для разных видов комментариев в зависимости от того, что они описывают. Так, в C++ вы могли бы

использовать комментарии вида *@keyword*, где *keyword* — код, определяющий тип комментария. Комментарий *@param* мог бы описывать параметр метода, *@version* — версию файла, *@throws* — исключения, генерируемые методом, и т. д. Эта методика позволяет применять инструменты для извлечения разных видов информации из файлов исходного кода. Так, для получения документации обо всех исключениях, генерируемых методами программы, вы могли бы выполнить поиск комментариев *@throws*.

http://cc2e.com/3259

Эта конвенция C++ основана на общепринятой конвенции Javadoc документирования интерфейсов в программах Java (*java.sun.com/j2se/javadoc/*). Для других языков вы можете определить собственные соглашения.

## Комментирование классов, файлов и программ

**Перекрестная ссылка** О форматировании классов, файлов и программ см. раздел 31.8. Об использовании классов см. главу 6.

Классы, файлы и программы объединяет то, что все они включают много методов: файл или класс должен содержать набор родственных методов, программа содержит все методы. В каждом этом случае цель документирования в том, чтобы предоставить выразительную высокоуровневую информацию о содержании файла, класса или программы.

### Общие принципы документирования классов

Создайте перед каждым классом блочный комментарий, описывающий общие атрибуты класса. Учтите при этом следующее.

***Опишите подход к проектированию класса*** Обзорные комментарии, предоставляющие информацию, которую нельзя быстро извлечь из самого кода, особенно полезны. Опишите философию проектирования класса, общий подход к его проектированию, альтернативные варианты проектирования, которые были рассмотрены и отброшены, и т. д.

***Опишите ограничения класса, предположения о его использовании и т. д.*** Как и в случае методов, убедитесь, что вы описали все ограничения, вытекающие из проекта класса. Опишите также предположения о входных и выходных данных, аспекты ответственности за обработку ошибок, глобальные эффекты, источники алгоритмов и т. д.

***Прокомментируйте интерфейс класса*** Может ли другой программист разобраться в использовании класса, не изучив его реализацию? Если нет, инкапсуляция класса подвергается серьезной опасности. Интерфейс класса должен содержать информацию, нужную для использования класса. Конвенция Javadoc требует документирования хотя бы каждого параметра и каждого возвращаемого значения (Sun Microsystems, 2000). Это надо сделать для каждого метода, предоставляемого каждым классом (Bloch, 2001).

***Не документируйте в интерфейсе класса детали реализации*** Главное правило инкапсуляции гласит, что предоставлять информацию нужно только в соответствии с принципом необходимого знания: если у вас есть хоть какие-то сомнения в том, предоставлять ли информацию, оставьте ее скрытой. Таким образом, файл интерфейса класса должен содержать информацию, нужную для использования класса, но не для реализации или сопровождения класса.

### Общие принципы документирования файлов

Создайте в начале файла блочный комментарий, описывающий содержание файла.

***Опишите назначение и содержание каждого файла*** В заголовочном комментарии к файлу следует описать классы или методы, содержащиеся в файле. Если все методы программы находятся в одном файле, назначение файла очевидно: он содержит весь код программы. Если файл содержит один класс, назначение файла также очевидно.

Если же файл включает более одного класса, объясните, почему классы объединены в одном файле.

Если программа разделена на несколько файлов исходного кода не ради модульности, а по иной причине, хорошее описание назначения файла окажется еще полезнее для программиста, которому придется изменять программу. Подумайте, поможет ли заголовочный комментарий к файлу определить, содержится ли в этом файле метод, выполняющий конкретное действие?

***Укажите в блочном комментарии свои имя/фамилию, адрес электронной почты и номер телефона*** При работе над крупными проектами большое значение имеют такие факторы, как авторство и ответственность за конкретные фрагменты исходного кода. В небольших проектах (реализуемых с участием менее 10 человек) годятся методики совместной разработки, при которых все члены группы в равной степени отвечают за все разделы кода. Разработка более крупных систем требует, чтобы программисты специализировались на разных областях кода, что делает совместное владение кодом нереальным.

В этом случае сведения об авторстве нужно включать в листинг. Они позволят другим программистам, работающим над кодом, узнать кое-что о стиле программирования и связаться с автором кода, если им понадобится помощь. В зависимости от того, над чем вы работаете (над отдельными методами, классами или программами), сведения об авторстве следует включать на уровне методов, классов или программ.

***Включите в файл тег версии*** Многие инструменты управления версиями вставляют сведения о версии в файл. Например, система CVS автоматически расширяет символы:

```
// $Id$
```

в комментарий:

```
// $Id: ClassName.java,v 1.1 2004/02/05 00:36:43 ismene Exp $
```

Это позволяет поддерживать актуальную информацию о версиях кода, не заставляя разработчиков прилагать никаких усилий, кроме вставки первоначального комментария *$Id$.*

***Включите в блочный комментарий юридическую информацию*** Многие компании любят включать в свои программы уведомления об авторских правах, конфиденциальности и другую юридическую информацию. Если вы работаете в такой компании, включите в код строку, похожую на указанную ниже. Узнайте у юриста своей компании, какую информацию включать в файлы, если это вообще следует делать.

### Пример уведомления об авторских правах (Java)

```
// (c) Copyright 1993-2004 Steven C. McConnell. All Rights Reserved.
...
```

***Присвойте файлу имя, характеризующее его содержание*** Как правило, имя файла должно быть тесно связано с именем открытого класса, содержащегося в файле. Так, если класс называется *Employee*, файл следует назвать *Employee.cpp*. Некоторые языки, например Java, требуют, чтобы имя файла соответствовало имени класса.

### Книжная парадигма документирования программ

**Дополнительные сведения** Это обсуждение основано на работах «The Book Paradigm for Improved Maintenance» (Oman and Cook, 1990a) и «Typographic Style Is More Than Cosmetic» (Oman and Cook, 1990b). Похожий подробный анализ см. в «Human Factors and Typography for More Readable Programs» (Baecker and Marcus, 1990).

Самые опытные программисты соглашаются с тем, что методики документирования, описанные в предыдущем разделе, полезны. Достоверных научных данных о полезности каждой из методик все еще мало, однако при объединении методик их эффективность не вызывает сомнений.

В 1990 году Пол Оман и Кертис Кук опубликовали результаты двух исследований «Книжной парадигмы (Book Paradigm)» документирования (Oman and Cook, 1990a, 1990b). Они искали стиль кодирования, который поддерживал бы несколько разных стилей чтения кода. Одной целью была поддержка нисходящего, восходящего и сфокусированного поиска. Другой целью было разделение кода на фрагменты, с которыми было бы легче работать, чем с длинным листингом однородного кода. Оман и Кук хотели, чтобы стиль предоставлял информацию и о высокоуровневой, и о низкоуровневой организации кода.

Они обнаружили, что этих целей можно достичь, если рассматривать код как специфический вид книги и форматировать его соответствующим образом. Книжная парадигма предполагает, что код и документация организуются в несколько компонентов, похожих на компоненты книги и помогающих программистам получить высокоуровневое представление о программе.

«Предисловие» — это группа вводных комментариев, таких как комментарии, обычно встречающиеся в начале файла. Они аналогичны предисловию книги и предоставляют программисту обзорную информацию о программе.

«Содержание» определяет высокоуровневые файлы, классы и методы (главы), которые могут быть указаны в форме списка, как главы обычной книги, или графически — в виде структурной схемы.

«Разделы» соответствуют отдельным частям методов, например, объявлениям методов, объявлениям данных и исполняемым операторам.

«Перекрестные ссылки» — это карты перекрестных ссылок в коде, включающие номера строк.

Низкоуровневые методики, опираясь на которые Оман и Кук воспользовались преимуществами сходств между книгой и листингом, похожи на методики, описанные в главе 31 и этой главе.

Результат использования этих методик организации кода таков: когда Оман и Кук попросили группу опытных программистов выполнить одну задачу на сопровождение программы, среднее время выполнения этой задачи в случае программы из 1000 строк составило только около трех четвертей от времени, потребовавшегося для решения той же задачи при традиционной организации исходного кода (Oman and Cook, 1990b). Более того, при сопровождении кода, задокументированного в соответствии с Книжной парадигмой, программисты получили в среднем на 20% более высокие оценки, чем при сопровождении кода, задокументированного традиционным образом. Оман и Кук пришли к выводу, что, обращая внимание на принципы структурирования книг, можно улучшить понятность кода на 10–20%. В исследовании, проведенном в Университете Торонто, были получены похожие результаты (Baecker and Marcus, 1990).

Книжная парадигма подчеркивает важность создания документации, объясняющей и высокоуровневую, и низкоуровневую организацию программы.

## 32.6. Стандарты IEEE

Ценными источниками сведений о документации, не относящейся к уровню исходного кода, являются стандарты разработки ПО, принятые Институтом инженеров по электротехнике и электронике (Institute for Electric and Electrical Engineers, IEEE). Стандарты IEEE разрабатываются группами профессионалов и ученых — экспертов в конкретной области. Каждый стандарт содержит резюме описываемой стандартом области и обычно включает обзор документации, уместной для данной области.

В разработке стандартов принимают участие несколько национальных и международных организаций. Ведущая роль в определении стандартов разработки ПО принадлежит группе IEEE. Некоторые стандарты совместно приняты Международной организацией по стандартизации (International Standards Organization, ISO), Альянсом электронной промышленности (Electronic Industries Alliance, EIA) и Международным инженерным консорциумом (International Engineering Consortium, IEC).

Названия стандартов включают номер стандарта, год его принятия и само название. Так, «IEEE/EIA Std 12207-1997, информационные технологии — процессы жизненного цикла ПО» — это стандарт номер 12207.2, принятый в 1997 году IEEE и EIA.

Ниже я указал некоторые национальные и международные стандарты, в наибольшей степени относящиеся к проектам разработки ПО.

Стандартом верхнего уровня является международный стандарт «ISO/IEC Std 12207, Information Technology — Software Life Cycle Processes», определяющий схему разработки программных проектов и управления ими. В США этот стандарт был принят как «IEEE/EIA Std 12207, Information Technology—Software Life Cycle Processes».

http://cc2e.com/3266

### Стандарты разработки ПО

*IEEE Std 830-1998* — рекомендуемая методика составления спецификаций требований к ПО

http://cc2e.com/3273

*IEEE Std 1233-1998* — руководство по разработке спецификаций требований к системе

*IEEE Std 1016-1998* — рекомендуемая методика описания проекта ПО

*IEEE Std 828-1998* — стандарт планов управления конфигурацией ПО

*IEEE Std 1063-2001* — стандарт пользовательской документации

*IEEE Std 1219-1998* — стандарт сопровождения ПО

## Стандарты контроля качества ПО

http://cc2e.com/3280

*IEEE Std 730-2002* — стандарт планирования контроля качества ПО

*IEEE Std 1028-1997* — стандарт обзоров ПО

*IEEE Std 1008-1987 (R1993)* — стандарт блочного тестирования ПО

*IEEE Std 829-1998* — стандарт документирования тестов ПО

*IEEE Std 1061-1998* — стандарт методологии метрик качества ПО

## Стандарты управления

http://cc2e.com/3287

*IEEE Std 1058-1998* — стандарт планов управления проектами разработки ПО

*IEEE Std 1074-1997* — стандарт разработки процессов жизненного цикла ПО

*IEEE Std 1045-1992* — стандарт метрик продуктивности ПО

*IEEE Std 1062-1998* — рекомендуемая методика приобретения ПО

*IEEE Std 1540-2001* — стандарт процессов жизненного цикла ПО — управление риском

*IEEE Std 1490-1998* — руководство (заимствование стандарта PMI) к своду знаний по управлению проектами (PMBOK)

## Обзор стандартов

http://cc2e.com/3294

Обзоры стандартов можно найти в двух источниках, указанных ниже.

http://cc2e.com/3201

*IEEE Software Engineering Standards Collection, 2003 Edition*. New York, NY: Institute of Electrical and Electronics Engineers (IEEE). Этот всесторонний труд содержит 40 самых новых стандартов разработки ПО ANSI/IEEE на 2003 год. Каждый стандарт включает план документа, описание каждого компонента плана и обоснование этого компонента. Этот документ включает стандарты планов контроля качества, планов управления конфигурациями, документов тестирования, спецификаций требований, планов верификации и проверки, описаний проектирования, планов управления проектами и пользовательской документации. Данная книга представляет собой квинтэссенцию опыта сотен лучших специалистов в своих областях и была бы выгодным приобретением практически при любой цене. Некоторые из стандартов доступны и отдельно. Все стандарты можно заказать у организа-

ции IEEE Computer Society из города Лос-Аламитос, шт. Калифорния, и на сайте *www.computer.org/cspress*.

Moore, James W. *Software Engineering Standards: A User's Road Map*. Los Alamitos, CA: IEEE Computer Society Press, 1997. Мур приводит обзор стандартов IEEE разработки ПО.

## Дополнительные ресурсы

Кроме стандартов IEEE, есть и другие источники информации о документировании программ.

http://cc2e.com/3208

Spinellis, Diomidis. *Code Reading: The Open Source Perspective*. Boston, MA: Addison-Wesley, 2003. Эта книга является прагматичным исследованием методик чтения кода. В ней вы найдете советы по поиску нужного кода и чтению больших объемов кода, сведения об инструментах, облегчающих чтение кода, и массу полезных рекомендаций.

> Интересно, сколько великих писателей никогда не читали других авторов, сколько великих художников никогда не изучали произведения других мастеров, сколько высококвалифицированных хирургов никогда не учились, стоя за плечом коллеги… И все же именно этого мы ожидаем от программистов.
>
> *Дэйв Томас (Dave Thomas)*

*SourceForge.net*. Уже много лет обучению разработке ПО препятствует проблема поиска реальных примеров готового кода, которые можно было бы обсудить со студентами. Многие люди быстрее всего учатся на реальных примерах, однако большинство реальных программ является собственностью создавших их компаний. Благодаря Интернету и ПО с открытым исходным кодом эта ситуация улучшилась. Web-сайт Source Forge содержит код тысяч программ, написанных на C, C++, Java, Visual Basic, PHP, Perl, Python и других языках, причем весь этот код вы можете загрузить из сети совершенно бесплатно. Вы можете проанализировать реальные примеры, гораздо более объемные, чем примеры, приведенные в этой книге. Для начинающих программистов, не имевших ранее дела с объемными примерами готового кода, этот Web-сайт окажется особенно полезен как источник и хороших, и плохих методик кодирования.

http://cc2e.com/3215

Sun Microsystems. «How to Write Doc Comments for the Javadoc Tool», 2000. В этой статье (*http://java.sun.com/j2se/javadoc/writingdoccomments/*) описывается применение инструмента Javadoc для документирования программ Java. Она включает детальное описание разметки комментариев с использованием нотации стиля *@tag*, а также многие конкретные советы по написанию самих комментариев. Конвенции Javadoc являются, наверное, наиболее развитыми из нынешних стандартов документирования на уровне кода.

http://cc2e.com/3222

Ниже указаны источники информации о других аспектах документирования ПО.

McConnell, Steve. *Software Project Survival Guide*. Redmond, WA: Microsoft Press, 1998. В этой книге описывается документация, необходимая в критически важных проектах среднего размера. На Web-сайте книги вы можете найти много шаблонов документов.

http://cc2e.com/3229

*www.construx.com*. Этот Web-сайт (Web-сайт моей компании) содержит многочисленные шаблоны документов, описания конвенций кодирования и другие ресурсы, связанные со всеми аспектами разработки ПО, включая документирование ПО.

http://cc2e.com/3236

Post, Ed. «Real Programmers Don't Use Pascal», *Datamation*, July 1983, pp. 263–265. В этой ироничной статье автор тоскует по «старым добрым временам» программирования на Фортране, когда программисты могли не волноваться о таких неприятных вопросах, как читабельность кода.

http://cc2e.com/3243

### Контрольный список: хорошие методики комментирования

**Общие аспекты**

- ❑ Может ли программист, взглянувший на код, сразу понять его?
- ❑ Объясняют ли комментарии цель кода или резюмируют ли они выполняемые в коде действия вместо того, чтобы просто повторять код?
- ❑ Используете ли вы процесс программирования с псевдокодом для сокращения времени комментирования?
- ❑ Переписали ли вы хитрый код вместо того, чтобы комментировать его?
- ❑ Актуальны ли комментарии?
- ❑ Ясны ли комментарии? Корректны ли они?
- ❑ Позволяет ли стиль комментирования с легкостью изменять комментарии?

**Операторы и абзацы**

- ❑ Избегаете ли вы комментариев в концах строк?
- ❑ Стремитесь ли вы отвечать в комментариях на вопрос «почему», а не «как»?
- ❑ Готовят ли комментарии читателя к последующему коду?
- ❑ Каждый ли комментарий важен? Были ли удалены или улучшены избыточные, посторонние и неуместные комментарии?
- ❑ Документируете ли вы сюрпризы?
- ❑ Избегаете ли вы сокращений?
- ❑ Ясно ли различие между общими и детальными комментариями?
- ❑ Комментируете ли вы недокументированные возможности или код, предотвращающий ошибки языка или среды?

**Объявления данных**

- ❑ Указываете ли вы единицы измерения данных в местах объявления данных?
- ❑ Указываете ли вы диапазоны допустимых значений численных величин?
- ❑ Комментируете ли вы смысл закодированных значений?
- ❑ Комментируете ли вы ограничения входных данных?
- ❑ Документируете ли вы флаги до уровня отдельных битов?
- ❑ Комментируете ли вы каждую глобальную переменную в месте ее объявления?
- ❑ Поясняете ли вы при каждом использовании глобальной переменной ее глобальный характер при помощи конвенции именования, комментария или обоих способов?
- ❑ Заменили ли вы магические числа на именованные константы или переменные, вместо того чтобы ограничиться простым документированием?

**Управляющие структуры**

- ❑ Комментируете ли вы каждый управляющий оператор?
- ❑ Комментируете ли вы концы длинных или сложных управляющих структур или упрощаете ли вы эти структуры по мере возможности, чтобы они не требовали комментариев?

**Методы**

- ❑ Комментируете ли вы назначение каждого метода?
- ❑ Указываете ли вы в комментариях к методам в случае надобности другую информацию, такую как входные и выходные данные, предположения об интерфейсе, ограничения, исправленные ошибки, глобальные эффекты и источники алгоритмов?

**Файлы, классы и программы**

- ❑ Включает ли программа краткий документ, подобный документу Книжной парадигмы, предоставляющий общую информацию об организации программы?
- ❑ Описали ли вы назначение каждого файла?
- ❑ Указали ли вы в листинге фамилию и имя автора, адрес электронной почты и номер телефона?

# Ключевые моменты

- Вопрос о том, стоит ли комментировать код, вполне обоснован. При плохом выполнении комментирование является пустой тратой времени и иногда причиняет вред. При грамотном применении комментирование полезно.
- Основная часть критически важной информации о программе должна содержаться в исходном коде. На протяжении жизни программы актуальности исходного кода уделяется больше внимания, чем актуальности любого другого ресурса, поэтому важную информацию полезно интегрировать в код.
- Хороший код сам является самой лучшей документацией. Если код настолько плох, что требует объемных комментариев, попытайтесь сначала улучшить его.
- Комментарии должны сообщать о коде что-то такое, что он не может сообщить сам — на уровне резюме или уровне цели.
- Некоторые стили комментирования заставляют выполнять много нудной канцелярской работы. Используйте стиль, который было бы легко поддерживать.

ГЛАВА 33

# Личность

http://cc2e.com/3313

**Содержание**



**Связаные темы**

- Мастерство разработки ПО: глава 34
- Сложность: разделы 5.2 и 19.6

С момента публикации в 1965 году знаменитой статьи Эдсгера Дейкстры «Programming Considered as a Human Activity» («Программирование как человеческая деятельность») личности программистов привлекают самое пристальное внимание ученых. Названия книг «Психология конструирования мостов» и «Экспериментальные исследования поведения юристов» могли бы показаться абсурдными, однако работы «Психология программирования» («The Psychology of Computer Programming»), «Экспериментальные исследования поведения программистов» («Exploratory Experiments in Programmer Behavior») и т. п. уже стали классическими.

В какой бы области ни работали инженеры, они должны знать возможности применяемых инструментов и материалов. Инженеры-электрики знают проводимость разных металлов и сотни способов использования вольтметра. Инженерам-проектировщикам строительных конструкций известны параметры прочности дерева, бетона и стали.

Если вы разрабатываете ПО, вашим основным строительным материалом является интеллект, а главным инструментом — *вы сами*. Вместо того чтобы спроектировать структуру до последних деталей и передать чертежи конструкторам, вы проектируете фрагмент ПО до последних деталей, и на этом работа над ним завершается. По сути все программирование состоит в создании воздушных замков — это умственная деятельность почти что в самом чистом виде. Соответственно, когда разработчики ПО изучают существенные свойства своих инструментов и материалов, они изучают людей: интеллект, характер и другие атрибуты, не столь осязаемые, как дерево, бетон и сталь.

Если вам нужны конкретные советы, эта глава может показаться вам чересчур абстрактной, чтобы быть полезной. Поэтому прочтите следующий раздел и решите, следует ли ее пропустить.

## 33.1. Причем тут характер?

Крайняя замкнутость программирования придает свойствам личности программиста особую значимость. Вы сами знаете, как сложно поддерживать сосредоточенность на протяжении восьми часов рабочего дня. Наверное, вы можете вспомнить день или месяц, прошедший впустую из-за чрезмерной концентрации в предыдущий день или месяц. Наверняка бывали дни, когда вы продуктивно работали с 8 часов утра до 2 ночи, после чего чувствовали, что пора отдохнуть. Но вы продолжали работать до 5 часов утра и тратили остальные дни недели на исправление того, что написали ночью.

Программирование плохо поддается контролю особенно потому, что никто на самом деле не знает, над чем вы работаете. У всех нас были проекты, при реализации которых мы проводили 80% времени над небольшим фрагментом, казавшимся интересным и тратили 20% времени на создание остальных 80% программы.

Ваш работодатель не может заставить вас стать хорошим программистом, а зачастую он даже не может оценить, насколько хороши вы как программист. Если вы хотите стать отличным программистом, вы отвечаете за это сами. Это зависит от вашего характера.

Как только вы решили стать отличным программистом, перед вами открываются широкие перспективы. Исследования показывают, что лучшие программисты создают программы в 10 раз быстрее, чем их менее квалифицированные коллеги. Время, уходящее на отладку кода, а также объем и быстродействие итоговой программы, уровень ошибок и число обнаруженных ошибок также различаются примерно в 10 раз (Sackman, Erikson, and Grant, 1968; Curtis, 1981; Mills, 1983; DeMarco and Lister, 1985; Curtis et al., 1986; Card, 1987; Valett and McGarry, 1989).

Вы ничего не сделаете со своим интеллектом, но вы можете изменить свой характер — именно от характера зависит, станете ли вы превосходным программистом.

## 33.2. Интеллект и скромность

> Чтобы стать экспертом в практической или научной области, нужны огромный труд и долгое время. Если человек добросовестно трудится каждый час рабочего дня, когда-нибудь он проснется одним из самых компетентных специалистов своего поколения.
>
> *Уильям Джеймс (William James)*

Интеллект не кажется чертой характера и на самом деле не является им. Высочайший уровень интеллекта — далеко не главное условие для человека, желающего стать хорошим программистом.

Что? Для этого не нужно быть суперинтеллектуальным?

Нет, не нужно. Никто не обладает достаточным для программирования уровнем интеллекта. Чтобы полностью охватить и понять сразу все детали даже средней программы, человек должен был бы обладать почти неограниченными возможностями. Способ использования интеллекта важнее, чем его уровень.

Как было сказано в главе 5, лекцию, посвященную получению премии Тьюринга в 1972 году, Эдсгер Дейкстра назвал «The Humble Programmer» («Скромный программист»). Дейкстра заявил, что большинство аспектов программирования являет собой попытки компенсации строго ограниченных способностей разума. Самые лучшие программисты — те, кто понимают, насколько ограничены их возможности. Они скромны. Худшие программисты отказываются признать, что их способности не соответствуют задаче. Характер не позволяет им стать отличными программистами. Чем усерднее вы работаете над компенсацией ограниченных возможностей своего разума, тем лучше будете программировать. Быстрота вашего развития напрямую зависит от вашей скромности.

Многие методики эффективного программирования призваны снизить нагрузку на мозг. Вот несколько примеров.

- Цель «декомпозиции» системы — сделать систему проще для понимания (см. подраздел «Уровни проектирования» раздела 5.2).
- Обзоры, инспекции и тестирование представляют собой способы компенсации ожидаемых человеческих оплошностей. Эти методики обзоров возникли как часть «обезличенного программирования» (Weinberg, 1998). Если бы мы никогда не совершали ошибок, нам не нужно было бы выполнять обзоры своего кода. Однако наши интеллектуальные способности ограничены, поэтому мы дополняем их способностями других людей.
- Ограничение объема методов снижает нагрузку на ум.
- Написание программ в терминах проблемной области, а не низкоуровневых деталей реализации, преследует ту же цель.
- Использование всевозможных конвенций освобождает разум от забот о банальных аспектах программирования, не приносящих особой пользы.

Возможно, вы думаете, что было бы благороднее развить умственные способности и отказаться от этих костылей. Возможно, вам кажется, что программист, использующий умственные костыли, идет окольными путями. Однако эмпирически было показано, что скромные программисты, компенсирующие свои недостатки, пишут более понятный код, содержащий меньше ошибок. Настоящий окольный путь — это путь ошибок и сорванных планов.

## 33.3. Любопытство

Как только вы признали, что ваши способности слишком малы для понимания большинства программ, и поняли, что эффективное программирование — это поиск способов компенсировать данный недостаток, вы начинаете этот поиск, продолжающийся вплоть до окончания карьеры. А поэтому важная черта характера программиста — любопытство к техническим вопросам. Релевантная техническая информация постоянно изменяется. Многим Web-программистам никогда не приходилось программировать для Microsoft Windows, а многие разработчики программ для Windows никогда не имели дела с DOS, UNIX или перфокартами. Специфические особенности технической среды изменяются каждые 5—10 лет. Если вы недостаточно любопытны для того, чтобы не отставать от изменений, вы рискуете разделить участь динозавров.

Программисты часто так заняты работой, что у них не остается времени на изучение более эффективных способов работы. Если это относится и к вам, вы не одиноки. Ниже я расскажу, как развить любопытство и сделать обучение приоритетом.

***Изучите процесс разработки*** Чем больше вы узнаете о процессе разработки ПО из книг или на собственном опыте, тем лучше будете понимать изменения и тем эффективнее вести свою группу в правильном направлении.

**Перекрестная ссылка** О важности процесса разработки ПО см. раздел 34.2.

Если ваша работа состоит исключительно из краткосрочных заданий, не способствующих развитию навыков, подумайте, как исправить эту ситуацию. Если вы работаете на конкурентном рынке ПО, половина ваших знаний устареет за три года. Без обучения вы превратитесь в ископаемое.

Не тратьте время, работая на компанию, не учитывающую ваших интересов. Несмотря на экономические подъемы и спады и перемещение некоторых рабочих мест в другие регионы, ожидается, что среднее число рабочих мест, связанных с программированием, за период с 2002 до 2012 года в США значительно увеличится. Ожидается, что число должностей системных аналитиков вырастет примерно на 60%, а разработчиков ПО — примерно на 50%. Что касается всех должностей, связанных с компьютерами, то в дополнение к 3 миллионам имеющихся рабочих мест за это время будет создано еще около 1 миллиона мест (Hecker, 2001; BLS, 2004). Если работа не способствует вашему развитию, найдите другую работу.

***Экспериментируйте*** Экспериментирование с процессами программирования и разработки — эффективный способ самообразования. Если вы не знаете, как работает какая-то возможность языка, напишите небольшую программу и узнайте. Создайте прототип. Изучите выполнение программы в отладчике. Гораздо лучше написать короткую программу для проверки какой-то возможности, чем создать большую программу, реализовав в ней возможность, которую вы не совсем понимаете.

**Перекрестная ссылка** Идея экспериментирования лежит в основе нескольких ключевых аспектов программирования (см. подраздел «Экспериментирование» раздела 34.9).

А если короткая программа показывает, что какая-то функция работает не так, как вы хотите? Этого-то вам и нужно! Лучше обнаружить это в небольшой программе, чем в большой. Быстро совершая ошибки и извлекая из них уроки, вы облег-

чите себе путь к эффективному программированию. Ошибка — не грех. Неспособность к обучению на основе ошибок — вот что такое грех.

**Дополнительные сведения** Отличный учебник по решению проблем — книга Джеймса Адамса «Conceptual Blockbusting» (Adams, 2001).

***Читайте о решении проблем*** Решение проблем — главный аспект создания ПО. Эксперименты показали, что люди не всегда находят эффективные стратегии решения проблем сами, хотя легко обучаются этим стратегиям (Simon, 1996). Так что, даже если вы хотите изобрести колесо, успех не гарантирован — ваше колесо может оказаться квадратным.

***Анализируйте и планируйте, прежде чем действовать*** Анализ и действие несколько противоречат друг другу. В какой-то момент вы должны прекратить сбор данных и начать действовать. Однако проблемой большинства программистов является не избыточный анализ. Как правило, чаша действия значительно перевешивает чашу анализа, и проблема «аналитического паралича» крайне редко угрожает программистам.

http://cc2e.com/3320

***Изучайте успешные проекты*** Особенно хороший способ самообразования — изучение опыта лучших программистов. Джон Бентли (Jon Bentley) считает, что вы должны иметь возможность сесть в кресло со стаканом бренди и сигарой и читать программу, как хороший рассказ. Это не так неестественно, как кажется. Большинству людей не хотелось бы тратить свободное время на анализ 500-страничного листинга, но многим вполне понравилось бы изучить высокоуровневый проект кода и погрузиться в некоторые тонкости.

При разработке ПО крайне редко используются данные о прошлых успехах и неудачах. Если бы вас интересовала архитектура, вы изучили бы чертежи Луиса Салливана, Фрэнка Ллойда Райта и других известных архитекторов. Возможно, вы посетили бы спроектированные ими здания. Если бы вас интересовало проектирование строительных конструкций, вы изучили бы мосты Brooklyn Bridge, Tacoma Narrows Bridge и другие структуры из бетона, стали и дерева. Вы изучили бы примеры успехов и неудач в своей отрасли.

Томас Кун указывает на то, что частью любой зрелой науки является набор удачных решений проблем, которые можно использовать в качестве образцов при последующей работе (Kuhn, 1996). Разработка ПО только начинает приближаться к этому уровню зрелости. В 1990 году организация Computer Science and Technology Board пришла к выводу, что задокументированных исследований успехов и неудач в отрасли разработки ПО мало (CSTB, 1990).

В журнале «Communications of the ACM» приводятся доводы в пользу обучения на основе исследований конкретных проблем программирования (Linn and Clancy, 1992). Это кое о чем говорит. То, что один из самых популярных разделов в компьютерных журналах — «Programming Pearls» — был создан на основе исследований конкретных случаев проблем программирования, также наводит на мысли. А одна из самых популярных книг по разработке ПО — «Мифический человеко-месяц» — основана на исследовании управления проектом разработки IBM OS/360.

Имеется ли у вас книга, включающая исследования конкретных проблем программирования, или нет, ищите код, написанный лучшими программистами, и читайте его. Постарайтесь изучить код программистов, которых вы уважаете. Взгляни-

те на код программистов, которых вы не уважаете. Сравните их варианты кода между собой и со своим собственным. Каковы различия? Почему код различается? Какой вариант лучше? Почему?

Не ограничивайтесь чтением кода других программистов — старайтесь также узнать, что эксперты думают о вашем коде. Найдите высококлассных программистов, которые согласились бы покритиковать ваш код. Слушая критику, игнорируйте аспекты, связанные с их субъективными взглядами, и сосредоточьтесь на важных моментах. После этого измените свой подход к программированию так, чтобы он стал еще лучше.

***Читайте!*** У многих программистов документация вызывает страх. Очень часто она плохо написана и организована, и все же преодоление «документофобии» может принести большую пользу. Документация — это ключ к замку ПО, и ее чтение никогда не бывает пустой тратой времени. Игнорирование имеющейся информации стало настолько частым явлением, что привело к возникновению специального акронима «RTFM!», который расшифровывается как «Read the !#*%*@ Manual!»

Почти все современные языки дополняются огромным объемом библиотечного кода. Время, потраченное на изучение документации к библиотекам, будет хорошей инвестицией. Скорее всего компания, разработавшая данную версию языка, уже создала массу нужных вам классов. Если это так, приложите все усилия, чтобы ознакомиться с ними. Просматривайте документацию примерно каждые два месяца.

***Читайте другие книги и периодические издания*** Похвалите себя за чтение этой книги. За это время вы продвинулись вперед гораздо дальше, чем большинство ваших коллег, потому что объем этой книги превышает годичный объем чтения большинства программистов (DeMarco and Lister, 1999). Неторопливое, но регулярное чтение — надежный путь к высоким профессиональным достижениям. Если, читая примерно по 35 страниц в неделю, вы будете прочитывать одну хорошую книгу по программированию каждые два месяца, скоро вы получите основательный багаж знаний и начнете выгодно отличаться почти от всех окружающих вас разработчиков.

**Перекрестная ссылка** Книги, которые следовало бы прочитать любому программисту, указаны в разделе 35.4.

***Общайтесь с единомышленниками*** Познакомьтесь с другими людьми, стремящимися улучшить навыки разработки ПО. Посещайте конференции, вступите в группу пользователей, общайтесь на Интернет-форумах.

***Постоянно стремитесь к профессиональному развитию*** Хорошие программисты всегда ищут возможности дальнейшего совершенствования. Обдумайте следующую лестницу профессионального развития, используемую в моей и нескольких других компаниях.

**Дополнительные сведения** Об уровнях развития программиста см. главу 16 книги «Professional Software Development» (McConnell, 2004).

- **Уровень 1: начало** Новичок — это программист, способный использовать базовые возможности одного языка. Такой человек может писать классы, методы, циклы и условные операторы, а также использовать многие средства, поддерживаемые языком.

- **Уровень 2: средний уровень** Программист среднего уровня прошел начальный этап, может использовать базовые возможности нескольких языков и очень хорошо владеет по меньшей мере одним языком.
- **Уровень 3: компетентность** Компетентный программист обладает экспертными знаниями языка, среды или того и другого. Программист этого уровня может знать все тонкости J2EE или помнить все аннотированное справочное руководство по C++. Такие программисты очень ценны для работодателей, и многие из программистов никогда не поднимаются выше этого уровня.
- **Уровень 4: лидерство** Лидер имеет квалификацию программиста 3-го уровня и понимает, что программирование на 85% состоит из общения с людьми и лишь на 15% — из общения с компьютером. Средний программист только 30% времени работает в одиночку (McCue, 1978) и еще меньше времени тратит на взаимодействие с компьютером. Гуру программирования пишут код для людей, а не для машин. Истинные гуру пишут абсолютно ясный код, не забывая при этом комментировать его. Они не хотят тратить свои драгоценные умственные ресурсы на восстановление логики какого-то фрагмента кода, если ее можно было бы легко понять, прочитав краткий комментарий.

Прекрасный программист, не уделяющий должного внимания удобочитаемости кода, вероятно, сможет достичь лишь уровня 3, да и то в самом лучшем случае. Судя по моему опыту, главная причина нечитабельности кода в том, что код плох. Никто не говорит себе: «Мой код плох, поэтому я сделаю его непонятным». Программисты просто не понимают свой код достаточно хорошо, чтобы сделать его удобочитаемым, и это запирает их на одном из более низких уровней.

Самый худший код, который я когда-либо видел, написала женщина, никому не позволявшая изучать ее программы. В конце концов начальник пригрозил ей увольнением, если она не будет сотрудничать с коллегами. В ее коде полностью отсутствовали комментарии, но зато в избытке имелись глобальные переменные с именами вроде *x, xx, xxx, xx1* и *xx2*. Начальник ее начальника считал ее отличным программистом, потому что она быстро исправляла ошибки. Качество ее кода предоставило ей массу возможностей продемонстрировать свой талант исправления ошибок на деле.

Быть программистом начального или среднего уровня — не грех. Быть компетентным программистом, а не лидером, также не грех. Но если вы знаете, что нужно делать для собственного развития, и ничего не предпринимаете, иначе как грехом это назвать нельзя.

## 33.4. Профессиональная честность

Становление высококвалифицированного программиста предполагает развитие обостренного чувства профессиональной честности, которая может проявляться в самых разных формах:

- отказ от притязаний на роль эксперта, если вы им не являетесь;
- охотное признание собственных ошибок;
- стремление разобраться в предупреждениях компилятора вместо их отключения;

- желание ясно понять программу и отказ от компиляции кода с той лишь целью, чтобы узнать, работает ли он;
- предоставление реалистичных отчетов о статусе проекта;
- предоставление реалистичных оценок срока выполнения проекта и отстаивание своей позиции, даже если руководители просят адаптировать оценку.

Два первых элемента этого списка — признание собственных недостатков и ошибок — можно считать отголосками интеллектуальной скромности, рассмотренной выше. Как вы узнаете что-то новое, если будете считать, что уже все знаете? Если на то пошло, гораздо лучше считать, что вы ничего не знаете. Прислушивайтесь к мнениям людей и учитесь у них, но старайтесь при этом понять, действительно ли *они* знают, что говорят.

Оценивайте степень своей уверенности в тех или иных суждениях. Если вы обычно уверены в них на 100%, это предупреждающий знак.

> Любой дурак способен отстаивать свои ошибки — большинство дураков именно так и делают.
>
> *Дейл Карнеги (Dale Carnegie)*

Отказ от признания ошибок — особенно вредная привычка. Если Салли отказывается признать ошибку, она, очевидно, считает, что сможет убедить в этом окружающих, но на самом деле имеет место обратное. Все будут знать, что ошибку допустила Салли. Ошибки — естественный результат подъемов и спадов сложных интеллектуальных процессов, и если Салли не была небрежной, никто не будет ее ни в чем упрекать.

Если она отказывается признать ошибку, она сможет одурачить только одного человека — себя. Что касается ее коллег, то они поймут, что они работают с надменным и не совсем искренним программистом. Это уже не просто ошибка, а нечто худшее. Если вы совершили ошибку, признавайте это быстро и охотно.

Еще один распространенный недостаток — убежденность в понимании сообщений компилятора. Если вы не понимаете предупреждение компилятора или думаете, что понимаете, но вам не хватает времени, чтобы проверить свое предположение, к чему это приведет? Вероятно, в итоге вам придется решать проблему с нуля, тогда как компилятор буквально размахивал готовым решением перед вашим носом. Если меня просят помочь отладить программу, я спрашиваю, компилируется ли она без ошибок и предупреждений. Довольно часто программисты отвечают «да» и начинают объяснять симптомы проблемы. Я отвечаю: «Хм... Похоже на неинициализированный указатель, но компилятор должен был предупредить вас об этом». В ответ на это они заявляют: «О да, он предупреждал об этом, но мы думали, что это означает что-то другое». Совершив ошибку, вы едва ли сможете убедить людей в том, что она не ваша. Обмануть компьютер еще сложнее, так что не тратьте попусту свое время.

Похожий вид интеллектуальной небрежности имеет место тогда, когда вы не совсем понимаете свою программу и «просто компилируете ее, чтобы увидеть, будет ли она работать». Примером может служить запуск программы с целью определить, какое условие следует использовать: < или <=. На самом деле работоспособность программы в этой ситуации не играет никакой роли, потому что вы понимаете ее недостаточно хорошо, чтобы сказать, почему она работает. Помните: тестирование может указать только на наличие, но не на отсутствие ошибок. Если вы не понимаете программу, вы не сможете тщательно ее протестировать. Ком-

пиляция программы с целью «посмотреть, что будет» — предупреждающий знак. Это может означать, что вам нужно вернуться к проектированию или что вы приступили к кодированию, плохо понимая свои действия. Прежде чем компилировать программу, убедитесь, что вы хорошо понимаете ее.

> На первые 90% кода приходятся первые 90% времени разработки. На оставшиеся 10% кода приходятся другие 90% времени разработки.
>
> *Том Каргилл (Tom Cargill)*

Оценка статуса проекта — один из главных источников лжи в нашей работе. Программисты печально известны своими заявлениями, согласно которым на протяжении всей второй половины проекта программа неизменно «готова на 90%». Если вы плохо представляете ход работы над проектом, постарайтесь лучше разобраться в методах работы. Но если вы не говорите правду, потому что хотите угодить своим руководителям, это совсем другое дело. Как правило, руководители будут благодарны вам за объективные оценки статуса проекта, даже если эти оценки им не понравятся. Если ваши выводы обоснованы, высказывайте их лично и без всяких прикрас. Для координации процессов разработки руководителям нужна точная информация, и искреннее сотрудничество играет при этом очень важную роль.

> http://cc2e.com/3341

С сообщением неверного статуса проекта связана неверная оценка сроков выполнения проекта. Типичный сценарий таков: руководитель спрашивает у Берта, сколько времени потребуется на разработку новой БД. Берт беседует с программистами, подсчитывает некоторые показатели, возвращается и говорит, что над проектом должны будут работать восемь программистов в течение шести месяцев. Руководитель говорит: «Нас это не устраивает. Можно ли выполнить проект быстрее и с меньшим числом программистов?» Берт уходит, думает и решает, что он мог бы сократить время обучения и отпуска и попросить каждого из программистов поработать сверхурочно. Он возвращается и говорит, что проект будет реализован шестью программистами за четыре месяца. Руководитель заявляет: «Отлично. Это довольно низкоприоритетный проект, поэтому постарайтесь выполнить его точно вовремя: мы не справимся с дополнительными расходами».

К сожалению, Берт не учел, что оценка не может являться предметом переговоров. Он может пересмотреть оценку и сделать ее более точной, но переговоры с руководителем не повлияют на время, необходимое для реализации проекта. Билл Ваймер из IBM однажды заявил: «Мы обнаружили, что технические специалисты в целом очень точно оценивали требования, предъявляемые к проектам, и сроки их реализации. Но они не могли защитить свои решения — им нужно научиться отстаивать свое мнение» (Weimer в Metzger and Boddie, 1996). Пообещав выполнить проект за четыре месяца и выполнив за шесть, Берт едва ли обрадует своего руководителя сильнее, чем пообещав выполнить проект за шесть месяцев и сдержав свое слово. Не сдержав обещания, Берт потеряет доверие, а отстаивая свою позицию, лишь заслужит уважение.

Если руководство давит на вас, желая услышать более приемлемую оценку, дайте понять, что окончательное решение о том, следует ли браться за проект, остается за ними: «Смотрите: вот сколько это будет стоить. Я не знаю, приемлема ли такая цена для компании, — это ваша работа. Но я могу сказать, сколько времени потребуется для разработки ПО, — это моя работа. Обсуждать сроки в данном слу-

чае неуместно: это все равно что обсуждать, сколько футов в миле. Мы не можем пересмотреть законы природы. Однако мы можем пересмотреть другие аспекты проекта, влияющие на срок его выполнения, и переоценить его. Мы можем отказаться от некоторых возможностей, снизить производительность, выполнить проект инкрементно, а также привлечь к работе меньшее число людей и установить более длительный срок выполнения проекта или наоборот: привлечь больше разработчиков и реализовать проект быстрее».

Один из самых ужасных советов я получил на лекции об управлении проектами разработки ПО. Лектором был автор бестселлера по управлению проектами. Один из присутствующих спросил: «Что вы делаете, если руководители просят оценить срок выполнения проекта, но вы знаете, что если вы сообщите верную оценку, они откажутся от проекта?» Лектор ответил, что в такой щекотливой ситуации следует склонить руководителей к реализации проекта, недооценив его. Он сказал, что, как только руководители инвестируют средства в первую часть проекта, им придется довести его до конца.

Абсолютное заблуждение! Начальство отвечает за управление компанией в целом. Если какая-то функция программы обойдется компании в 250 000 долларов, а вы оцените ее в 750 000, разрабатывать программу не следует. Принимать такие решения должны руководители. Когда лектор отстаивал недооценку проекта, он отстаивал скрытое воровство власти у руководителей. Если вы считаете, что проект интересен, открывает перед компанией новые возможности или позволяет многому научиться, так и скажите. Руководители тоже способны взвесить эти факторы. Но подталкивание их к неверным решениям может стоить компании сотен тысяч долларов. Если из-за этого вы лишитесь работы, пеняйте на себя.

## 33.5. Общение и сотрудничество

По-настоящему отличные программисты учатся эффективно сотрудничать, что всегда подразумевает написание удобочитаемого кода. Вероятно, компьютер читает вашу программу так же часто, как другие люди, но он читает плохой код гораздо лучше, чем люди. Работая над кодом, не забывайте про людей, которым придется изменять его в будущем. Программирование — это в первую очередь общение с другим программистом и только во вторую — с компьютером.

## 33.6. Творчество и дисциплина

*Закончив институт, я считал себя лучшим программистом в мире. Я мог разработать непобедимый алгоритм игры в крестики-нолики, владел пятью разными языками и мог писать программы из 1000 строк, которые РАБОТАЛИ (в самом деле!). Затем я очутился в Реальном Мире. Первая моя задача в Реальном Мире требовала, чтобы я разобрался в программе из 200 000 строк, написанной на Фортране, и ускорил ее в два раза. Любой Реальный Программист скажет вам, что никакие методики Структурного Кодирования никогда не помогут решить подобную проблему — для этого требуется настоящий талант.*

*Эд Пост (Ed Post)*

Выпускнику института трудно объяснить важность конвенций и дисциплины. Самая крупная программа, написанная мной в институте, состояла примерно из 500 строк исполняемого кода. За свою профессиональную карьеру я написал десятки утилит, включающих менее 500 строк, однако в среднем мои проекты содержали от 5000 до 25 000 строк, а объем некоторых проектов, в которых я принимал участие, достигал полумиллиона строк. Подобные проекты требуют не тех же навыков в более крупном масштабе, а совершенно иных навыков.

Некоторые программисты считают, что стандарты и конвенции подавляют свободу творчества, но с этим трудно согласиться. Можете ли вы представить Web-сайт, на каждой странице которого использовались бы разные шрифты, цвета, способы выравнивания текста, графические стили и способы навигации? Какое уж тут творчество — это хаос. Если стандарты и конвенции не используются в крупном проекте, завершить его становится невозможно. Не тратьте свою творческую энергию на то, что не играет никакой роли. Установите конвенции для второстепенных областей и сосредоточьтесь на действительно важных аспектах.

Анализируя 15-летний опыт работы в Лаборатории проектирования ПО NASA, Макгарри и Паджерски пришли к выводу, что особенно эффективными были методики и инструменты, акцентированные на дисциплине (McGarry and Pajerski, 1990). Многие в высшей степени творческие люди отличаются крайней дисциплинированностью. «Форма освобождает», — гласит пословица. Прекрасные архитекторы и художники работают в условиях ограниченности физических материалов, времени и расходов. Любой, кто изучал произведения Леонардо да Винчи, не может не восхищаться его дисциплинированным вниманием к подробностям. Перед росписью свода Сикстинской капеллы Микеланджело разделил его на симметричные наборы геометрических фигур, таких как треугольники, окружности и прямоугольники. Он разделил свод на три зоны в соответствии с тремя этапами Платона. Без этой структуры и дисциплины 300 человеческих фигур были бы просто хаотично разбросанными фрагментами, а не согласованными элементами художественного шедевра.

Создание шедевра программирования требует не меньшей дисциплины. Если вы не проанализируете требования и проект до начала кодирования, многие знания вам придется приобретать во время кодирования, и результат вашей работы будет больше напоминать мазню трехлетнего ребенка, а не произведение искусства.

## 33.7. Лень

Лень: качество, которое заставляет прилагать больше усилий для снижения общих затрат энергии. Она заставляет писать программы, облегчающие труд, и документировать написанное, чтобы вам не пришлось отвечать на лишние вопросы.

*Ларри Уолл (Larry Wall)*

Лень может проявляться несколькими способами:

- отсрочка выполнения неприятной задачи;
- немедленное выполнение неприятной задачи с целью как можно более быстрого избавления от нее;
- написание инструмента для выполнения неприятной задачи, чтобы ее никогда не пришлось выполнять снова.

Одни проявления лени лучше, другие — хуже. Первое едва ли когда-нибудь бывает выгодным. Наверное, и вы когда-то тратили несколько часов на выполнение необязательных

задач, желая отсрочить решение относительно неважной задачи, избежать которой тем не менее вы не могли. Я ненавижу ввод данных, и многие программы требуют ввода небольшого объема данных. Я иногда откладываю работу над программой на несколько дней только затем, чтобы отсрочить неизбежный ввод нескольких страниц чисел вручную. Это «истинная лень». Она проявляется и в компиляции класса с целью проверки его работоспособности и избавления от проверки класса в уме.

Небольшие задачи никогда не бывают такими плохими, какими кажутся. Вы сможете избавиться от первого типа лени, если выработаете привычку выполнять эти задачи сразу. Эта привычка соответствует второму типу лени — «просвещенному». Вы все так же ленитесь, но решаете неприятные проблемы, тратя на них как можно меньше времени.

Третий вариант лени предполагает написание инструмента для выполнения неприятной задачи. Это «долговременная лень». Несомненно, это самый продуктивный вид лени (если, конечно, инструмент позволяет в итоге сэкономить время). В этом контексте определенная лень даже выгодна.

Однако лень имеет и обратную сторону. «Спешка» и «усилия» ценятся в программировании совсем не так высоко, как на уроках физкультуры. Спешка — это дополнительные, ненужные усилия. Она указывает на активность, но не на выполнение работы. Движение нетрудно спутать с прогрессом, а занятость с продуктивностью. Главную роль в эффективном программировании играет мышление, а размышляющие люди обычно не кажутся занятыми. Если бы я видел, что какой-то программист постоянно занят, я подумал бы, что он — неважный программист, потому что он не использует свой наиболее ценный инструмент, которым, как известно, является голова.

## 33.8. Свойства, которые менее важны, чем кажется

Помимо спешки есть и другие свойства, уместные в других областях жизни, но не особо эффективные при разработке ПО.

### Настойчивость

Как и большинство подобных субъективных понятий, настойчивость в зависимости от обстоятельств может быть и достоинством, и недостатком и обозначается разными словами. Если вы хотите считаете ее плохим качеством, вы называете ее «упрямством», а то и «ослиным упрямством». Если вы желаете придать ей хороший оттенок, можете назвать ее «упорством».

Как правило, при разработке ПО настойчивость принимает форму ослиного упрямства, т. е. пользы не приносит. Настойчивое стремление довести код до ума с использованием одного подхода трудно признать достоинством. Попробуйте перепроектировать класс, перепишите код или вернитесь к проблеме позже. Если один подход не работает, самое время испытать другой (Pirsig, 1974).

**Перекрестная ссылка** О настойчивости при отладке см. подраздел «Советы по поиску причин дефектов» раздела 23.2.

Во время отладки иногда очень увлекательно отслеживать надоедливую ошибку четыре часа, но, если в течение какого-то времени (скажем, 15 минут) вы не добиваетесь прогресса, обычно лучше отложить поиск ошибки. Позвольте своему подсознанию немного поработать над проблемой. Попробуйте подумать об альтернативном подходе, который вообще устранил бы проблему. Перепишите проблемный фрагмент кода с нуля. Вернитесь к нему со свежей головой. В борьбе с компьютерными проблемами нет ничего благородного. Лучше их избегать.

Трудно сказать, когда отложить работу, но очень важно, чтобы вы спрашивали себя об этом. Если вы чувствуете замешательство, задайте себе этот вопрос. Это не всегда означает, что пришло время сдаться, однако скорее всего это значит, что пора установить некоторые временные параметры: «Если я не решу эту проблему с использованием этого подхода в следующие полчаса, я проведу 10-минутный „мозговой штурм“ в поиске других подходов и попробую в течение следующего часа самый лучший из них».

## Опыт

По ряду причин важность практического опыта в сравнении с книжным образованием в области разработки ПО не столь велика, как во многих других областях. В других областях базовые положения изменяются довольно медленно, вследствие чего люди, закончившие вуз с интервалом в 10 лет, обладают по сути одинаковыми знаниями. В отрасли разработки ПО даже основы претерпевают быстрые изменения. Человек, закончивший вуз через 10 лет после вас, вероятно, будет знать вдвое больше об эффективных методиках программирования. К программистам старшего возраста относятся с подозрением не только потому, что они якобы не имеют представления об отдельных технологиях, а потому, что они, возможно, никогда и не сталкивались с базовыми концепциями программирования, распространившимися после того, как они закончили вуз.

В других областях текущие знания человека о работе будут служить ему и завтра. Что до разработки ПО, то, если программист неспособен пересматривать привычные способы мышления, приобретенные им при использовании предыдущего языка программирования, или методики оптимизации кода, работавшие на старом компьютере, наличие опыта окажется худшим вариантом, чем его полное отсутствие. Многие программисты тратят время на подготовку к завершившейся битве, а не предстоящей. Если вы не изменяетесь вместе со временем, опыт скорее вредит, чем помогает.

Кроме того, опираясь на собственный опыт, люди часто делают неверные выводы. Трудно объективно оценить свою жизнь. Вы можете упустить из виду ключевые элементы своего опыта, которые подтолкнули бы вас к другим выводам, если бы вы их учли. В этом смысле могут помочь книги о работе других программистов, потому что так вы узнаете опыт других людей, достаточно очищенный, чтобы его можно было изучить объективно.

Стоит упомянуть также абсурдное внимание к *объему* опыта программистов. «Нам нужен программист, обладающий 5-летним опытом программирования на C» —

глупое высказывание. Если программист не изучил C за год или два, еще три года не сыграют особой роли. Подобный вид «опыта» очень слабо связан с производительностью труда.

Быстрые изменения информации в сфере программирования создают странную динамику в области «опыта». Во многих отраслях специалист, имеющий за плечами массу успехов и достижений, может расслабиться и наслаждаться заслуженным уважением. В то же время знания расслабившегося программиста очень быстро устаревают. Чтобы поддерживать компетентность, вы должны идти в ногу со временем. Для молодых и энергичных программистов это преимущество. Более пожилые программисты иногда чувствуют, что они уже заслужили свои эполеты, и возмущаются, когда их год за годом принуждают подтверждать квалификацию.

Главное то, что опыт может иметь разное качество. Если вы работаете 10 лет, получаете ли вы 10 лет опыта или 1 год опыта 10 раз? Чтобы приобрести истинный опыт, вы должны действовать с умом. Если вы постоянно учитесь, вы приобретаете опыт. Если вы не учитесь, о приобретении опыта не может быть и речи, как бы долго вы ни работали.

### Страсть к программированию

*Если вы не проводили хотя бы месяц, работая над одной программой — работая по 16 часов в день, грезя о ней в остальные 8 часов беспокойного сна, работая несколько ночей подряд над устранением из программы «одной последней ошибки», — вы не писали сложную компьютерную программу. Тогда вам трудно понять, что в программировании есть что-то захватывающее.*

*Эдвард Йордон (Edward Yourdon)*

Настолько щедрая дань богам программирования — едва ли не самый верный путь к неудаче. Ночные бдения позволят вам на какое-то время почувствовать себя самым великим программистом в мире, но потом вам придется потратить несколько недель на исправление дефектов, внесенных в код в безудержном порыве. Во что бы то ни стало вызовите у себя увлечение программированием, но помните, что увлечение никогда не заменит компетентности.

## 33.9. Привычки

*Следовательно, нравственные добродетели существуют в нас не от природы и не вопреки природе... а благодаря приучению мы в них совершенствуемся... Ибо если нечто следует делать, пройдя обучение, то учимся мы, делая это... Хорошо строя дома, люди станут добрыми зодчими, а строя худо — худыми... Так что вовсе не мало, а очень много, пожалуй, даже все зависит от того, к чему именно приучаться с самого детства.*

*Аристотель*

Выработать хорошие привычки крайне важно, потому что очень многое из того, что вы делаете как программист, вы делаете не задумываясь. Например, когда-то вы могли думать о форматировании циклов, но вы не думаете об этом при написании каждого нового цикла. Вы пишете их как привыкли. Это относится почти ко всем аспектам форматирования кода. Когда вы в последний раз всерьез заду-

мывались о своем стиле форматирования? Если вы программируете около пяти лет, скорее всего четыре с половиной года назад. Позднее вы просто следовали привычке.

Привычки связаны со многими областями. Так, программисты обычно тщательно проверяют индексы циклов и не проверяют операторы присваивания, из-за чего ошибки в операторах присваивания гораздо сложнее искать, чем ошибки индексов циклов (Gould, 1975). Вы отвечаете на критику дружелюбно или недружелюбно. Вы всегда стремитесь сделать код удобочитаемым или быстрым или не обращаете на это никакого внимания. Если при выборе между написанием быстрого или удобочитаемого кода вы каждый раз делаете один и тот же выбор, вы на самом деле не выбираете — вами движет привычка.

Взгляните на изречение Аристотеля еще раз и замените слова «нравственные добродетели» на «программистские добродетели». Аристотель утверждает, что вы не предрасположены ни к хорошему, ни к плохому поведению, поэтому вы можете стать как хорошим, так и плохим программистом. Главным способом становления хорошим или плохим в своей области является сама деятельность: строительство в случае строителей и программирование в случае программистов. То, что вы делаете, становится привычкой, и со временем именно привычки начинают определять, хороший вы программист или плохой.

Билл Гейтс говорит, что любой программист, который впоследствии станет хорошим, хорош уже в первые несколько лет. После этого измениться практически невозможно (Lammers, 1986). Если вы программируете уже много лет, вряд ли вы внезапно зададитесь вопросом: «Как я делаю этот цикл быстрее?» или «Как я делаю этот код более удобочитаемым?» Это привычки, которые хорошие программисты вырабатывают на самых ранних стадиях обучения.

Обучаясь делать что-то, сразу учитесь делать это правильно. В первый раз вы активно обдумываете свои действия и все еще можете с легкостью изменить свой подход. Выполнив что-то несколько раз, вы начинаете уделять меньше внимания своим действиям, и «сила привычки» берет свое. Проверяйте, что вы приобретаете именно те привычки, какие хотите.

Что, если вы еще не выработали самые эффективные привычки? Как изменить плохую привычку? Будь у меня окончательный ответ на этот вопрос, я бы продавал видеокассеты с записями курсов самопомощи!.. Но один стоящий совет я дам. Вы не можете заменить плохую привычку на отсутствие привычки. Именно поэтому люди, пытающиеся бросить курить, сквернословить или переедать, испытывают огромные затруднения, пока не заменят старую привычку на какую-то другую, например, жевание жевательной резинки. Легче заменить старую привычку на новую, чем полностью избавиться от привычки. Таким образом, попробуйте выработать новые, эффективные привычки. Например, выработайте привычку писать класс на псевдокоде перед кодированием или тщательно читать код перед его компиляцией. Тогда вам не придется беспокоиться об избавлении от плохих привычек — они естественным путем будут вытеснены новыми привычками.

# Дополнительные ресурсы

«Человеческому фактору» в разработке ПО посвящены следующие материалы.

http://cc2e.com/3327

Dijkstra, Edsger. «The Humble Programmer.» Turing Award Lecture. *Communications of the ACM* 15, no. 10 (October 1972): 859–66. Эта классическая работа способствовала началу исследования степени зависимости программирования от умственных способностей программиста. Дейкстра постоянно подчеркивает мысль, что основная задача программирования — обуздание огромной сложности компьютерных наук. Он утверждает, что программирование является единственной профессией, представителям которой приходится иметь дело с девятью порядками разницы между самым низким и самым высоким уровнями детальности. Многие мысли Дейкстры свежи и по сей день, однако эту работу интересно прочитать хотя бы только из-за ее исторического значения. Вы почувствуете, что значило быть программистом на ранних этапах развития вычислительной техники.

http://cc2e.com/3334

Weinberg, Gerald M. *The Psychology of Computer Programming: Silver Anniversary Edition*. New York, NY: Dorset House, 1998. В этой классической книге подробно рассматривается идея обезличенного программирования и другие аспекты человеческой стороны программирования. Она содержит много увлекательных историй и является одной из самых увлекательных книг, посвященных разработке ПО.

Pirsig, Robert M. *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values*. William Morrow, 1974. Пирсиг подробно обсуждает «качество» в контексте обслуживания мотоциклов. Во время написания этой книги Пирсиг работал техническим писателем в отрасли программирования, и его проницательные комментарии относятся к психологии программирования в не меньшей степени, чем к обслуживанию мотоциклов.

Curtis, Bill, ed. *Tutorial: Human Factors in Software Development*. Los Angeles, CA: IEEE Computer Society Press, 1985. Это великолепная подборка статей, посвященных человеческим аспектам создания программ. 45 статей разделены на следующие категории: психологические модели знаний о программировании, обучение программированию, решение проблем и проектирование, следствия репрезентаций проектирования, характеристики языка, диагностика ошибок и методология. Если программирование — одна из самых сложных интеллектуальных задач в истории человечества, программисты просто обязаны хорошо разбираться в человеческих умственных способностях. Кроме того, эти статьи помогут вам узнать, как лично вы можете стать более эффективным программистом.

McConnell, Steve. *Professional Software Development*. Boston, MA: Addison-Wesley, 2004. В седьмой главе этой книги обсуждаются вопросы личности и характера программистов.

## Ключевые моменты

- Способность к написанию программ напрямую зависит от личного характера.
- Важнейшие качества программиста — скромность, любопытство, профессиональная честность, творчество и дисциплина, а также «просвещенная» лень.
- Чтобы стать отличным программистом, можно не обладать особым талантом, но необходимо постоянно стремиться к самосовершенствованию.
- Удивительно, но интеллект, опыт и настойчивость вредят программистам не меньше, чем помогают.
- Многие программисты не ведут активного поиска новых сведений и методик, а полагаются на случайные столкновения с новой информацией на работе. Если вы посвятите небольшую долю своего времени чтению книг и изучению программирования, через несколько месяцев вы будете намного превосходить почти всех своих коллег.
- Хороший характер во многом — продукт правильных привычек. Если хотите стать великолепным программистом, выработайте правильные привычки, а все остальное придет само собой.

ГЛАВА 34

# Основы мастерства

**Содержание**

http://cc2e.com/3444



**Связанные темы**

- Вся книга

Эта книга посвящена главным образом деталям конструирования ПО: созданию высококачественных классов и циклов, выбору имен переменных, форматированию исходного кода, интеграции системы и т. д. Абстрактные вопросы в ней отчасти принесены в жертву более конкретным темам.

Поскольку конкретные вопросы уже рассмотрены в предыдущих главах, то, чтобы получить представление об абстрактных концепциях, нам нужно лишь вернуться к темам разных глав и посмотреть, как они взаимосвязаны. В этой главе явно обсуждаются абстрактные аспекты программирования, такие как сложность, абстракция, процесс разработки, удобочитаемость кода, итерация и т. д. Эти аспекты во многом объясняют разницу между хакерством и мастерством разработки ПО.

## 34.1. Боритесь со сложностью

Стремление к снижению сложности играет такую важную роль в программировании, что в главе 5 управление сложностью получило статус Главного Технического Императи-

**Перекрестная ссылка** О важности склада ума для борьбы со сложностью см. раздел 33.2.

ва Разработки ПО. Конечно, соблазнительно попытаться сыграть героя и сражаться с компьютерными проблемами на всех уровнях, однако никто не обладает умом, способным охватить девять порядков разницы в подробности. В компьютерных науках разработаны многие интеллектуальные инструменты борьбы с такой сложностью, которые мы уже затрагивали в других главах.

- Разделите систему на подсистемы на уровне архитектуры, чтобы концентрироваться в каждый конкретный момент времени на меньшей части системы.
- Тщательно определяйте интерфейсы классов, чтобы можно было игнорировать внутреннее устройство классов.
- Поддерживайте абстракцию, формируемую интерфейсом класса, чтобы не запоминать ненужных деталей.
- Избегайте глобальных данных, потому что их использование значительно увеличивает процент кода, который нужно удерживать в уме в любой момент времени.
- Избегайте глубоких иерархий наследования, потому что они предъявляют высокие требования к интеллекту.
- Избегайте глубокой вложенности циклов и условных операторов, поскольку их можно заменить на более простые управляющие структуры, позволяющие бережнее расходовать умственные ресурсы.
- Избегайте операторов *goto*, так как они вносят в программу нелинейность, за которой большинству людей трудно следовать.
- Тщательное определите подход к обработке ошибок, вместо того чтобы использовать произвольную комбинацию разных методик.
- Систематично используйте встроенный механизм исключений, поскольку он может стать нелинейной управляющей структурой, которую при недисциплинированном применении понять почти так же трудно, как и операторы *goto*.
- Не позволяйте классам превращаться в монстров, достигающих размера целых программ.
- Поддерживайте методы короткими.
- Используйте ясные, очевидные имена переменных, чтобы не вспоминать детали вроде «*i* — это индекс счета, а *j* — индекс клиента или наоборот?».
- Минимизируйте число параметров, передаваемых в метод, или, что еще важнее, передавайте только те параметры, которые нужны для поддержания абстракции, формируемой интерфейсом метода.
- Используйте соглашения, чтобы не запоминать произвольные, несущественные различия между разными фрагментами кода.
- В целом боритесь по мере возможности с тем, что в главе 5 было названо «несущественной сложностью».

Создавая для сложного теста булеву функцию и абстрагируя суть теста, вы упрощаете код. Заменяя сложную цепь логических структур на обращение к таблице, вы делаете то же самое. Создавая хорошо определенный согласованный интерфейс класса, вы избегаете тревоги насчет деталей реализации класса и в целом упрощаете свою работу.

Сутью соглашений кодирования также является преимущественно снижение сложности. Стандартизуя решения о форматировании, циклах, именах переменных, нотациях моделирования и т. д., вы освобождаете умственные ресурсы, которые пригодятся вам для концентрации на более сложных аспектах программирования. Одна из причин таких жарких споров по поводу соглашений кодирования состоит в том, что выбор того или иного варианта по сути произволен, хотя и имеет некоторую эстетическую основу. Наиболее страстно люди спорят о самых небольших различиях соглашений. Соглашения наиболее полезны, когда освобождают вас от принятия и защиты несущественных решений, и менее ценны, если налагают ограничения в более важных областях.

Особенно мощным средством управления сложностью является абстракция в разных проявлениях. Программирование развивается преимущественно за счет повышения абстрактности программных компонентов. Фред Брукс утверждает, что самым крупным достижением в компьютерных науках можно считать переход от машинного языка к высокоуровневым языкам: он освободил программистов от забот об особенностях отдельных устройств и позволил сосредоточиться на самом программировании (Brooks, 1995). Другим важным достижением стало изобретение методов, за которыми последовали классы и пакеты.

Функциональное именование переменных, отвечающее на вопрос «что?» уровня проблемы, а не «как?» уровня реализации, повышает уровень абстракции. Если вы говорите: «Я выталкиваю элемент из стека, получая данные о самом последнем сотруднике», — абстракция может избавить вас от выполнения умственного этапа «Я выталкиваю элемент из стека». Вы просто говорите: «Я получаю данные о самом последнем сотруднике». Эта выгода невелика, но если вы пытаетесь сократить диапазон сложности, простирающийся от 1 до $10^9$, важен каждый шаг к цели. Использование именованных констант вместо литералов также повышает уровень абстракции. Объектно-ориентированное программирование обеспечивает уровень абстракции, относящийся одновременно и к алгоритмам, и к данным, который функциональная декомпозиция сама по себе обеспечить не может.

Короче, главная цель проектирования и конструирования ПО — укрощение сложности. Снижение сложности лежит в основе многих методик программирования, и, наверное, его можно считать самым важным условием эффективного программирования.

## 34.2. Анализируйте процесс разработки

Вторая красная нить, проходящая через всю эту книгу, — подчеркивание на удивление большой важности используемого вами процесса разработки ПО. В небольшом проекте качество программы сильнее всего зависит от таланта конкретного программиста, а успешность программиста частично определяется используемыми им процессами.

В проектах, реализуемых с участием более одного программиста, более важную роль играют организационные характеристики, а не индивидуальные навыки. Даже если у вас отличная группа, ее коллективная способность не эквивалентна сумме способностей отдельных членов. Именно способ совместной работы определяет,

суммируются способности отдельных программистов или одна вычитается из другой. Используемый группой процесс — вот от чего зависит, будет ли работа конкретного программиста повышать или понижать эффективность работы остальных членов группы.

**Перекрестная ссылка** Советы по повышению стабильности требований см. в разделе 3.4, о различиях подходов к разработке — раздел 3.2.

Примером важности процесса разработки может служить обеспечение стабильности требований до начала проектирования и кодирования. Если вы не знаете, что́ создаете, вы не можете это хорошо спроектировать. Если требования, а позднее и проект, изменятся во время разработки, код тоже придется изменить, что может ухудшить качество системы.

«Конечно, — скажете вы, — но в реальном мире требования никогда не бывают по-настоящему стабильными, поэтому данный пример неуместен». Опять-таки используемый вами процесс определяет, насколько стабильны ваши требования и насколько стабильными они должны быть. Если вы хотите сделать требования более гибкими, вы можете использовать инкрементную разработку, планируя поставить ПО заказчику несколькими частями, а не все сразу. Именно внимание к процессу и используемый вами процесс в конечном счете определяют успех или неудачу проекта. Данные табл. 3-1 (см. раздел 3.1), ясно говорят, что ошибки в требованиях гораздо дороже ошибок конструирования, поэтому концентрация на этой части процесса влияет также на стоимость и график выполнения проекта.

Серьезным программистам я хочу сказать: уделяйте часть рабочего дня изучению и улучшению собственных методик. Хотя на программистов всегда давит какой-то будущий или прошедший крайний срок, методологическая абстракция — мудрая долговременная инвестиция.

*Роберт У. Флойд (Robert W. Floyd)*

Тот же принцип осознанного внимания к процессу справедлив и для проектирования. Перед началом стройки нужно создать надежный фундамент. Если вы начнете писать код, не завершив создания фундамента, изменить архитектуру системы будет труднее. Программисты будут дорожить уже написанным кодом. Трудно избавиться от плохого фундамента, если вы начали возводить на нем здание.

Процесс разработки важен в первую очередь потому, что качество должно встраиваться в ПО с самого начала. Это противоречит давней «мудрости», согласно которой вы можете написать сколь угодно плохой код и устранить все ошибки на этапе тестирования. Это заблуждение. Тестирование может только указать на отдельные дефектные области программы — оно не сделает вашу программу удобнее в использовании, более быстрой, компактной, удобочитаемой или расширяемой.

Преждевременная оптимизация — еще один изъян процесса разработки. Эффективный процесс подразумевает, что вы выполняете грубую работу в начале и тонкую в конце. Если бы вы были скульптором, вы придавали бы композиции общую форму и только потом начинали бы работать над отдельными деталями. Преждевременно выполняя оптимизацию, вы тратите время на полирование фрагментов кода, которые полировать не нужно. Вы можете отполировать фрагменты, которые и так достаточно малы и быстры, вы можете отполировать код, который позднее придется выбросить, и можете отказаться от выбрасывания плохого кода, потому что уже потратили время на его полировку. Всегда спрашивайте себя: «Делаю ли я это в правильном порядке? Что изменилось бы при изменении порядка?» Используйте адекватный процесс разработки и делайте это осознанно.

Низкоуровневые процессы тоже важны. Если вы пишете псевдокод, после чего создаете соответствующий ему реальный код, вы используете все преимущества нисходящего проектирования. Кроме того, так вы всегда будете иметь комментарии в коде, и потом вам не придется их писать.

**Перекрестная ссылка** Об итерации см. раздел 34.8.

Чтобы проанализировать общие и частные процессы, нужно сделать паузу и обратить внимание на то, как вы создаете ПО. Это время тратится с пользой. Программисты, руководствующиеся принципом «код — вот, что важно; сосредоточиваться надо на качестве кода, а не какого-то абстрактного процесса», поступают недальновидно, игнорируя горы экспериментальных и практических данных, свидетельствующих об обратном. Разработка ПО — процесс творческий. Если вы не понимаете сути этого процесса, ваш главный инструмент — мозг — часто работает вхолостую. Плохо организованный процесс заставляет тратить умственные ресурсы впустую, хорошо — позволяет извлечь из них максимальную выгоду.

## 34.3. Пишите программы в первую очередь для людей и лишь во вторую — для компьютеров

***Ваша программа***. *Лабиринт нелогичных заключений, замусоренный замысловатыми хитростями и нерелевантными комментариями. Сравните с МОЕЙ ПРОГРАММОЙ.*

***Моя программа***. *Бриллиант детерминированной точности, воплотивший в себе совершенное равновесие между компактностью и эффективностью кода, с одной стороны, и превосходной удобочитаемостью, дополненной полным набором комментариев, — с другой. Сравните с ВАШЕЙ ПРОГРАММОЙ.*

*Стэн Келли-Бутл (Stan Kelly-Bootle)*

Еще один лейтмотив данной книги — удобочитаемость кода. Общение с другими людьми — вот Святой Грааль самодокументирующегося кода.

Компьютерам все равно, насколько удобочитаем ваш код. Они вообще лучше читают двоичные команды, а не операторы высокоуровневых языков. Удобочитаемый код нужно писать для того, чтобы он был понятен людям. Читабельность положительно влияет на такие аспекты программы, как:

- понятность;
- легкость выполнения обзоров;
- уровень ошибок;
- удобство отладки;
- легкость изменения;
- время разработки — следствие всего вышеперечисленного;
- внешнее качество — следствие всего вышеперечисленного.

На заре программирования программа считалась частной собственностью программиста. Чтение чужой программы без спроса было не меньшей наглостью, чем чтение любовного письма. По сути этим и являлась программа — любовным письмом программиста компьютеру, полным интимных деталей, известных только партнерам. Программы заполнялись кличками домашних животных и сокращениями, столь популярными у влюбленных, живущих в благословенной абстракции и не замечающих больше никого во Вселенной. Всем остальным людям такие программы непонятны.

*Майкл Маркотти (Michael Marcotty)*

Удобочитаемый код писать ничуть не дольше, чем запутанный — по крайней мере в далекой перспективе. В работоспособности кода легче убедиться, если вы можете с легкостью прочесть то, что написали, и уже одного этого достаточно для работы над понятностью кода. Однако код читают и во время обзоров. И при исправлении ошибок. И при изменениях программы. Наконец, его читают, когда кто-то другой пытается использовать фрагмент вашего кода в похожей программе.

Работу над читабельностью кода не следует считать необязательной частью процесса разработки, и достижение удобства во время написания за счет удобства во время чтения едва ли можно признать удачным решением. Лучше один раз написать хороший код, чем раз за разом читать плохой.

«Что, если я просто пишу код для себя? Почему я должен делать его удобочитаемым?» Потому что через неделю-две вы будете работать над другой программой и подумаете: «Эй! Я уже написал этот класс на прошлой неделе. Я просто возьму мой старый протестированный отлаженный код и сэкономлю время». Если код неудобочитаем, желаю удачи — она вам пригодится.

Оправдывать написание нечитаемого кода тем, что над проектом работаете только вы, опасно. Помните, как ваша мать говорила: «Что, если твое лицо застынет в этом выражении?», а отец — «Ты играешь так, как тренируешься». Привычки влияют на всю вашу работу, и вы не можете изменить их, просто захотев этого, поэтому убедитесь, что используемые вами подходы, став привычками, вас устроят. Профессиональные программисты пишут удобочитаемый код, и точка.

Поймите также, что утверждение, согласно которому код может принадлежать одному программисту, спорно. В связи с этим Дуглас Комер провел полезное различие между личными и общими программами (Comer, 1981): «личные программы» используются только программистом. Никто другой их не использует. Никто другой их не изменяет. Никто даже не знает об их существовании. Такие программы обычно тривиальны и крайне редки. А вот «общие программы» используются или изменяются не только автором, но и кем-то еще.

Стандарты, которым должны соответствовать общие и личные программы, могут различаться. Личные программы могут быть плохо написаны и полны ограничений, и это не затронет никого, кроме автора. Общие программы нужно писать более внимательно: их ограничения следует документировать, а сами программы следует делать надежными и модифицируемыми. Опасайтесь превращения личных программ в общие, что происходит довольно часто. Преобразовать личную программу в общую нужно до того, как она поступит в обращение, и один из аспектов этого преобразования — улучшение читабельности кода.

Даже если вы думаете, что код будете читать только вы, в реальном мире высока вероятность того, что его придется изменять кому-то другому. Одно из исследований показало, что среднюю программу сопровождали 10 поколений программистов, пока она не была переписана (Thomas, 1984). Програм-

мисты, отвечающие за сопровождение, тратят от 50 до 60% времени, пытаясь понять код, и они по достоинству оценят ваши усилия, потраченные на его документирование (Parikh and Zvegintzov, 1983).

В предыдущих главах были рассмотрены методики улучшения удобочитаемости кода: грамотный выбор имен классов, методов и переменных, тщательное форматирование, сокращение объема методов, сокрытие сложных логических тестов в булевых функциях, присваивание промежуточных результатов сложных вычислений переменным и т. д. Никакая одна методика не сделает запутанную программу читабельной, однако сумма многих небольших улучшений будет существенной.

Если вы считаете, что какой-то код не нужно делать удобочитаемым, потому что никто другой никогда не будет иметь с ним дело, проверьте, не путаете ли вы причину и следствие.

## 34.4. Программируйте с использованием языка, а не на языке

Не ограничивайте свое мышление только теми концепциями, которые непосредственно поддерживаются языком. Самые лучшие программисты думают о том, что они хотят сделать, после чего определяют, как достичь этих целей при помощи инструментов программирования, имеющихся в их распоряжении.

Разумно ли создавать метод-член класса, не согласующийся с абстракцией класса, только потому, что он удобнее метода, обеспечивающего более высокую согласованность? Код должен как можно надежнее защищать абстракцию, формируемую интерфейсом класса. Не нужно использовать глобальные данные или операторы *goto* только потому, что их поддерживает язык. Вы можете отказаться от опасных возможностей и применять вместо них соглашения программирования, компенсирующие слабости языка. Выбирать самые очевидные пути — значит программировать *на* языке, а не *с использованием* языка; в программировании этот выбор эквивалентен вопросу: «Если Фредди спрыгнет с моста, прыгнете ли вы за ним?» Подумайте о своих целях и решите, как лучше всего достичь их, программируя *с использованием* языка.

Ваш язык не поддерживает утверждения? Напишите собственный метод *assert()*. Пусть он работает не совсем так, как встроенный *assert()*, но вы все же сможете задействовать большинство преимуществ этого подхода. Ваш язык не поддерживает перечисления или именованные константы? Прекрасно: определите собственные перечисления и именованные константы путем дисциплинированного использования глобальных переменных, дополненного ясными конвенциями именования.

В крайних случаях — особенно в новых технологических средах — инструменты бывают такими примитивными, что разработчикам приходится значительно изменять свой желательный подход к программированию. Иногда это заставляет уравновешивать желание программировать с использованием языка и несущественные сложности, возникающие из-за того, что особенности языка делают ваш подход слишком неуклюжим. Однако, попав в такие условия, вы сможете извлечь даже большую выгоду из соглашений программирования, помогающих избавиться от наиболее опасных возможностей среды. Как бы то ни было, обычно несоответ-

ствие между тем, что вы хотите сделать, и тем, что уже поддерживают инструменты, невелико и заставляет идти лишь на небольшие уступки.

## 34.5. Концентрируйте внимание с помощью соглашений

**Перекрестная ссылка** О полезности соглашений в контексте форматирования кода см. подразделы «Насколько важно хорошее форматирование?» и «Цели хорошего форматирования» раздела 31.1.

Набор соглашений — один из интеллектуальных инструментов управления сложностью. В предыдущих главах мы говорили о специфических конвенциях. В этом разделе я поясню на примерах общие преимущества соглашений.

Многие аспекты программирования в чем-то произвольны. Какой длины отступ делать перед циклом? Как форматировать комментарии? Как упорядочивать методы класса? Большинство подобных вопросов не имеет одного правильного ответа. Конкретный ответ на такой вопрос менее важен, чем его согласованность. Соглашения избавляют программистов от необходимости снова и снова отвечать на те же вопросы и принимать все те же произвольные решения. В проектах, реализуемых многими программистами, соглашения предотвращают замешательство, возникающее, когда разные программисты принимают разные решения.

Конвенция лаконично сообщает важную информацию. В случае соглашений именования один символ может обеспечить различие между локальными переменными, глобальными и переменными класса, регистр букв может указать на типы, именованные константы и переменные. Соглашения использования отступов могут охарактеризовать логическую структуру программы. Соглашения выравнивания указывают на связь операторов.

Соглашения защищают от известных опасностей. Вы можете задать соглашения для исключения применения опасных методик, для ограничения их использования в ситуациях, когда эти методики действительно нужны, или для компенсации их известных недостатков. Так, вы можете исключить опасность, запретив применение глобальных переменных или объединение нескольких команд в одной строке. Вы можете компенсировать слабости опасных методик, потребовав заключать сложные выражения в скобки или устанавливать указатели в *NULL* сразу после их освобождения для предотвращения «зависания» указателей.

Соглашения делают более предсказуемыми низкоуровневые задачи. Наличие соглашений обработки запросов памяти и обработки ошибок или соглашений ввода/вывода и создания интерфейсов классов добавляет в код выразительную структуру и делает его понятнее программистам, знающим об этих соглашениях. Как я уже говорил, одно из главных преимуществ устранения глобальных данных состоит в исключении потенциальных взаимодействий между разными классами и подсистемами. Программист, читающий код, примерно представляет, чего можно ожидать от локальных данных и данных класса, но едва ли он может определить, что изменение глобальных данных портит какой-то бит в коде подсистемы, находящейся на обратной стороне программы. Глобальные данные вносят в код неопределенность. Хорошие соглашения позволяют вам и людям, читающим ваш код, больше принимать как данное. Число деталей, которые нужно охватить, уменьшается, а это в свою очередь облегчает понимание программы.

Соглашения могут компенсировать недостатки языков. Если язык не поддерживает именованные константы (к таким языкам относятся Python, Perl, языки оболочек UNIX и т. д.), конвенция позволяет провести различие между переменными, допускающими и чтение, и запись, и переменными, служащими для эмуляции констант, предназначенных только для чтения. В качестве других примеров компенсирования недостатков языка при помощи соглашений можно назвать соглашения дисциплинированного использования глобальных данных и указателей.

В крупных проектах программисты иногда злоупотребляют конвенциями. Они создают так много стандартов и правил, что их запоминание само по себе становится полноценной работой. Но в небольших проектах программисты из-за плохого понимания достоинств разумных соглашений обычно впадают в другую крайность. Поймите подлинную ценность соглашений и извлекайте из них выгоду; используйте их для структурирования тех областей, которые страдают от недостатка структуры.

## 34.6. Программируйте в терминах проблемной области

Другим специфическим методом борьбы со сложностью является работа на максимально высоком уровне абстракции. Один способ достижения этой цели заключается в работе в терминах проблемы программирования, а не ее компьютерного решения.

Высокоуровневый код не должен включать подробных сведений о файлах, стеках, очередях, массивах, символах и подобных объектах, имеющих имена вроде *i*, *j* и *k*. Высокоуровневый код должен описывать решаемую проблему. Он должен быть наполнен описательными именами классов и вызовами методов, ясно характеризующими выполняемые действия, а не подробными сведениями о том, что файл открывается в режиме «только для чтения». Высокоуровневый код не должен быть загроможден комментариями, гласящими, что «здесь переменная *i* представляет индекс записи из файла о сотрудниках, а чуть позже она используется для индексации файла счетов клиентов».

Это неуклюжая методика программирования. На самом высоком уровне программы не нужно знать, что данные о сотрудниках представлены в виде записей или хранятся в файле. Информацию, относящуюся к этому уровню детальности, надо скрыть. На самом высоком уровне вы не должны иметь понятия о том, как хранятся данные. Вы не должны читать комментарии, объясняющие роль переменной *i* и то, что она используется с двойной целью. Вместо этого вы должны видеть две переменные с выразительными именами, такими как *employeeIndex* и *clientIndex*.

### Разделение программы на уровни абстракции

Очевидно, что на некотором уровне надо работать и в терминах реализации, но вы можете изолировать эти части программы от частей, разработанных в терминах проблемной области. Проектируя программу, обдумайте уровни абстракции (рис. 34-1).

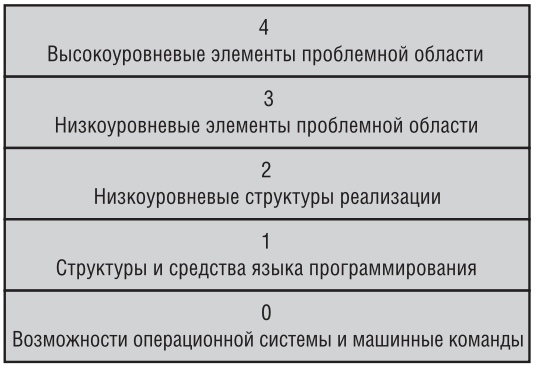



***Рис. 34-1.*** *Программа может быть разделена на несколько уровней абстракции. Удачное проектирование позволяет программистам проводить значительную часть времени, сосредоточившись только на верхних уровнях, игнорируя более низкие уровни*

### Уровень 0: возможности операционной системы и машинные команды

Если вы программируете на высокоуровневом языке, можете не беспокоиться о самом низком уровне: язык позаботится об этом автоматически. Если же вы используете низкоуровневый язык, попробуйте создать ради своего удобства более высокие уровни, хотя многие программисты этого не делают.

### Уровень 1: структуры и средства языка программирования

Структуры языка программирования — это элементарные типы данных, управляющие структуры и т. д. Кроме того, большинство популярных языков снабжено дополнительными библиотеками, предоставляют доступ к вызовам ОС и т. д. Вы используете эти структуры и средства естественным образом, так как программировать без них невозможно. Многие программисты никогда не поднимаются выше этого уровня абстракции, чем значительно осложняют себе жизнь.

### Уровень 2: низкоуровневые структуры реализации

Низкоуровневые структуры реализации относятся к чуть более высокому уровню, чем структуры, предоставляемые самим языком. В большинстве своем это операции и типы данных, которые вы изучали в вузе: стеки, очереди, связные списки, деревья, индексированные файлы, последовательные файлы, алгоритмы сортировки, поиска и т. д. Если вы будете писать программу полностью на этом уровне, вам придется работать со слишком большим числом деталей, чтобы победить в битве со сложностью.

### Уровень 3: низкоуровневые элементы проблемной области

На этом уровне вы имеете дело с примитивами, нужными для работы в терминах проблемной области. Это клей, скрепляющий нижележащие структуры компьютерных наук и высокоуровневый код проблемной области. Чтобы писать код на этом уровне, вы должны определить словарь проблемной области и создать строительные блоки, годные для решения поставленной задачи. Во многих приложениях этим уровнем является уровень бизнес-объектов или уровень сервисов.

В качестве элементов словаря и строительных блоков данного уровня выступают классы. Возможно, эти классы слишком примитивны, чтобы их можно было задействовать для решения проблемы непосредственно на этом уровне, однако они формируют каркас, на основе которого можно решить проблему, используя классы более высокого уровня.

#### Уровень 4: высокоуровневые элементы проблемной области

Этот уровень формирует абстракцию, позволяющую работать с проблемой в ее собственных терминах. Код, написанный на этом уровне, должен быть частично понятен даже людям, далеким от программирования — возможно, и вашим заказчикам. Он будет слабо зависеть от специфических аспектов языка программирования, потому что вы будете использовать для работы над проблемой собственный набор средств. Так что на этом уровне ваш код больше зависит от средств, созданных вами на уровне 3, чем от возможностей языка.

Детали реализации уже должны быть скрыты на два уровня ниже — на уровне структур компьютерных наук, чтобы изменения оборудования или ОС совсем не влияли на этот уровень. Выразите в программе на этом уровне пользовательское представление о мире, потому что когда программа изменяется, она изменяется в терминах пользователя. Изменения проблемной области будут сильно влиять на этот уровень, но вы сможете легко адаптировать к ним программу, создавая новую версию на основе строительных блоков предыдущего уровня.

Многие программисты находят полезным дополнение этих концептуальных уровней другими, перпендикулярными «уровнями». Например, типичная трехуровневая архитектура пересекает описанные выше уровни, предоставляя дополнительные средства интеллектуального управления аспектами проектирования и кодом.

### Низкоуровневые методики работы в проблемной области

Даже не выработав полного архитектурного подхода к словарю проблемной области, вы можете использовать многие методики этой книги для работы в терминах проблемы реального мира, а не ее компьютерного решения.

- Используйте классы для реализации структур, значимых в проблемной области.
- Скрывайте информацию о низкоуровневых типах данных и деталях их реализации.
- Используйте именованные константы для документирования смысла строк и численных литералов.
- Присваивайте промежуточным переменным промежуточные результаты вычислений с целью документирования этих результатов.
- Используйте булевы функции для пояснения сложных булевых тестов.

## 34.7. Опасайтесь падающих камней

Программирование не является ни полностью искусством, ни полностью наукой. В своей обычной форме оно представляет собой «мастерство», занимающее промежуточное место между искусством и наукой. В лучшем случае это инженерная дисциплина, основанная на синергической интеграции науки и искусства

(McConnell, 2004). Чем бы программирование ни было — искусством, ремеслом или инженерной дисциплиной, создание работающей программы требует изрядной доли рассудительности. А для этого нужно обращать внимание на широкий диапазон предупреждающих знаков — тонких намеков на проблемы в вашей программе. Предупреждающие знаки в программировании указывают на возможные проблемы, но обычно они не настолько очевидны, как дорожный знак, предупреждающий о камнепадах.

Слова «Это по-настоящему хитрый код» обычно предупреждают о том, что код плох. «Хитрый» код — это другое название «плохого» кода. Если код кажется вам хитроумным, подумайте, не переписать ли его, чтобы он таким не был.

Класс, число ошибок в котором превышает средний уровень, — тоже предупреждающий знак. Несколько классов, подверженных ошибкам, обычно оказываются самой дорогой частью программы. Если число ошибок в каком-то классе превышает средний уровень, такая ситуация, вероятно, сохранится и в будущем. Подумайте о том, чтобы переписать его.

Если бы программирование было наукой, с каждым предупреждающим знаком был бы связан конкретный, хорошо определенный способ исправления проблемы. Но так как программирование еще и мастерство, предупреждающие знаки просто указывают на проблемы, которые вы должны рассмотреть. Переписывать хитроумный код или улучшать класс, подверженный ошибкам, нужно не всегда.

Как аномальное число дефектов в классе предупреждает о низком качестве класса, так и аномальное число дефектов в программе свидетельствует о неэффективности процесса разработки. Хороший процесс не привел бы к получению дефектного кода. Он включил бы проверку архитектуры, за которой последовали бы обзоры архитектуры, проектирование с обзорами проекта и кодирование с обзорами кода. Ко времени тестирования кода большинство ошибок было бы устранено. Для достижения высочайшей производительности труда нужно работать не просто усердно, но и разумно. Большой объем отладки предупреждает о том, что программисты не работают разумно. Написать большой фрагмент кода за день и потратить две недели на его отладку — это и есть неразумная работа.

Метрики проектирования также могут быть предупреждающими знаками. Большинство таких метрик — это эвристические правила, характеризующие качество проектирования. Если класс содержит более семи членов, это не всегда означает, что он плохо спроектирован, но предупреждает о том, что класс сложен. Более 10 точек принятия решения в методе, более трех уровней логической вложенности, необычно большое число переменных, высокая степень сопряжения одного класса с другими или низкий уровень внутреннего сопряжения класса или метода — все это предупреждающие знаки. Они не всегда означают, что класс спроектирован плохо, но наличие любого из них должно заставлять вас взглянуть на класс скептически.

Любой предупреждающий знак должен заставить вас сомневаться в качестве программы. Как говорит Чарльз Саундерс Пирс (Charles Saunders Peirce), «сомнение — это неловкое и неприятное состояние, от которого мы стараемся освободиться, перейдя в состояние убежденности». Рассматривайте предупреждающие знаки как «причины сомнения», побуждающие искать более приятное состояние убежденности.

Если вы ловите себя на том, что работаете над повторяющимся кодом или вносите похожие изменения в несколько фрагментов, вам следует почувствовать себя «неловко и неприятно», усомнившись в том, что управление было адекватно централизовано в классах или методах. Если из-за проблем с отдельным классом вы не можете с легкостью создать тестовые леса, вы должны почувствовать сомнение и спросить себя, не слишком ли сильно класс сопряжен с другими классами. Если вы не можете повторно использовать код в других программах, потому что некоторые классы слишком взаимозависимы, это также предупреждает о чересчур сильном сопряжении классов.

Погрузившись в детали программы, обращайте внимание на предупреждающие знаки, указывающие на то, что часть проекта программы недостаточно хорошо определена для кодирования. Трудности при написании комментариев, именовании переменных и декомпозиции проблемы на связные классы с ясными интерфейсами — все это говорит о том, что нужно более тщательно выполнить проектирование перед началом кодирования. Невыразительные имена и сложности при описании фрагментов кода лаконичными комментариями — другие признаки проблем. Если вы хорошо представляете проект программы, реализовать низкоуровневые детали будет легко.

Обращайте внимание на признаки того, что программу трудно понять. Об этом свидетельствуют любые неудобства. Если это трудно для вас, для будущих программистов это окажется еще труднее. Они по достоинству оценят ваши дополнительные усилия по улучшению кода. Если вы разгадываете код, а не читаете его, он слишком сложен. Если он сложен, он неверен. Сделайте его проще.

Если вы хотите воспользоваться всеми преимуществами предупреждающих знаков, программируйте так, чтобы создать собственные предупреждения. Это полезно потому, что, даже если вам известны предупреждающие знаки, их на удивление легко упустить из виду. Гленфорд Майерс исследовал исправления дефектов и обнаружил, что самой частой причиной плохого обнаружения ошибок была простая невнимательность. Ошибки были видны в результатах тестов, но программисты их не замечали (Myers, 1978b).

Программируйте так, чтобы ошибки было трудно не заметить. Примером этого может служить установка указателей в *NULL* после их освобождения, чтобы их ошибочное использование вызывало безобразные проблемы. Освобожденный указатель даже после освобождения может указывать на корректную область памяти. Установка указателя в *NULL* гарантирует, что он будет указывать на некорректный адрес, благодаря чему ошибку будет сложнее не заметить.

Предупреждения компилятора являются буквальными предупреждающими знаками, которые часто упускают из виду. Если ваша программа генерирует предупреждения или ошибки, устраните их. Невелика вероятность того, что вы заметите тонкие предупреждающие знаки, если вы игнорируете те, на которых прямо написано «ПРЕДУПРЕЖДЕНИЕ».

Почему внимание к интеллектуальным предупреждающим знакам особенно важно при разработке ПО? Качество мышления, воплощаемое в программе, во многом определяет качество самой программы, поэтому внимание к предупреждениям о качестве мышления напрямую влияет на итоговый продукт.

## 34.8. Итерируйте, итерируйте и итерируйте

Итерация полезна на многих этапах разработки ПО. При разработке первоначальной спецификации системы вы составляете с заказчиком несколько версий требований, пока не достигнете согласия. Это итеративный процесс. Гибкий процесс разработки, предусматривающий создание и поставку системы по частям, тоже итеративен. Прототипирование, имеющее целью быструю и дешевую разработку предварительных вариантов решений, — еще одна форма итерации. Итеративная выработка требований, наверное, не менее важна, чем любой другой аспект процесса разработки ПО. Проекты завершаются неудачей потому, что разработчики приступают к решению проблемы, не изучив альтернативных вариантов. Итерация позволяет лучше узнать систему перед ее созданием.

Оценки сроков при первоначальном планировании проекта могут сильно различаться в зависимости от используемой методики оценки (см. главу 28). Итеративный подход дает более точную оценку, чем единственная методика.

Проектирование ПО — процесс эвристический и, как все эвристические процессы, допускает итеративные ревизии и улучшения. Правильность ПО обычно проверяется, а не доказывается, а значит, оно тестируется и разрабатывается итеративно, пока на все вопросы не будут получены правильные ответы. И высокоуровневые, и низкоуровневые попытки проектирования следует повторять. Первая попытка может привести к работоспособному решению, но едва ли оно окажется наилучшим. Неоднократное применение разных подходов позволяет узнать о проблеме много такого, что ускользает при использовании единственного подхода.

Идея итерации снова вступает в игру при оптимизации кода. Как только ПО доведено до работоспособного состояния, вы можете переписать небольшие фрагменты кода и добиться значительного повышения общей производительности системы. Однако многие попытки оптимизации ухудшают, а не улучшают код. Это неинтуитивный процесс, и некоторые методики, которые будто должны сделать систему более компактной и быстрой, на самом деле увеличивают ее и замедляют. Неопределенность результатов любого вида оптимизации заставляет пробовать один вид, оценивать быстродействие и пробовать другой. Если от устранения узкого места зависит достижение нужной производительности, вы можете оптимизировать код несколько раз, и более поздние попытки могут оказаться успешнее первой.

Весь процесс разработки охвачен обзорами, что встраивает итерацию в любой этап, на котором они проводятся. Цель обзора — проверка качества работы, выполненной на данный момент. Если система не проходит обзора, она возвращается на переработку. Если обзор проходит успешно, дальнейшая итерация не нужна.

Говорят, инженерное дело — это умение сделать за 10 центов то, что любой может сделать за доллар. Итерация на более поздних этапах — это трата двух долларов на то, что любой может сделать за один. Фред Брукс советует «планировать выбросить один вариант программы, потому что это придется сделать в любом случае» (Brooks, 1995). Хитрость разработки ПО в том, чтобы создавать выбрасываемые части как можно быстрее и дешевле — это и есть суть итерации на ранних этапах разработки.

# 34.9. И да отделена будет религия от разработки ПО

Религия проявляется в разработке ПО по-разному: как догматичное следование единственной методике проектирования, как непоколебимая убежденность в превосходстве отдельного стиля форматирования или комментирования или как рьяный отказ от глобальных данных. Как бы то ни было, это всегда неуместно.

## Оракулы программирования

Увы, догматиками бывают и некоторые из наиболее уважаемых людей в нашей отрасли. Конечно, инновационные методики нужно разглашать, чтобы их могли попробовать практикующие разработчики. До того как эффективность методик можно будет полностью доказать или опровергнуть, их нужно испытать. Распространение результатов исследований на практикующих специалистов называется «передачей технологий» и играет важную роль в совершенствовании способов разработки ПО. Однако распространение новой методологии нужно отличать от продажи чудодейственного средства от всех болезней программирования. Передача технологий — это совсем не то, чем занимаются торговцы догматичными методологиями, пытающиеся убедить вас в том, что их высокотехнологичная новинка решит все проблемы всех программистов во Вселенной. Забудьте обо всем, что вы изучили, потому что эта великолепная новая методика повысит производительность вашего труда в два раза во всех областях!

**Перекрестная ссылка** О том, как руководителям следует подходить к религии программирования, см. подраздел «Вопросы религии» раздела 28.5.

Вместо чрезмерного увлечения модными штучками используйте смесь методик. Экспериментируйте с интересными новыми методиками, но делайте ставку на старые и надежные.

## Эклектизм

Слепая вера в одну методику подавляет возможность выбора, нужную для обнаружения наиболее эффективных решений проблем. Будь разработка ПО детерминированным алгоритмическим процессом, вы могли бы следовать жесткой методологии. Однако разработка ПО — не детерминированный, а эвристический процесс, а значит, жесткие подходы редко приводят к успеху. Например, при проектировании иногда хорошо работает нисходящая декомпозиция, но иногда лучшим вариантом будет объектно-ориентированный подход, восходящая композиция или подход, основанный на структурах данных. Вы должны быть готовы к испытанию нескольких подходов, зная, что некоторые окажутся неэффективными, а какие-то приведут к успеху, но не зная заранее, какие из них какие. Вам следует быть эклектичным.

**Перекрестная ссылка** О различии между алгоритмическим и эвристическим подходами см. раздел 2.2, об эклектизме при проектировании — подраздел «Используйте итерацию» раздела 5.4.

Приверженность одной методике вредна и тем, что она заставляет подгонять проблему под решение. Выбирая методику решения до полного понимания проблемы, вы слишком спешите. Чрезмерное ограничение набора возможных решений может исключить из области видимости наиболее эффективное решение.

Любая новая методология поначалу вызывает дискомфорт, и совет избегать религии в программировании не подразумевает, что от новой методики следует отказаться при возникновении небольшой проблемы. Пересмотрите новую методику, но не забывайте пересматривать и старые методики.

**Перекрестная ссылка** О метафоре инструментария см. подраздел «Применение методов разработки ПО: интеллектуальный инструментарий» раздела 2.3.

С позиций эклектизма полезно подходить и к методикам, описанным в этой книге, и к методикам, рассматриваемым в других источниках. Некоторые подходы, представленные мной, имеют улучшенные альтернативы, и вы не можете использовать их одновременно. Вы должны выбрать для конкретной проблемы один или другой подход. Рассматривайте методики как инструменты и выбирайте наиболее подходящую для работы. В большинстве случаев выбор инструмента не играет особой роли. Вы можете использовать торцовый ключ, плоскогубцы или разводной ключ. Однако иногда выбор очень важен, поэтому вам следует всегда делать выбор со всей тщательностью. Разработка предполагает нахождение компромисса между конкурирующими методиками. Вы не сможете достичь компромисса, если преждевременно ограничите выбор единственным инструментом.

Метафора инструментария полезна потому, что она делает абстрактную идею эклектизма конкретной. Если бы вы строили дом и ваш приятель Симпл Саймон всегда работал только плоскогубцами, отказываясь от торцового или разводного ключей, вы, вероятно, подумали бы, что он ведет себя странно, потому что не использует все инструменты, имеющиеся в его распоряжении. То же верно и при разработке ПО. На высоком уровне у вас есть альтернативные методики проектирования. На более детальном уровне вы можете выбрать для представления конкретной сущности один из нескольких типов данных. На еще более детальном уровне вы можете выбирать схемы форматирования и комментирования кода, именования переменных, определения интерфейсов классов и передачи параметров в методы.

Догматизм конфликтует с эклектичным подбором инструментов к конструированию ПО. Она несовместима с психологической установкой, необходимой для создания высококачественных программ.

## Экспериментирование

Эклектизм тесно связан с экспериментированием. Вам нужно экспериментировать на всем протяжении процесса разработки, но непреклонность подавляет этот импульс. Чтобы экспериментирование было эффективным, вы должны охотно изменять свои убеждения на основе результатов экспериментов — иначе экспериментирование становится пустой тратой времени.

Многие негибкие подходы к разработке ПО основаны на страхе допустить ошибку, но нет ошибки серьезнее, чем глобальное стремление избежать ошибок. Проектирование — это процесс тщательного планирования мелких ошибок с целью предотвращения крупных. Экспериментирование при разработке ПО позволяет узнать, эффективен ли тот или иной подход, — сам эксперимент является успехом, если он решает проблему.

Экспериментирование уместно на стольких же уровнях, что и эклектизм. На каждом уровне, предоставляющем возможность эклектичного выбора, вы, вероятно, можете провести соответствующий эксперимент для определения оптимального подхода. На уровне разработки архитектуры эксперимент может заключаться в проектировании архитектуры ПО с использованием трех разных подходов. На уровне детального проектирования эксперимент может состоять в детализации высокоуровневой архитектуры с использованием трех разных подходов к низкоуровневому проектированию. На уровне языка программирования экспериментом может быть написание небольшой экспериментальной программы для изучения плохо знакомых вам аспектов языка. Эксперимент может заключаться в оптимизации фрагмента кода и оценке того, действительно ли он стал меньше или быстрее. На уровне общего процесса разработки ПО эксперимент может заключаться в сборе данных о качестве и производительности труда, отвечающих на вопрос, действительно ли инспекции позволяют обнаружить больше ошибок, чем анализ кода.

Суть сказанного в том, что вы должны быть восприимчивы ко всем аспектам разработки ПО. Вы должны разобраться и в используемом процессе разработки, и в создаваемой системе. Непредвзятое экспериментирование и религиозное следование предопределенному подходу исключают друг друга.

## Ключевые моменты

- Главная цель программирования — управление сложностью.
- Процесс программирования оказывает большое влияние на итоговый продукт.
- Групповое программирование является в большей степени общением с другими людьми, а не с компьютером. Индивидуальное программирование — это в первую очередь общение с самим собой, а не с компьютером.
- При неадекватном использовании конвенция программирования может оказаться лекарством, причиняющим больше вреда, чем болезнь; при грамотном — конвенция добавляет ценную структуру в среду разработки, помогает управлять сложностью и облегчает общение.
- Программирование в терминах проблемы, а не решения помогает управлять сложностью.
- Внимание к интеллектуальным предупреждающим знакам вроде сомнения особенно важно в программировании, потому что программирование — почти исключительно умственная деятельность.
- Чем больше внимания итерации вы уделяете на конкретном этапе разработки, тем лучше будет результат этого этапа.
- Догматичные методологии и разработка высококачественного ПО исключают друг друга. Заполняйте свой интеллектуальный инструментарий альтернативными подходами к программированию и улучшайте навык выбора инструмента, лучше всего подходящего для работы.

ГЛАВА 35

# Где искать дополнительную информацию

http://cc2e.com/3560

**Содержание**



**Связанные темы**

- Web-ресурсы: *www.cc2e.com*

Если вы так далеко продвинулись в чтении этой книги, то уже знаете, как много написано о практике эффективной разработки ПО. Доступной информации гораздо больше, чем можно представить. Все ошибки, которые вы делаете сейчас, люди уже сделали до вас. И если вы не хотите стать мальчиком для битья, то предпочтете читать их книги, чтобы не повторять их ошибки и не изобретать велосипед.

Поскольку в этой книге упоминаются сотни других книг и статей по разработке ПО, трудно сказать, с чего начать чтение. Библиотека программиста включает в себя несколько видов информации. Основу составляют книги, объясняющие фундаментальные концепции эффективного программирования. В других более подробно рассматриваются технические, управленческие, интеллектуальные проблемы программирования. Подробные справочники по языкам, операционным системам, средам разработки и аппаратному обеспечению содержат информацию, полезную для конкретных проектов.

http://cc2e.com/3581

Обычно книги последней категории представляют интерес в рамках одного проекта; в большей или меньшей степени они являются временными и здесь не обсуждаются. Что касается других категорий, полезно иметь библиотечку, в которой основные виды деятельности по разработке ПО обсуждаются более глубоко: книги по выработке требований, конструированию, проектированию, управлению, тестированию и т. д. В следующих разделах подробно описываются ресурсы по конструированию, а

затем предлагается обзор материалов, относящихся к других вопросам разработки ПО. В разделе 35.4 все ресурсы систематизированы и дан список литературы для разработчика ПО.

# 35.1. Информация о конструировании ПО

http://cc2e.com/3588

Сначала я писал эту книгу, потому что не мог найти публикаций, в которых бы всесторонне обсуждались вопросы конструирования ПО. За годы, прошедшие с момента первого издания, появилось несколько хороших книг.

Книга «Pragmatic Programmer» (Hunt and Thomas, 2000) заостряет внимание на деятельности, непосредственно связанной с кодированием, включая тестирование, отладку, использование утверждений и т. д. Не вдаваясь в детали кода, она знакомит с многочисленными правилами создания хорошей программы

В книге Джона Бентли «Programming Pearls», 2-е изд. (Bentley, 2000) обсуждаются искусство и наука проектирования ПО. Книга состоит из отлично написанных очерков, в которых глубокое понимание приемов эффективного конструирования сочетается с увлеченностью этой темой. В каждой своей программе я использую что-нибудь полезное из очерков Бентли.

**Перекрестная ссылка** Об экономике экстремального и быстрого программирования см. cc2e.com/3545.

Работа Кента Бека «Extreme Programming Explained: Embrace Change» (Beck, 2000) определяет подход к разработке ПО, опирающийся на конструирование. Как разъясняется в разделе 3.1, утверждения автора относительно экономики экстремального программирования не подтверждаются результатами исследований, но многие из его рекомендаций полезны при конструировании независимо от того, что применяется: экстремальное программирование или другой подход.

В более специализированном труде Стива Мэгуаера «Writing Solid Code — Microsoft's Techniques for Developing Bug-Free C Software» (Maguire, 1993) рассматривается практика конструирования ПО в масштабных коммерческих приложениях. Книга главным образом отражает опыт автора, приобретенный при работе над приложениями Microsoft Office. В ней также обсуждаются технические приемы программирования на языке C. Автор в целом не затрагивает вопросы объектно-ориентированного программирования, но большинство рассматриваемых тем представляет интерес в любой среде.

Брайан Керниган и Роб Пайк написали другую узкоспециализированную книгу — «The Practice of Programming» (Kernighan and Pike, 1999), в которой уделяют внимание будничным, но жизненно необходимым моментам в работе программиста, практическим аспектам программирования, сокращая разрыв между академическими знаниями в информатике и практическими навыками. В книге, предполагающей знание языков C/C++, обсуждается стиль программирования, проектирование, отладка и тестирование.

http://cc2e.com/3549

Книга Сузан Ламмерс «Programmers at Work» (Lammers, 1986) не переиздавалась, но ее стоит найти. Она содержит интервью с высококвалифицированными программистами, в которых раскрываются их личности, профессиональные привычки и философия программирования. Среди тех, кто дает интервью, такие светила, как Билл Гейтс (основатель Microsoft), Джон Уорнок (основатель Adobe), Энди Херцфельд (ведущий разработчик ОС Macintosh), Батлер Лэмпсон (старший инженер компании DEC, теперь работающий в Microsoft), Уэйн Рэтлифф (изобретатель dBase), Дан Бриклин (изобретатель VisiCalc), и дюжина других.

## 35.2. Не связанные с конструированием темы

Помимо основной литературы, рассмотренной в предыдущем разделе, здесь представлены книги, не имеющие прямого отношения к теме конструирования ПО.

### Обзорный материал

http://cc2e.com/3595

Ряд книг позволяет взглянуть на процесс разработки ПО с разных сторон.

Роберт Л. Гласс в работе «Facts and Fallacies of Software Engineering» (Glass, 2003) предлагает интересную трактовку традиционных представлений о возможном и невозможном в разработке ПО. Книга содержит много указателей на дополнительные ресурсы.

В книге «Professional Sofware Development» (2004) я рассматриваю практику разработки ПО в ее современном виде и размышляю о том, какой она должна быть в идеале.

Книга «The Swebok: Guide to the Software Engineering Body of Knowledge» (Abran, 2001) разделяет на составляющие прикладную отрасль знаний, которая занимается оптимизацией и повышением эффективности разработки ПО, и погружает в детали конструирования ПО. Именно этот труд показывает, что в нашей области значительно больше достижений, чем можно представить.

Книга Джеральда Вейнберга «The Psychology of Computer Programming» (Weinberg, 1998) наполнена замечательными анекдотами о программировании. Она несколько устарела, так как написана во времена, когда программирование считалось единственным аспектом создания ПО. Совет, прозвучавший при первом обсуждении этой книги в «ACM Computing Reviews», актуален и сегодня:

«Каждый руководитель программистов должен иметь собственный экземпляр этой книги. Он должен читать ее, хранить у сердца, следовать ее наставлениям, а затем оставлять на своем столе, чтобы подчиненные могли ее украсть. Взамен украденным экземплярам следует подбрасывать новые, пока все не успокоятся». (Weiss, 1972).

Если вы не смогли найти «The Psychology of Computer Programming», ищите «The Mythical Man-Month» (Brooks, 1995) или «PeopleWare» (DeMarco and Lister, 1999). Обе книги убеждают в том, что программирование — прежде всего результат человеческой деятельности и лишь во вторую очередь — нечто связанное с компьютерами.

И, наконец, превосходный обзор проблем разработки ПО сделан в «Software Creativity» (Glass, 1995). Эту книгу следовало бы считать прорывом в отношении к программированию как к творчеству. Такую же роль сыграла книга «PeopleWare» в вопросах коллективной разработки. Гласс сопоставляет творчество и дисциплину, теорию и практику, эвристику и методологию, процесс и результат, а также многие другие противоположные и в то же время взаимосвязанные понятия из области ПО. Спустя годы после обсуждения этой книги со своими коллегами я понял, в чем ее противоречие. Дело в том, что Гласс — редактор, а не автор очерков, вошедших в книгу, и поэтому у некоторых читателей возникает ощущение незавершенности. Тем не менее я до сих пор требую от каждого разработчика моей компании прочесть ее. Книга не переиздается, ее трудно найти, но затраченные на поиск усилия не окажутся напрасными.

## Общие вопросы разработки ПО

Каждый практикующий программист должен иметь руководства высокого уровня по вопросам разработки ПО. Такие книги скорее очерчивают общую картину методологии, а не отдельные, характерные детали этой картины. Они знакомят с эффективными методиками разработки и предлагают сжатые описания специальных технических приемов. Сокращенные описания недостаточно подробны, чтобы обучаться этим приемам, но, с другой стороны, для этого понадобилась бы книга, содержащая несколько тысяч страниц. И все же полученная информация позволяет выбрать необходимые для дальнейшей работы технологии и научиться применять их в комплексе.

Роджер С. Прэссман в книге «Software Engineering: A Practitioner's Approach», 6-е изд. (Pressman, 2004) дает стройное толкование таким терминам, как требования, проектирование, качественная проверка данных и управление. На 900 страницах этой книги уделено мало внимания практике программирования, но это не является существенным недостатком, особенно если у вас есть книга по конструированию, подобная той, что вы читаете сейчас.

Шестое издание книги Иэна Соммервилла «Software Engineering» (Sommerville, 2000) сопоставимо с книгой Прэссмана и также содержит прекрасный обзор процесса разработки ПО.

## Другие аннотированные библиографии

Хорошие библиографии по вычислительной технике — редкость. Далее рассматриваются те, что заслуживают внимания.

http://cc2e.com/3502

«ACM Computing Reviews» — специализированное издание Ассоциации по вычислительной технике (ACM), представляющие собой обзор книг, охватывающих все аспекты знаний о компьютерах и программировании. Разветвленная схема размещении материалов в этом издании позволяет без труда найти книги по интересующим темам. Для получения информации об этой публикации и о членстве в ACM пишите по адресу: ACM, PO Box 12114, Church Street Station, New York, NY 10257.

Информация компании Construx Software о технологиях обучения и повышения профессионального уровня содержит-

http://cc2e.com/3509

ся на сайте *www.construx.com/ladder/*. Здесь представлены списки источников, рекомендованные для разработчиков, тестировщиков и руководителей в области ПО.

# 35.3. Периодические издания

## Журналы для начинающих программистов

Журналы этого уровня часто продаются в газетных киосках.

http://cc2e.com/3516

«Software Development», *www.sdmagazine.com*. Этот журнал освещает общие вопросы программирования, а не проблемы использования конкретных сред. Статьи довольно хорошо написаны; журнал содержит обзор новых продуктов.

http://cc2e.com/3523

«Dr. Dobb's Journal», *www.ddj.com*. Этот журнал ориентирован на активных программистов. Его статьи тяготеют к детализации материала и содержат огромное количество кода.

Если не удается найти эти журналы в киоске, издатели могут прислать вам поощрительный экземпляр, а большинство статей доступно в сети.

## Журналы для продвинутых программистов

Как правило, в киосках этих изданий не найти. Вам нужно обратиться в крупную университетскую библиотеку или подписаться на них.

http://cc2e.com/3530

«IEEE Software», *www.computer.org/software/*. В этом журнале, выходящем раз в два месяца, обсуждаются вопросы конструирования и проектирования ПО, выработки требований и управления, а также другие темы передовых технологий создания ПО. Его цель — «создать организацию, объединяющую ведущих специалистов в области ПО». В 1993 году я писал, что это «самый полезный журнал, на который может подписаться программист». Позже мне довелось занимать в нем пост главного редактора, и я по-прежнему убежден, что это лучшее периодическое издание, предназначенное для серьезной практической деятельности.

http://cc2e.com/3537

«IEEE Computer», *www.computer.org/computer/*. Этот ежемесячный журнал является лидером публикаций компьютерной ассоциации IEEE (Института инженеров по электротехнике и электронике). Он охватывает широкий круг вопросов по вычислительной технике, использует жесткие критерии при отборе статей, обеспечивая высокое качество публикуемых материалов. Так как интересы издания довольно обширны, в нем вы, пожалуй, найдете меньше полезных статей, чем в «IEEE Software».

http://cc2e.com/3544

«Communications of the ACM», *www.acm.org/cacm/*. Это одно из старейших и наиболее уважаемых компьютерных изданий имеет огромные полномочия, позволяющие расширять границы такого предмета, как компьютерология, который сегодня включает в себя значительно больше терминов и определений, чем несколько лет назад. Как и в случае с «IEEE Computer», диапазон вопросов издания довольно широк, и темы многих статей, возможно, не отвечают вашим интересам. Журнал отличает дух академизма, что имеет как хорошую, так и плохую стороны. К плохой следует

отнести путаный, сложный для восприятия стиль некоторых авторов. Хорошая проявляется в том, что он содержит наиболее передовую информацию, которая не попадет с годами в низкопробные издания.

## Специализированные публикации

Часть периодики более глубоко рассматривает специальные темы.

### Профессиональные издания

http://cc2e.com/3551

Компьютерная ассоциация IEEE выпускает специализированные журналы по следующим вопросам: проектирование ПО, безопасность и защита информации, компьютерная графика и анимация, Интернет-технологии, мультимедиа, интеллектуальные системы, история вычислительной техники и др. Подробнее см. по адресу *www.computer.org*.

http://cc2e.com/3558

ACM также публикует специальные выпуски по таким разделам, как искусственный интеллект, взаимодействие человека и компьютера, базы данных, встроенные системы, графика, языки программирования, ПО для математических задач, построение сетей, проектирование ПО и др. Подробнее см. по адресу *www.acm.org*.

### Популярные коммерческие издания

http://cc2e.com/3565

Темы следующих журналов следуют из названий.

«The C/C++ Users Journal», *www.cuj.com*.

«Java Developer's Journal», *www.sys-con.com/java/*.

«Embedded Systems Programming», *www.embedded.com*.

«Linux Journal», *www.linuxjournal.com*.

«Unix Review», *www.unixreview.com*.

«Windows Developer's Network», *www.wd-mag.com*.

# 35.4. Список литературы для разработчика ПО

http://cc2e.com/3507

В этом разделе предлагается план чтения, позволяющий разработчику ПО стать полноценным специалистом в моей компании, Construx Software. Этот план является предметным базовым курсом и рассчитан на тех, кто уже работает в области ПО и желает уделить особое внимание теме разработки. Наше методическое пособие дает возможность перестраивать базовую программу с учетом индивидуальных интересов. В рамках программы Construx предусматривается также повышение квалификации и обмен опытом работы по конкретным вопросам.

## Вводный курс

Чтобы пройти «начальный» уровень по методике Construx, необходимо прочесть следующие книги.

Adams, James L. *Conceptual Blockbusting: A Guide to Better Ideas*, 4th ed. Cambridge, MA: Perseus Publishing, 2001.

Bentley, Jon. *Programming Pearls*, 2d ed. Reading, MA: Addison-Wesley, 2000.

Glass, Robert L. *Facts and Fallacies of Software Engineering*. Boston, MA: Addison-Wesley, 2003.

McConnell, Steve. *Software Project Survival Guide*. Redmond, WA: Microsoft Press, 1998.

McConnell, Steve. *Code Complete*, 2d ed. Redmond, WA: Microsoft Press, 2004.

## Курс практической подготовки

Далее предлагаются дополнительные источники, которые должен освоить программист, чтобы соответствовать «промежуточному» уровню в Construx.

Berczuk, Stephen P. and Brad Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Boston, MA: Addison-Wesley, 2003.

Fowler, Martin. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3d ed. Boston, MA: Addison-Wesley, 2003.

Glass, Robert L. *Software Creativity*. Reading, MA: Addison-Wesley, 1995.

Kaner, Cem, Jack Falk, Hung Q. Nguyen *Testing Computer Software*, 2d ed. New York, NY: John Wiley & Sons, 1999.

Larman, Craig. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2d ed. Englewood Cliffs, NJ: Prentice Hall, 2001.

McConnell, Steve. *Rapid Development*. Redmond, WA: Microsoft Press, 1996.

Wiegers, Karl. *Software Requirements*, 2d ed. Redmond, WA: Microsoft Press, 2003.

http://cc2e.com/3514

*Manager's Handbook for Software Development*, NASA Goddard Space Flight Center. Получить информацию можно по адресу *sel.gsfc.nasa.gov/website/documents/online-doc.htm*.

## Курс профессиональной подготовки

Изучив следующую литературу, разработчик ПО добьется полного профессионального соответствия в Construx («уровень лидера»). К каждому разработчику предъявляются отдельные дополнительные требования, здесь перечислены лишь общие для всех требования.

Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*, 2d ed. Boston, MA: Addison-Wesley, 2003.

Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999.

Gamma, Erich, et al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley, 1988.

Maguire, Steve. *Writing Solid Code*. Redmond, WA: Microsoft Press, 1993.

Meyer, Bertrand. *Object-Oriented Software Construction*, 2d ed. New York, NY: Prentice Hall PTR, 1997.

http://cc2e.com/3521

*Software Measurement Guidebook*, NASA Goddard Space Flight Center. Информация доступна по адресу: *sel.gsfc.nasa.gov/website/documents/online-doc.htm*.

Наш сайт *www.construx.com/professionaldev/* подробно знакомит с программой повышения профессионального уровня и содержит обновленный список литературы.

http://cc2e.com/3528

## 35.5. Профессиональные ассоциации

Общение с другими программистами, работающими в той же области, что и вы, — один из лучших способов узнать о программировании больше. Региональные группы пользователей конкретных аппаратных средств и языков программирования — один из видов сообществ. К другим относятся национальные и международные организации. Компьютерная ассоциация IEEE — наиболее профессионально-ориентированная организация, выпускающая журналы «IEEE Computer» и «IEEE Software». Информацию о членстве в IEEE смотрите на сайте *www.computer.org*.

http://cc2e.com/3535

ACM была первой профессиональной организацией. Она издает журнал «Communications of the ACM», многие другие специализированные журналы и в сравнении с IEEE больше внимания уделяет теории. Информацию о членстве в ACM см. на сайте *www.acm.org*.

http://cc2e.com/3542

# Библиография


«A C Coding Standard.» 1991. *Unix Review* 9, no. 9 (September): 42–43.

Abdel-Hamid, Tarek K. 1989. «The Dynamics of Software Project Staffing: A System Dynamics Based Simulation Approach.» *IEEE Transactions on Software Engineering* SE-15, no. 2 (February): 109–19.

Abran, Alain, et al. 2001. Swebok: Guide to the Software Engineering Body of Knowledge: Trial Version 1.00-May 2001. Los Alamitos, CA: IEEE Computer Society Press.

Abrash, Michael. 1992. «Flooring It: The Optimization Challenge.» *PC Techniques* 2, no. 6 (February/March): 82–88.

Ackerman, A. Frank, Lynne S. Buchwald, and Frank H. Lewski. 1989. «Software Inspections: An Effective Verification Process.» *IEEE Software*, May/June 1989, 31–36.

Adams, James L. 2001. *Conceptual Blockbusting: A Guide to Better Ideas*, 4th ed. Cambridge, MA: Perseus Publishing.

Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberg. 1977. *The AWK Programming Language*. Reading, MA: Addison-Wesley.

Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman. 1983. *Data Structures and Algorithms*. Reading, MA: Addison-Wesley.

Albrecht, Allan J. 1979. «Measuring Application Development Productivity.» Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium, October 1979: 83–92.

Ambler, Scott. 2003. *Agile Database Techniques*. New York, NY: John Wiley & Sons.

Anand, N. 1988. «Clarify Function!» *ACM Sigplan Notices* 23, no. 6 (June): 69–79.

Aristotle. *The Ethics of Aristotle: The Nicomachean Ethics*. Trans. by J.A.K. Thomson. Rev. by Hugh Tredennick. Harmondsworth, Middlesex, England: Penguin, 1976.

Armenise, Pasquale. 1989. «A Structured Approach to Program Optimization.» *IEEE Transactions on Software Engineering* SE-15, no. 2 (February): 101–8.

Arnold, Ken, James Gosling, and David Holmes. 2000. *The Java Programming Language*, 3d ed. Boston, MA: Addison-Wesley.

Arthur, Lowell J. 1988. *Software Evolution: The Software Maintenance Challenge*. New York, NY: John Wiley & Sons.

Augustine, N. R. 1979. «Augustine's Laws and Major System Development Programs.» *Defense Systems Management Review*: 50–76.

Babich, W. 1986. *Software Configuration Management*. Reading, MA: Addison-Wesley.

Bachman, Charles W. 1973. «The Programmer as Navigator.» Turing Award Lecture. *Communications of the ACM* 16, no. 11 (November): 653.

Baecker, Ronald M., and Aaron Marcus. 1990. *Human Factors and Typography for More Readable Programs*. Reading, MA: Addison-Wesley.

Bairdain, E. F. 1964. «Research Studies of Programmers and Programming.» Unpublished studies reported in Boehm 1981.

Baker, F. Terry, and Harlan D. Mills. 1973. «Chief Programmer Teams.» *Datamation* 19, no. 12 (December): 58–61.

Barbour, Ian G. 1966. *Issues in Science and Religion*. New York, NY: Harper & Row.

Barbour, Ian G. 1974. Myths, Models, and Paradigms: A Comparative Study in Science and Religion. New York, NY: Harper & Row.

Barwell, Fred, et al. 2002. *Professional VB.NET*, 2d ed. Birmingham, UK: Wrox.

Basili, V. R., and B. T. Perricone. 1984. «Software Errors and Complexity: An Empirical Investigation.» *Communications of the ACM* 27, no. 1 (January): 42–52.

Basili, Victor R., and Albert J. Turner. 1975. «Iterative Enhancement: A Practical Technique for Software Development.» *IEEE Transactions on Software Engineering* SE-1, no. 4 (December): 390–96.

Basili, Victor R., and David M. Weiss. 1984. «A Methodology for Collecting Valid Software Engineering Data.» *IEEE Transactions on Software Engineering* SE-10, no. 6 (November): 728–38.

Basili, Victor R., and Richard W. Selby. 1987. «Comparing the Effectiveness of Software Testing Strategies.» *IEEE Transactions on Software Engineering* SE-13, no. 12 (December): 1278–96.

Basili, Victor R., et al. 2002. «Lessons learned from 25 years of process improvement: The Rise and Fall of the NASA Software Engineering Laboratory,» *Proceedings of the 24th International Conference on Software Engineering*, Orlando, FL.

Basili, Victor R., Richard W. Selby, and David H. Hutchens. 1986. «Experimentation in Software Engineering.» *IEEE Transactions on Software Engineering* SE-12, no. 7 (July): 733–43.

Basili, Victor, L. Briand, and W.L. Melo. 1996. «A Validation of Object-Oriented Design Metrics as Quality Indicators,» *IEEE Transactions on Software Engineering*, October 1996, 751–761.

Bass, Len, Paul Clements, and Rick Kazman. 2003. *Software Architecture in Practice*, 2d ed. Boston, MA: Addison-Wesley.

Bastani, Farokh, and Sitharama Iyengar. 1987. «The Effect of Data Structures on the Logical Complexity of Programs.» *Communications of the ACM* 30, no. 3 (March): 250–59.

Bays, Michael. 1999. *Software Release Methodology*. Englewood Cliffs, NJ: Prentice Hall.

Beck, Kent. 2000. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley.

Beck, Kent. 2003. *Test-Driven Development: By Example*. Boston, MA: Addison-Wesley.

Beck, Kent. 1991. «Think Like An Object.» *Unix Review* 9, no. 10 (October): 39–43.

Beck, Leland L., and Thomas E. Perkins. 1983. «A Survey of Software Engineering Practice: Tools, Methods, and Results.» *IEEE Transactions on Software Engineering* SE-9, no. 5 (September): 541–61.

Beizer, Boris. 1990. *Software Testing Techniques*, 2d ed. New York, NY: Van Nostrand Reinhold.

Bentley, Jon, and Donald Knuth. 1986. «Literate Programming.» *Communications of the ACM 29*, no. 5 (May): 364–69.

Bentley, Jon, Donald Knuth, and Doug McIlroy. 1986. «A Literate Program.» *Communications of the ACM* 29, no. 5 (May): 471–83.

Bentley, Jon. 1982. *Writing Efficient Programs*. Englewood Cliffs, NJ: Prentice Hall.

Bentley, Jon. 1988. *More Programming Pearls: Confessions of a Coder*. Reading, MA: Addison-Wesley.

Bentley, Jon. 1991. «Software Exploratorium: Writing Efficient C Programs.» *Unix Review* 9, no. 8 (August): 62–73.

Bentley, Jon. 2000. *Programming Pearls*, 2d ed. Reading, MA: Addison-Wesley.

Berczuk, Stephen P. and Brad Appleton. 2003. Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Boston, MA: Addison-Wesley.

Berry, R. E., and B. A. E. Meekings. 1985. «A Style Analysis of C Programs.» *Communications of the ACM* 28, no. 1 (January): 80–88.

Bersoff, Edward H. 1984. «Elements of Software Configuration Management.» *IEEE Transactions on Software Engineering* SE-10, no. 1 (January): 79–87.

Bersoff, Edward H., and Alan M. Davis. 1991. «Impacts of Life Cycle Models on Software Configuration Management.» *Communications of the ACM* 34, no. 8 (August): 104–18.

Bersoff, Edward H., et al. 1980. *Software Configuration Management*. Englewood Cliffs, NJ: Prentice Hall.

Birrell, N. D., and M. A. Ould. 1985. *A Practical Handbook for Software Development*. Cambridge, England: Cambridge University Press.

Bloch, Joshua. 2001. *Effective Java Programming Language Guide*. Boston, MA: Addison-Wesley.

BLS 2002. Occupational Outlook Handbook 2002-03 Edition, Bureau of Labor Statistics.

BLS 2004. Occupational Outlook Handbook 2004-05 Edition, Bureau of Labor Statistics.

Blum, Bruce I. 1989. «A Software Environment: Some Surprising Empirical Results.» *Proceedings of the Fourteenth Annual Software Engineering Workshop, November 29, 1989*. Greenbelt, MD: Goddard Space Flight Center. Document SEL-89-007.

Boddie, John. 1987. *Crunch Mode*. New York, NY: Yourdon Press.

Boehm, Barry and Richard Turner. 2004. *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA: Addison-Wesley.

Boehm, Barry W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall.

Boehm, Barry W. 1984. «Software Engineering Economics.» *IEEE Transactions on Software Engineering* SE-10, no. 1 (January): 4–21.

Boehm, Barry W. 1987a. «Improving Software Productivity.» *IEEE Computer*, September, 43–57.

Boehm, Barry W. 1987b. «Industrial Software Metrics Top 10 List.» *IEEE Software* 4, no. 9 (September): 84–85.

Boehm, Barry W. 1988. «A Spiral Model of Software Development and Enhancement.» *Computer*, May, 61–72.

Boehm, Barry W., and Philip N. Papaccio. 1988. «Understanding and Controlling Software Costs.» *IEEE Transactions on Software Engineering* SE-14, no. 10 (October): 1462–77.

Boehm, Barry W., ed. 1989. *Tutorial: Software Risk Management*. Washington, DC: IEEE Computer Society Press.

Boehm, Barry W., et al. 1978. *Characteristics of Software Quality*. New York, NY: North-Holland.

Boehm, Barry W., et al. 1984. «A Software Development Environment for Improving Productivity.» *Computer*, June, 30–44.

Boehm, Barry W., T. E. Gray, and T. Seewaldt. 1984. «Prototyping Versus Specifying: A Multiproject Experiment.» *IEEE Transactions on Software Engineering* SE-10, no. 3 (May): 290–303. Also in Jones 1986b.

Boehm, Barry, et al. 2000a. *Software Cost Estimation with Cocomo II*. Boston, MA: Addison-Wesley.

Boehm, Barry. 2000b. «Unifying Software Engineering and Systems Engineering,» *IEEE Computer*, March 2000, 114–116.

Boehm-Davis, Deborah, Sylvia Sheppard, and John Bailey. 1987. «Program Design Languages: How Much Detail Should They Include?» *International Journal of Man-Machine Studies* 27, no. 4: 337–47.

Böhm, C., and G. Jacopini. 1966. «Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules.» *Communications of the ACM* 9, no. 5 (May): 366–71.

Booch, Grady. 1987. *Software Engineering with Ada*, 2d ed. Menlo Park, CA: Benjamin/Cummings.

Booch, Grady. 1994. *Object Oriented Analysis and Design with Applications*, 2d ed. Boston, MA: Addison-Wesley.

Booth, Rick. 1997. Inner Loops : A Sourcebook for Fast 32-bit Software Development. Boston, MA: Addison-Wesley.

Boundy, David. 1991. «A Taxonomy of Programmers.» *ACM SIGSOFT Software Engineering Notes* 16, no. 4 (October): 23–30.

Brand, Stewart. 1995. How Buildings Learn: What Happens After They're Built. Penguin USA.

Branstad, Martha A., John C. Cherniavsky, and W. Richards Adrion. 1980. «Validation, Verification, and Testing for the Individual Programmer.» *Computer*, December, 24–30.

Brockmann, R. John. 1990. Writing Better Computer User Documentation: From Paper to Hypertext: Version 2.0. New York, NY: John Wiley & Sons.

Brooks, Frederick P., Jr. 1987. «No Silver Bullets—Essence and Accidents of Software Engineering.» *Computer*, April, 10–19.

Brooks, Frederick P., Jr. 1995. The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2d ed.). Reading, MA: Addison-Wesley.

Brooks, Ruven. 1977. «Towards a Theory of the Cognitive Processes in Computer Programming.» *International Journal of Man-Machine Studies* 9: 737–51.

Brooks, W. Douglas. 1981. «Software Technology Payoff—Some Statistical Evidence.» *The Journal of Systems and Software* 2: 3–9.

Brown, A. R., and W. A. Sampson. 1973. *Program Debugging*. New York, NY: American Elsevier.

Buschman, Frank, et al. 1996. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. New York, NY: John Wiley & Sons.

Bush, Marilyn, and John Kelly. 1989. «The Jet Propulsion Laboratory's Experience with Formal Inspections.» *Proceedings of the Fourteenth Annual Software Engineering Workshop, November 29, 1989*. Greenbelt, MD: Goddard Space Flight Center. Document SEL-89-007.

Caine, S. H., and E. K. Gordon. 1975. «PDL—A Tool for Software Design.» *AFIPS Proceedings of the 1975 National Computer Conference 44*. Montvale, NJ: AFIPS Press, 271–76.

Card, David N. 1987. «A Software Technology Evaluation Program.» *Information and Software Technology* 29, no. 6 (July/August): 291–300.

Card, David N., Frank E. McGarry, and Gerald T. Page. 1987. «Evaluating Software Engineering Technologies.» *IEEE Transactions on Software Engineering* SE-13, no. 7 (July): 845–51.

Card, David N., Victor E. Church, and William W. Agresti. 1986. «An Empirical Study of Software Design Practices.» *IEEE Transactions on Software Engineering* SE-12, no. 2 (February): 264–71.

Card, David N., with Robert L. Glass. 1990. *Measuring Software Design Quality*. Englewood Cliffs, NJ: Prentice Hall.

Card, David, Gerald Page, and Frank McGarry. 1985. «Criteria for Software Modularization.» *Proceedings of the 8th International Conference on Software Engineering*. Washington, DC: IEEE Computer Society Press, 372–77.

Carnegie, Dale. 1981. *How to Win Friends and Influence People*, Revised Edition. New York, NY: Pocket Books.

Chase, William G., and Herbert A. Simon. 1973. «Perception in Chess.» *Cognitive Psychology* 4: 55–81.

Clark, R. Lawrence. 1973. «A Linguistic Contribution of GOTO-less Programming,» *Datamation*, December 1973.

Clements, Paul, ed. 2003. *Documenting Software Architectures: Views and Beyond*. Boston, MA: Addison-Wesley.

Clements, Paul, Rick Kazman, and Mark Klein. 2002. *Evaluating Software Architectures: Methods and Case Studies*. Boston, MA: Addison-Wesley.

Coad, Peter, and Edward Yourdon. 1991. *Object-Oriented Design*. Englewood Cliffs, NJ: Yourdon Press.

Cobb, Richard H., and Harlan D. Mills. 1990. «Engineering Software Under Statistical Quality Control.» *IEEE Software* 7, no. 6 (November): 45–54.

Cockburn, Alistair. 2000. *Writing Effective Use Cases*. Boston, MA: Addison-Wesley.

Cockburn, Alistair. 2002. *Agile Software Development*. Boston, MA: Addison-Wesley.

Collofello, Jim, and Scott Woodfield. 1989. «Evaluating the Effectiveness of Reliability Assurance Techniques.» *Journal of Systems and Software* 9, no. 3 (March).

Comer, Douglas. 1981. «Principles of Program Design Induced from Experience with Small Public Programs.» *IEEE Transactions on Software Engineering* SE-7, no. 2 (March): 169–74.

Constantine, Larry L. 1990a. «Comments on 'On Criteria for Module Interfaces.'» *IEEE Transactions on Software Engineering* SE-16, no. 12 (December): 1440.

Constantine, Larry L. 1990b. «Objects, Functions, and Program Extensibility.» *Computer Language*, January, 34–56.

Conte, S. D., H. E. Dunsmore, and V. Y. Shen. 1986. *Software Engineering Metrics and Models*. Menlo Park, CA: Benjamin/ Cummings.

Cooper, Doug, and Michael Clancy. 1982. *Oh! Pascal!* 2d ed. New York, NY: Norton.

Cooper, Kenneth G. and Thomas W. Mullen. 1993. «Swords and Plowshares: The Rework Cycles of Defense and Commercial Software Development Projects,» *American Programmer*, May 1993, 41–51.

Corbaty, Fernando J. 1991. «On Building Systems That Will Fail.» 1991 Turing Award Lecture. *Communications of the ACM* 34, no. 9 (September): 72–81.

Cornell, Gary and Jonathan Morrison. 2002. *Programming VB .NET: A Guide for Experienced Programmers*, Berkeley, CA: Apress.

Corwin, Al. 1991. Private communication.

CSTB 1990. «Scaling Up: A Research Agenda for Software Engineering.» Excerpts from a report by the Computer Science and Technology Board. *Communications of the ACM* 33, no. 3 (March): 281–93.

Curtis, Bill, ed. 1985. *Tutorial: Human Factors in Software Development*. Los Angeles, CA: IEEE Computer Society Press.

Curtis, Bill, et al. 1986. «Software Psychology: The Need for an Interdisciplinary Program.» *Proceedings of the IEEE* 74, no. 8: 1092–1106.

Curtis, Bill, et al. 1989. «Experimentation of Software Documentation Formats.» *Journal of Systems and Software* 9, no. 2 (February): 167–207.

Curtis, Bill, H. Krasner, and N. Iscoe. 1988. «A Field Study of the Software Design Process for Large Systems.» *Communications of the ACM* 31, no. 11 (November): 1268–87.

Curtis, Bill. 1981. «Substantiating Programmer Variability.» *Proceedings of the IEEE* 69, no. 7: 846.

Cusumano, Michael and Richard W. Selby. 1995. *Microsoft Secrets*. New York, NY: The Free Press.

Cusumano, Michael, et al. 2003. «Software Development Worldwide: The State of the Practice,» *IEEE Software*, November/December 2003, 28–34.

Dahl, O. J., E. W. Dijkstra, and C. A. R. Hoare. 1972. *Structured Programming*. New York, NY: Academic Press.

Date, Chris. 1977. *An Introduction to Database Systems*. Reading, MA: Addison-Wesley.

Davidson, Jack W., and Anne M. Holler. 1992. «Subprogram Inlining: A Study of Its Effects on Program Execution Time.» *IEEE Transactions on Software Engineering* SE-18, no. 2 (February): 89–102.

Davis, P. J. 1972. «Fidelity in Mathematical Discourse: Is One and One Really Two?» *American Mathematical Monthly*, March, 252–63.

DeGrace, Peter, and Leslie Stahl. 1990. Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms. Englewood Cliffs, NJ: Yourdon Press.

DeMarco, Tom and Timothy Lister. 1999. *Peopleware: Productive Projects and Teams*, 2d ed. New York, NY: Dorset House.

DeMarco, Tom, and Timothy Lister. 1985. «Programmer Performance and the Effects of the Workplace.» *Proceedings of the 8th International Conference on Software Engineering*. Washington, DC: IEEE Computer Society Press, 268–72.

DeMarco, Tom. 1979. Structured Analysis and Systems Specification: Tools and Techniques. Englewood Cliffs, NJ: Prentice Hall.

DeMarco, Tom. 1982. *Controlling Software Projects*. New York, NY: Yourdon Press.

DeMillo, Richard A., Richard J. Lipton, and Alan J. Perlis. 1979. «Social Processes and Proofs of Theorems and Programs.» *Communications of the ACM* 22, no. 5 (May): 271–80.

Dijkstra, Edsger. 1965. «Programming Considered as a Human Activity.» *Proceedings of the 1965 IFIP Congress*. Amsterdam: North-Holland, 213–17. Reprinted in Yourdon 1982.

Dijkstra, Edsger. 1968. «Go To Statement Considered Harmful.» *Communications of the ACM* 11, no. 3 (March): 147–48.

Dijkstra, Edsger. 1969. «Structured Programming.» Reprinted in Yourdon 1979.

Dijkstra, Edsger. 1972. «The Humble Programmer.» *Communications of the ACM* 15, no. 10 (October): 859–66.

Dijkstra, Edsger. 1985. «Fruits of Misunderstanding.» *Datamation*, February 15, 86– 87.

Dijkstra, Edsger. 1989. «On the Cruelty of Really Teaching Computer Science.» *Communications of the ACM* 32, no. 12 (December): 1397–1414.

Dunn, Robert H. 1984. *Software Defect Removal*. New York, NY: McGraw-Hill.

Ellis, Margaret A., and Bjarne Stroustrup. 1990. *The Annotated C++ Reference Manual*. Boston, MA: Addison-Wesley.

Elmasri, Ramez, and Shamkant B. Navathe. 1989. *Fundamentals of Database Systems*. Redwood City, CA: Benjamin/Cummings.

Elshoff, James L. 1976. «An Analysis of Some Commercial PL/I Programs.» *IEEE Transactions on Software Engineering* SE-2, no. 2 (June): 113–20.

Elshoff, James L. 1977. «The Influence of Structured Programming on PL/I Program Profiles.» *IEEE Transactions on Software Engineering* SE-3, no. 5 (September): 364–68.

Elshoff, James L., and Michael Marcotty. 1982. «Improving Computer Program Readability to Aid Modification.» *Communications of the ACM* 25, no. 8 (August): 512–21.

Endres, Albert. 1975. «An Analysis of Errors and Their Causes in System Programs.» *IEEE Transactions on Software Engineering* SE-1, no. 2 (June): 140–49.

Evangelist, Michael. 1984. «Program Complexity and Programming Style.» *Proceedings of the First International Conference on Data Engineering*. New York, NY: IEEE Computer Society Press, 534–41.

Fagan, Michael E. 1976. «Design and Code Inspections to Reduce Errors in Program Development.» *IBM Systems Journal* 15, no. 3: 182–211.

Fagan, Michael E. 1986. «Advances in Software Inspections.» *IEEE Transactions on Software Engineering* SE-12, no. 7 (July): 744–51.

Federal Software Management Support Center. 1986. *Programmers Work-bench Handbook*. Falls Church, VA: Office of Software Development and Information Technology.

Feiman, J., and M. Driver. 2002. «Leading Programming Languages for IT Portfolio Planning,» Gartner Research report SPA-17-6636, September 27, 2002.

Fetzer, James H. 1988. «Program Verification: The Very Idea.» *Communications of the ACM* 31, no. 9 (September): 1048–63.

FIPS PUB 38, *Guidelines for Documentation of Computer Programs and Automated Data Systems*. 1976. U.S. Department of Commerce. National Bureau of Standards. Washington, DC: U.S. Government Printing Office, Feb. 15.

Fishman, Charles. 1996. «They Write the Right Stuff,» *Fast Company*, December 1996.

Fjelstad, R. K., and W. T. Hamlen. 1979. «Applications Program Maintenance Study: Report to our Respondents.» *Proceedings Guide 48*, Philadelphia. Reprinted in *Tutorial on Software Maintenance*, G. Parikh and N. Zvegintzov, eds. Los Alamitos, CA: CS Press, 1983: 13–27.

Floyd, Robert. 1979. «The Paradigms of Programming.» *Communications of the ACM* 22, no. 8 (August): 455–60.

Fowler, Martin. 1999. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley.

Fowler, Martin. 2002. *Patterns of Enterprise Application Architecture*. Boston, MA: Addison-Wesley.

Fowler, Martin. 2003. UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3d ed. Boston, MA: Addison-Wesley.

Fowler, Martin. 2004. *UML Distilled,* 3d ed. Boston, MA: Addison-Wesley.

Fowler, Priscilla J. 1986. «In-Process Inspections of Work Products at AT&T.» *AT&T Technical Journal*, March/April, 102–12.

Foxall, James. 2003. *Practical Standards for Microsoft Visual Basic .NET*. Redmond, WA: Microsoft Press.

Freedman, Daniel P., and Gerald M. Weinberg. 1990. *Handbook of Walkthroughs, Inspections and Technical Reviews*, 3d ed. New York, NY: Dorset House.

Freeman, Peter, and Anthony I. Wasserman, eds. 1983. *Tutorial on Software Design Techniques*, 4th ed. Silver Spring, MD: IEEE Computer Society Press.

Gamma, Erich, et al. 1995. *Design Patterns*. Reading, MA: Addison-Wesley.

Gerber, Richard. 2002. Software Optimization Cookbook: High-Performance Recipes for the Intel Architecture. Intel Press.

Gibson, Elizabeth. 1990. «Objects—Born and Bred.» *BYTE*, October, 245–54.

Gilb, Tom, and Dorothy Graham. 1993. *Software Inspection*. Wokingham, England: Addison-Wesley.

Gilb, Tom. 1977. *Software Metrics*. Cambridge, MA: Winthrop.

Gilb, Tom. 1988. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley.

Gilb, Tom. 2004. *Competitive Engineering*. Boston, MA: Addison-Wesley. Downloadable from *www.result-planning.com*.

Ginac, Frank P. 1998. *Customer Oriented Software Quality Assurance*. Englewood Cliffs, NJ: Prentice Hall.

Glass, Robert L. 1982. *Modern Programming Practices: A Report from Industry*. Englewood Cliffs, NJ: Prentice Hall.

Glass, Robert L. 1988. *Software Communication Skills*. Englewood Cliffs, NJ: Prentice Hall.

Glass, Robert L. 1991. Software Conflict: Essays on the Art and Science of Software Engineering. Englewood Cliffs, NJ: Yourdon Press.

Glass, Robert L. 1995. *Software Creativity*. Reading, MA: Addison-Wesley.

Glass, Robert L. 1999. «Inspections—Some Surprising Findings,» *Communications of the ACM*, April 1999, 17–19.

Glass, Robert L. 1999. «The realities of software technology payoffs,» *Communications of the ACM*, February 1999, 74–79.

Glass, Robert L. 2003. *Facts and Fallacies of Software Engineering*. Boston, MA: Addison-Wesley.

Glass, Robert L., and Ronald A. Noiseux. 1981. *Software Maintenance Guidebook*. Englewood Cliffs, NJ: Prentice Hall.

Gordon, Ronald D. 1979. «Measuring Improvements in Program Clarity.» *IEEE Transactions on Software Engineering* SE-5, no. 2 (March): 79–90.

Gordon, Scott V., and James M. Bieman. 1991. «Rapid Prototyping and Software Quality: Lessons from Industry.» *Ninth Annual Pacific Northwest Software Quality Conference, October 7–8*. Oregon Convention Center, Portland, OR.

Gorla, N., A. C. Benander, and B. A. Benander. 1990. «Debugging Effort Estimation Using Software Metrics.» *IEEE Transactions on Software Engineering* SE-16, no. 2 (February): 223–31.

Gould, John D. 1975. «Some Psychological Evidence on How People Debug Computer Programs.» *International Journal of Man-Machine Studies* 7: 151–82.

Grady, Robert B. 1987. «Measuring and Managing Software Maintenance.» *IEEE Software* 4, no. 9 (September): 34–45.

Grady, Robert B. 1993. «Practical Rules of Thumb for Software Managers.» *The Software Practitioner* 3, no. 1 (January/February): 4–6.

Grady, Robert B. 1999. «An Economic Release Decision Model: Insights into Software Project Management.» In *Proceedings of the Applications of Software Measurement Conference*, 227–239. Orange Park, FL: Software Quality Engineering.

Grady, Robert B., and Tom Van Slack. 1994. «Key Lessons in Achieving Widespread Inspection Use,» *IEEE Software*, July 1994.

Grady, Robert B. 1992. Practical Software Metrics For Project Management And Process Improvement. Englewood Cliffs, NJ: Prentice Hall.

Grady, Robert B., and Deborah L. Caswell. 1987. *Software Metrics: Establishing a Company-Wide Program*. Englewood Cliffs, NJ: Prentice Hall.

Green, Paul. 1987. «Human Factors in Computer Systems, Some Useful Readings.» *Sigchi Bulletin* 19, no. 2: 15–20.

Gremillion, Lee L. 1984. «Determinants of Program Repair Maintenance Requirements.» *Communications of the ACM* 27, no. 8 (August): 826–32.

Gries, David. 1981. *The Science of Programming*. New York, NY: Springer-Verlag.

Grove, Andrew S. 1983. *High Output Management*. New York, NY: Random House.

Haley, Thomas J. 1996. «Software Process Improvement at Raytheon.» *IEEE Software*, November 1996.

Hansen, John C., and Roger Yim. 1987. «Indentation Styles in C.» *SIGSMALL/PC Notes* 13, no. 3 (August): 20–23.

Hanson, Dines. 1984. *Up and Running*. New York, NY: Yourdon Press.

Harrison, Warren, and Curtis Cook. 1986. «Are Deeply Nested Conditionals Less Readable?» *Journal of Systems and Software* 6, no. 4 (November): 335–42.

Hasan, Jeffrey and Kenneth Tu. 2003. Performance Tuning and Optimizing ASP.NET Applications. Apress.

Hass, Anne Mette Jonassen. 2003. *Configuration Management Principles and Practices*, Boston, MA: Addison-Wesley.

Hatley, Derek J., and Imtiaz A. Pirbhai. 1988. *Strategies for Real-Time System Specification*. New York, NY: Dorset House.

Hecht, Alan. 1990. «Cute Object-oriented Acronyms Considered FOOlish.» *Software Engineering Notes*, January, 48.

Heckel, Paul. 1994. *The Elements of Friendly Software Design*. Alameda, CA: Sybex.

Hecker, Daniel E. 2001. «Occupational Employment Projections to 2010.» *Monthly Labor Review*, November 2001.

Hecker, Daniel E. 2004. «Occupational Employment Projections to 2012.» *Monthly Labor Review*, February 2004, Vol. 127, No. 2, pp. 80-105.

Henry, Sallie, and Dennis Kafura. 1984. «The Evaluation of Software Systems' Structure Using Quantitative Software Metrics.» *Software—Practice and Experience* 14, no. 6 (June): 561–73.

Hetzel, Bill. 1988. *The Complete Guide to Software Testing*, 2d ed. Wellesley, MA: QED Information Systems.

Highsmith, James A., III. 2000. Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. New York, NY: Dorset House.

Highsmith, Jim. 2002. *Agile Software Development Ecosystems*. Boston, MA: Addison-Wesley.

Hildebrand, J. D. 1989. «An Engineer's Approach.» *Computer Language*, October, 5–7.

Hoare, Charles Anthony Richard, 1981. «The Emperor's Old Clothes.» *Communications of the ACM*, February 1981, 75–83.

Hollocker, Charles P. 1990. *Software Reviews and Audits Handbook*. New York, NY: John Wiley & Sons.

Houghton, Raymond C. 1990. «An Office Library for Software Engineering Professionals.» *Software Engineering: Tools, Techniques, Practice*, May/June, 35–38.

Howard, Michael, and David LeBlanc. 2003. *Writing Secure Code,* 2d ed. Redmond, WA: Microsoft Press.

Hughes, Charles E., Charles P. Pfleeger, and Lawrence L. Rose. 1978. *Advanced Programming Techniques: A Second Course in Programming Using Fortran*. New York, NY: John Wiley & Sons.

Humphrey, Watts S. 1989. *Managing the Software Process*. Reading, MA: Addison-Wesley.

Humphrey, Watts S. 1995. *A Discipline for Software Engineering*. Reading, MA: Addison-Wesley.

Humphrey, Watts S., Terry R. Snyder, and Ronald R. Willis. 1991. «Software Process Improvement at Hughes Aircraft.» *IEEE Software* 8, no. 4 (July): 11–23.

Humphrey, Watts. 1997. *Introduction to the Personal Software Process*. Reading, MA: Addison-Wesley.

Humphrey, Watts. 2002. *Winning with Software: An Executive Strategy*. Boston, MA: Addison-Wesley.

Hunt, Andrew, and David Thomas. 2000. *The Pragmatic Programmer*. Boston, MA: Addison-Wesley.

Ichbiah, Jean D., et al. 1986. *Rationale for Design of the Ada Programming Language*. Minneapolis, MN: Honeywell Systems and Research Center.

*IEEE Software* 7, no. 3 (May 1990).

IEEE Std 1008-1987 (R1993), Standard for Software Unit Testing

IEEE Std 1016-1998, Recommended Practice for Software Design Descriptions

IEEE Std 1028-1997, Standard for Software Reviews

IEEE Std 1045-1992, Standard for Software Productivity Metrics

IEEE Std 1058-1998, Standard for Software Project Management Plans

IEEE Std 1061-1998, Standard for a Software Quality Metrics Methodology

IEEE Std 1062-1998, Recommended Practice for Software Acquisition

IEEE Std 1063-2001, Standard for Software User Documentation

IEEE Std 1074-1997, Standard for Developing Software Life Cycle Processes

IEEE Std 1219-1998, Standard for Software Maintenance

IEEE Std 1233-1998, Guide for Developing System Requirements Specifications

IEEE Std 1233-1998. IEEE Guide for Developing System Requirements Specifications

IEEE Std 1471-2000. Recommended Practice for Architectural Description of Software Intensive Systems

IEEE Std 1490-1998, Guide - Adoption of PMI Standard - A Guide to the Project Management Body of Knowledge

IEEE Std 1540-2001, Standard for Software Life Cycle Processes - Risk Management

IEEE Std 730-2002, Standard for Software Quality Assurance Plans

IEEE Std 828-1998, Standard for Software Configuration Management Plans

IEEE Std 829-1998, Standard for Software Test Documentation

IEEE Std 830-1998, Recommended Practice for Software Requirements Specifications

*IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications. Los Alamitos, CA: IEEE Computer Society Press.*

IEEE, 1991. *IEEE Software Engineering Standards Collection, Spring 1991 Edition*. New York, NY: Institute of Electrical and Electronics Engineers.

IEEE, 1992. «Rear Adm. Grace Hopper dies at 85.» *IEEE Computer*, February, 84.

Ingrassia, Frank S. 1976. «The Unit Development Folder (UDF): An Effective Management Tool for Software Development.» TRW Technical Report TRW-SS-76-11. Also reprinted in Reifer 1986, 366–79.

Ingrassia, Frank S. 1987. «The Unit Development Folder (UDF): A Ten-Year Perspective.» *Tutorial: Software Engineering Project Management*, ed. Richard H. Thayer. Los Alamitos, CA: IEEE Computer Society Press, 405–15.

Jackson, Michael A. 1975. *Principles of Program Design*. New York, NY: Academic Press.

Jacobson, Ivar, Grady Booch, and James Rumbaugh. 1999. *The Unified Software Development Process*. Reading, MA: Addison-Wesley.

Johnson, Jim. 1999. «Turning Chaos into Success,» *Software Magazine*, December 1999, 30–39.

Johnson, Mark. 1994a. «Dr. Boris Beizer on Software Testing: An Interview Part 1,» *The Software QA Quarterly*, Spring 1994, 7–13.

Johnson, Mark. 1994b. «Dr. Boris Beizer on Software Testing: An Interview Part 2,» *The Software QA Quarterly*, Summer 1994, 41–45.

Johnson, Walter L. 1987. «Some Comments on Coding Practice.» *ACM SIGSOFT Software Engineering Notes* 12, no. 2 (April): 32–35.

Jones, T. Capers. 1977. «Program Quality and Programmer Productivity.» *IBM Technical Report TR 02.764*, January, 42–78. Also in Jones 1986b.

Jones, Capers. 1986a. *Programming Productivity*. New York, NY: McGraw-Hill.

Jones, T. Capers, ed. 1986b. *Tutorial: Programming Productivity: Issues for the Eighties*, 2d ed. Los Angeles, CA: IEEE Computer Society Press.

Jones, Capers. 1996. «Software Defect-Removal Efficiency,» *IEEE Computer*, April 1996.

Jones, Capers. 1997. Applied Software Measurement: Assuring Productivity and Quality, 2d ed. New York, NY: McGraw-Hill.

Jones, Capers. 1998. *Estimating Software Costs*. New York, NY: McGraw-Hill.

Jones, Capers. 2000. Software Assessments, Benchmarks, and Best Practices. Reading, MA: Addison-Wesley.

Jones, Capers. 2003. «Variations in Software Development Practices,» *IEEE Software*, November/December 2003, 22–27.

Jonsson, Dan. 1989. «Next: The Elimination of GoTo-Patches?» *ACM Sigplan Notices* 24, no. 3 (March): 85–92.

Kaelbling, Michael. 1988. «Programming Languages Should NOT Have Comment Statements.» *ACM Sigplan Notices* 23, no. 10 (October): 59–60.

Kaner, Cem, Jack Falk, and Hung Q. Nguyen. 1999. *Testing Computer Software*, 2d ed. New York, NY: John Wiley & Sons.

Kaner, Cem, James Bach, and Bret Pettichord. 2002. *Lessons Learned in Software Testing*. New York, NY: John Wiley & Sons.

Keller, Daniel. 1990. «A Guide to Natural Naming.» *ACM Sigplan Notices* 25, no. 5 (May): 95–102.

Kelly, John C. 1987. «A Comparison of Four Design Methods for Real-Time Systems.» *Proceedings of the Ninth International Conference on Software Engineering*. 238–52.

Kelly-Bootle, Stan. 1981. *The Devil's DP Dictionary*. New York, NY: McGraw-Hill.

Kernighan, Brian W., and Rob Pike. 1999. *The Practice of Programming*. Reading, MA: Addison-Wesley.

Kernighan, Brian W., and P. J. Plauger. 1976. *Software Tools*. Reading, MA: Addison-Wesley.

Kernighan, Brian W., and P. J. Plauger. 1978. *The Elements of Programming Style*. 2d ed. New York, NY: McGraw-Hill.

Kernighan, Brian W., and P. J. Plauger. 1981. *Software Tools in Pascal*. Reading, MA: Addison-Wesley.

Kernighan, Brian W., and Dennis M. Ritchie. 1988. *The C Programming Language*, 2d ed. Englewood Cliffs, NJ: Prentice Hall.

Killelea, Patrick. 2002. *Web Performance Tuning*, 2d ed. Sebastopol, CA: O'Reilly & Associates.

King, David. 1988. Creating Effective Software: Computer Program Design Using the Jackson Methodology. New York, NY: Yourdon Press.

Knuth, Donald. 1971. «An Empirical Study of FORTRAN programs,» *Software—Practice and Experience* 1: 105–33.

Knuth, Donald. 1974. «Structured Programming with go to Statements.» In *Classics in Software Engineering*, edited by Edward Yourdon. Englewood Cliffs, NJ: Yourdon Press, 1979.

Knuth, Donald. 1986. Computers and Typesetting, Volume B, TEX: The Program. Reading, MA: Addison-Wesley.

Knuth, Donald. 1997a. *The Art of Computer Programming*, vol. 1, *Fundamental Algorithms*, 3d ed. Reading, MA: Addison-Wesley.

Knuth, Donald. 1997b. *The Art of Computer Programming*, vol. 2, *Seminumerical Algorithms*, 3d ed. Reading, MA: Addison-Wesley.

Knuth, Donald. 1998. *The Art of Computer Programming*, vol. 3, *Sorting and Searching*, 2d ed. Reading, MA: Addison-Wesley.

Knuth, Donald. 2001. *Literate Programming*. Cambridge University Press.

Korson, Timothy D., and Vijay K. Vaishnavi. 1986. «An Empirical Study of Modularity on Program Modifiability.» In Soloway and Iyengar 1986: 168–86.

Kouchakdjian, Ara, Scott Green, and Victor Basili. 1989. «Evaluation of the Cleanroom Methodology in the Software Engineering Laboratory.» *Proceedings of the Fourteenth Annual Software Engineering Workshop, November 29, 1989*. Greenbelt, MD: Goddard Space Flight Center. Document SEL-89-007.

Kovitz, Benjamin, L. 1998 Practical Software Requirements: A Manual of Content and Style, Manning Publications Company.

Kreitzberg, C. B., and B. Shneiderman. 1972. *The Elements of Fortran Style*. New York, NY: Harcourt Brace Jovanovich.

Kruchten, Philippe B. «The 4+1 View Model of Architecture.» *IEEE Software*, pages 42–50, November 1995.

Kruchten, Philippe. 2000. The Rational Unified Process: An Introduction, 2d Ed., Reading, MA: Addison-Wesley.

Kuhn, Thomas S. 1996. *The Structure of Scientific Revolutions*, 3d ed. Chicago: University of Chicago Press.

Lammers, Susan. 1986. *Programmers at Work*. Redmond, WA: Microsoft Press.

Lampson, Butler. 1984. «Hints for Computer System Design.» *IEEE Software* 1, no. 1 (January): 11–28.

Larman, Craig and Rhett Guthrie. 2000. *Java 2 Performance and Idiom Guide*. Englewood Cliffs, NJ: Prentice Hall.

Larman, Craig. 2001. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2d ed. Englewood Cliffs, NJ: Prentice Hall.

Larman, Craig. 2004. *Agile and Iterative Development: A Manager's Guide*. Boston, MA: Addison-Wesley, 2004.

Lauesen, Soren. *Software Requirements: Styles and Techniques*. Boston, MA: Addison-Wesley, 2002.

Laurel, Brenda, ed. 1990. *The Art of Human-Computer Interface Design*. Reading, MA: Addison-Wesley.

Ledgard, Henry F., with John Tauer. 1987a. *C With Excellence: Programming Proverbs*. Indianapolis: Hayden Books.

Ledgard, Henry F., with John Tauer. 1987b. *Professional Software*, vol. 2, *Programming Practice*. Indianapolis: Hayden Books.

Ledgard, Henry, and Michael Marcotty. 1986. *The Programming Language Landscape: Syntax, Semantics, and Implementation*, 2d ed. Chicago: Science Research Associates.

Ledgard, Henry. 1985. «Programmers: The Amateur vs. the Professional.» *Abacus* 2, no. 4 (Summer): 29–35.

Leffingwell, Dean. 1997. «Calculating the Return on Investment from More Effective Requirements Management,» *American Programmer*, 10(4):13–16.

Lewis, Daniel W. 1979. «A Review of Approaches to Teaching Fortran.» *IEEE Transactions on Education*, E-22, no. 1: 23–25.

Lewis, William E. 2000. *Software Testing and Continuous Quality Improvement*, 2d ed. Auerbach Publishing.

Lieberherr, Karl J. and Ian Holland. 1989. «Assuring Good Style for Object-Oriented Programs.» *IEEE Software*, September 1989, pp. 38f.

Lientz, B. P., and E. B. Swanson. 1980. *Software Maintenance Management*. Reading, MA: Addison-Wesley.

Lind, Randy K., and K. Vairavan. 1989. «An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort.» *IEEE Transactions on Software Engineering* SE-15, no. 5 (May): 649–53.

Linger, Richard C., Harlan D. Mills, and Bernard I. Witt. 1979. *Structured Programming: Theory and Practice*. Reading, MA: Addison-Wesley.

Linn, Marcia C., and Michael J. Clancy. 1992. «The Case for Case Studies of Programming Problems.» *Communications of the ACM* 35, no. 3 (March): 121–32.

Liskov, Barbara, and Stephen Zilles. 1974. «Programming with Abstract Data Types.» *ACM Sigplan Notices* 9, no. 4: 50–59.

Liskov, Barbara. «Data Abstraction and Hierarchy,» *ACM SIGPLAN Not*ices, May 1988.

Littman, David C., et al. 1986. «Mental Models and Software Maintenance.» In Soloway and Iyengar 1986: 80–98.

Longstreet, David H., ed. 1990. *Software Maintenance and Computers*. Los Alamitos, CA: IEEE Computer Society Press.

Loy, Patrick H. 1990. «A Comparison of Object-Oriented and Structured Development Methods.» *Software Engineering Notes* 15, no. 1 (January): 44–48.

Mackinnon, Tim, Steve Freeman, and Philip Craig. 2000. «Endo-Testing: Unit Testing with Mock Objects,» *eXtreme Programmin*g and Flexible Processes Software Engineering - XP2000 Conference.

Maguire, Steve. 1993. *Writing Solid Code*. Redmond, WA: Microsoft Press.

Mannino, P. 1987. «A Presentation and Comparison of Four Information System Development Methodologies.» *Software Engineering Notes* 12, no. 2 (April): 26– 29.

Manzo, John. 2002. «Odyssey and Other Code Science Success Stories.» *Crosstalk*, October 2002.

Marca, David. 1981. «Some Pascal Style Guidelines.» *ACM Sigplan Notices* 16, no. 4 (April): 70–80.

March, Steve. 1999. «Learning from Pathfinder's Bumpy Start.» *Software Testing and Quality Engineering*, September/October 1999, pp. 10f.

Marcotty, Michael. 1991. *Software Implementation*. New York, NY: Prentice Hall.

Martin, Robert C. 2003. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ: Pearson Education.

McCabe, Tom. 1976. «A Complexity Measure.» *IEEE Transactions on Software Engineering*, SE-2, no. 4 (December): 308–20.

McCarthy, Jim. 1995. *Dynamics of Software Development*. Redmond, WA: Microsoft Press.

McConnell, Steve. 1996. *Rapid Development*. Redmond, WA: Microsoft Press.

McConnell, Steve. 1997a. «The Programmer Writing,» *IEEE Software*, July/August 1997.

McConnell, Steve. 1997b. «Achieving Leaner Software,» *IEEE Software*, November/December 1997.

McConnell, Steve. 1998a. *Software Project Survival Guide*. Redmond, WA: Microsoft Press.

McConnell, Steve. 1998b. «Why You Should Use Routines, Routinely,» *IEEE Software*, Vol. 15, No. 4, July/August 1998.

McConnell, Steve. 1999. «Brooks Law Repealed?» *IEEE Software*, November/December 1999.

McConnell, Steve. 2004. *Professional Software Development*. Boston, MA: Addison-Wesley.

McCue, Gerald M. 1978. «IBM's Santa Teresa Laboratory—Architectural Design for Program Development.» *IBM Systems Journal* 17, no. 1: 4–25.

McGarry, Frank, and Rose Pajerski. 1990. «Towards Understanding Software—15 Years in the SEL.» *Proceedings of the Fifteenth Annual Software Engineering Workshop, November 28–29, 1990*. Greenbelt, MD: Goddard Space Flight Center. Document SEL-90-006.

McGarry, Frank, Sharon Waligora, and Tim McDermott. 1989. «Experiences in the Software Engineering Laboratory (SEL) Applying Software Measurement.» *Proceedings of the Fourteenth Annual Software Engineering Workshop, November 29, 1989*. Greenbelt, MD: Goddard Space Flight Center. Document SEL-89-007.

McGarry, John, et al. 2001. Practical Software Measurement: Objective Information for Decision Makers. Boston, MA: Addison-Wesley.

McKeithen, Katherine B., et al. 1981. «Knowledge Organization and Skill Differences in Computer Programmers.» *Cognitive Psychology* 13: 307–25.

Metzger, Philip W., and John Boddie. 1996. *Managing a Programming Project: Processes and People*, 3d ed. Englewood Cliffs, NJ: Prentice Hall, 1996.

Meyer, Bertrand. 1997. *Object-Oriented Software Construction*, 2d ed. New York, NY: Prentice Hall.

Meyers, Scott. 1996. More Effective C++: 35 New Ways to Improve Your Programs and Designs. Reading, MA: Addison-Wesley.

Meyers, Scott. 1998. Effective C++: 50 Specific Ways to Improve Your Programs and Designs, 2d ed. Reading, MA: Addison-Wesley.

Miaria, Richard J., et al. 1983. «Program Indentation and Comprehensibility.» *Communications of the ACM* 26, no. 11 (November): 861–67.

Michalewicz, Zbigniew, and David B. Fogel. 2000. *How to Solve It: Modern Heuristics*. Berlin: Springer-Verlag.

Miller, G. A. 1956. «The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information.» *The Psychological Review* 63, no. 2 (March): 81–97.

Mills, Harlan D. 1983. *Software Productivity*. Boston, MA: Little, Brown.

Mills, Harlan D. 1986. «Structured Programming: Retrospect and Prospect.» *IEEE Software*, November, 58–66.

Mills, Harlan D., and Richard C. Linger. 1986. «Data Structured Programming: Program Design Without Arrays and Pointers.» *IEEE Transactions on Software Engineering* SE-12, no. 2 (February): 192–97.

Mills, Harlan D., Michael Dyer, and Richard C. Linger. 1987. «Cleanroom Software Engineering.» *IEEE Software*, September, 19–25.

Misfeldt, Trevor, Greg Bumgardner, and Andrew Gray. 2004. *The Elements of C++ Style*. Cambridge University Press.

Mitchell, Jeffrey, Joseph Urban, and Robert McDonald. 1987. «The Effect of Abstract Data Types on Program Development.» *IEEE Computer* 20, no. 9 (September): 85–88.

Mody, R. P. 1991. «C in Education and Software Engineering.» *SIGCSE Bulletin* 23, no. 3 (September): 45–56.

Moore, Dave. 1992. Private communication.

Moore, James W. 1997. *Software Engineering Standards: A User's Road Map*. Los Alamitos, CA: IEEE Computer Society Press.

Morales, Alexandra Weber. 2003. «The Consummate Coach: Watts Humphrey, Father of Cmm and Author of Winning with Software, Explains How to Get Better at What You Do,» *SD Show Daily*, September 16, 2003.

Myers, Glenford J. 1976. *Software Reliability*. New York, NY: John Wiley & Sons.

Myers, Glenford J. 1978a. *Composite/Structural Design*. New York, NY: Van Nostrand Reinhold.

Myers, Glenford J. 1978b. «A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections.» *Communications of the ACM* 21, no. 9 (September): 760–68.

Myers, Glenford J. 1979. *The Art of Software Testing*. New York, NY: John Wiley & Sons.

Myers, Ware. 1992. «Good Software Practices Pay Off—Or Do They?» *IEEE Software*, March, 96–97.

Naisbitt, John. 1982. *Megatrends*. New York, NY: Warner Books.

NASA Software Engineering Laboratory, 1994. Software Measurement Guidebook, June 1995, NASA-GB-001-94. Available from http://sel.gsfc.nasa.gov/website/documents/online-doc/94-102.pdf.

NCES 2002. National Center for Education Statistics, *2001 Digest of Educational Statistics,* Document Number NCES 2002130, April 2002.

Nevison, John M. 1978. *The Little Book of BASIC Style*. Reading, MA: Addison-Wesley.

Newcomer, Joseph M. 2000. «Optimization: Your Worst Enemy,» May 2000, *www.flounder. com/ optimization.htm*.

Norcio, A. F. 1982. «Indentation, Documentation and Programmer Comprehension.» *Proceedings: Human Factors in Computer Systems, March 15–17, 1982, Gaithersburg, MD:* 118–20.

Norman, Donald A. 1988. *The Psychology of Everyday Things*. New York, NY: Basic Books. (Also published in paperback as *The Design of Everyday Things*. New York, NY: Doubleday, 1990.)

Oman, Paul and Shari Lawrence Pfleeger, eds. 1996. *Applying Software Metrics*. Los Alamitos, CA: IEEE Computer Society Press.

Oman, Paul W., and Curtis R. Cook. 1990a. «The Book Paradigm for Improved Maintenance.» *IEEE Software*, January, 39–45.

Oman, Paul W., and Curtis R. Cook. 1990b. «Typographic Style Is More Than Cosmetic.» *Communications of the ACM* 33, no. 5 (May): 506–20.

Ostrand, Thomas J., and Elaine J. Weyuker. 1984. «Collecting and Categorizing Software Error Data in an Industrial Environment.» *Journal of Systems and Software* 4, no. 4 (November): 289–300.

Page-Jones, Meilir. 2000. *Fundamentals of Object-Oriented Design in UML*. Boston, MA: Addison-Wesley.

Page-Jones, Meilir. 1988. *The Practical Guide to Structured Systems Design*. Englewood Cliffs, NJ: Yourdon Press.

Parikh, G., and N. Zvegintzov, eds. 1983. *Tutorial on Software Maintenance*. Los Alamitos, CA: IEEE Computer Society Press.

Parikh, Girish. 1986. *Handbook of Software Maintenance*. New York, NY: John Wiley & Sons.

Parnas, David L. 1972. «On the Criteria to Be Used in Decomposing Systems into Modules.» *Communications of the ACM* 5, no. 12 (December): 1053–58.

Parnas, David L. 1976. «On the Design and Development of Program Families.» *IEEE Transactions on Software Engineering* SE-2, 1 (March): 1–9.

Parnas, David L. 1979. «Designing Software for Ease of Extension and Contraction.» *IEEE Transactions on Software Engineering* SE-5, no. 2 (March): 128–38.

Parnas, David L. 1999. ACM Fellow Profile: David Lorge Parnas,» *ACM Software Engineering Notes*, May 1999, 10–14.

Parnas, David L., and Paul C. Clements. 1986. «A Rational Design Process: How and Why to Fake It.» *IEEE Transactions on Software Engineering* SE-12, no. 2 (February): 251–57.

Parnas, David L., Paul C. Clements, and D. M. Weiss. 1985. «The Modular Structure of Complex Systems.» *IEEE Transactions on Software Engineering* SE-11, no. 3 (March): 259–66.

Perrott, Pamela. 2004. Private communication.

Peters, L. J., and L. L. Tripp. 1976. «Is Software Design Wicked» *Datamation*, Vol. 22, No. 5 (May 1976), 127–136.

Peters, Lawrence J. 1981. *Handbook of Software Design: Methods and Techniques*. New York, NY: Yourdon Press.

Peters, Lawrence J., and Leonard L. Tripp. 1977. «Comparing Software Design Methodologies.» *Datamation*, November, 89– 94.

Peters, Tom. 1987. Thriving on Chaos: Handbook for a Management Revolution. New York, NY: Knopf.

Petroski, Henry. 1994. *Design Paradigms: Case Histories of Error and Judgment in Engineering*. Cambridge, U.K.: Cambridge University Press.

Pietrasanta, Alfred M. 1990. «Alfred M. Pietrasanta on Improving the Software Process.» *Software Engineering: Tools, Techniques, Practices* 1, no. 1 (May/ June): 29–34.

Pietrasanta, Alfred M. 1991a. «A Strategy for Software Process Improvement.» *Ninth Annual Pacific Northwest Software Quality Conference, October 7–8, 1991*. Oregon Convention Center, Portland, OR

Pietrasanta, Alfred M. 1991b. «Implementing Software Engineering in IBM.» Keynote address. *Ninth Annual Pacific Northwest Software Quality Conference, October 7– 8, 1991*. Oregon Convention Center, Portland, OR.

Pigoski, Thomas M. 1997. *Practical Software Maintenance*. New York, NY: John Wiley & Sons.

Pirsig, Robert M. 1974. Zen and the Art of Motorcycle Maintenance: An Inquiry into Values. William Morrow.

Plauger, P. J. 1988. «A Designer's Bibliography.» *Computer Language*, July, 17–22.

Plauger, P. J. 1993. *Programming on Purpose: Essays on Software Design*. New York, NY: Prentice Hall.

Plum, Thomas. 1984. *C Programming Guidelines*. Cardiff, NJ: Plum Hall.

Polya, G. 1957. *How to Solve It: A New Aspect of Mathematical Method*, 2d ed. Princeton, NJ: Princeton University Press.

Post, Ed. 1983. «Real Programmers Don't Use Pascal,» *Datamation*, July 1983, 263– 265.

Prechelt, Lutz. 2000. «An Empirical Comparison of Seven Programming Languages,» *IEEE Computer*, October 2000, 23–29.

Pressman, Roger S. 1987. *Software Engineering: A Practitioner's Approach*. New York, NY: McGraw-Hill.

Pressman, Roger S. 1988. Making Software Engineering Happen: A Guide for Instituting the Technology. Englewood Cliffs, NJ: Prentice Hall.

Putnam, Lawrence H. 2000. «Familiar Metric Management – Effort, Development Time, and Defects Interact.» Downloadable from *www.qsm.com*.

Putnam, Lawrence H., and Ware Myers. 1992. *Measures for Excellence: Reliable Software On Time, Within Budget*. Englewood Cliffs, NJ: Yourdon Press, 1992.

Putnam, Lawrence H., and Ware Myers. 1997. *Industrial Strength Software: Effective Management Using Measurement*. Washington, DC: IEEE Computer Society Press.

Putnam, Lawrence H., and Ware Myers. 2000. «What We Have Learned.» Downloadable from *www.qsm.com*, June 2000.

Raghavan, Sridhar A., and Donald R. Chand. 1989. «Diffusing Software-Engineering Methods.» *IEEE Software*, July, 81–90.

Ramsey, H. Rudy, Michael E. Atwood, and James R. Van Doren. 1983. «Flowcharts Versus Program Design Languages: An Experimental Comparison.» *Communications of the ACM* 26, no. 6 (June): 445–49.

Ratliff, Wayne. 1987. Interview in *Solution System*.

Raymond, E. S. 2000. «The Cathedral and the Bazaar,» *www.catb.org/~esr/writings/cathedral-bazaar*.

Raymond, Eric S. 2004. *The Art of Unix Programming*. Boston, MA: Addison-Wesley.

Rees, Michael J. 1982. «Automatic Assessment Aids for Pascal Programs.» *ACM Sigplan Notices* 17, no. 10 (October): 33–42.

Reifer, Donald. 2002. «How to Get the Most Out of Extreme Programming/Agile Methods,» *Proceedings, XP/Agile Universe 2002*. New York, NY: Springer; 185–196.

Reingold, Edward M., and Wilfred J. Hansen. 1983. *Data Structures*. Boston, MA: Little, Brown.

Rettig, Marc. 1991. «Testing Made Palatable.» *Communications of the ACM* 34, no. 5 (May): 25–29.

Riel, Arthur J. 1996. *Object-Oriented Design Heuristics*. Reading, MA: Addison-Wesley.

Rittel, Horst, and Melvin Webber. 1973. «Dilemmas in a General Theory of Planning.» *Policy Sciences* 4: 155–69.

Robertson, Suzanne, and James Robertson, 1999. *Mastering the Requirements Process*. Reading, MA: Addison-Wesley.

Rogers, Everett M. 1995. *Diffusion of Innovations*, 4th ed. New York, NY: The Free Press.

Rombach, H. Dieter. 1990. «Design Measurements: Some Lessons Learned.» *IEEE Software*, March, 17–25.

Rubin, Frank. 1987. «'GOTO Considered Harmful' Considered Harmful.» Letter to the editor. *Communications of the ACM* 30, no. 3 (March): 195–96. Follow-up letters in 30, no. 5 (May 1987): 351–55; 30, no. 6 (June 1987): 475–78; 30, no. 7 (July 1987): 632–34; 30, no. 8 (August 1987): 659–62; 30, no. 12 (December 1987): 997, 1085.

Sackman, H., W. J. Erikson, and E. E. Grant. 1968. «Exploratory Experimental Studies Comparing Online and Offline Programming Performance.» *Communications of the ACM* 11, no. 1 (January): 3–11.

Schneider, G. Michael, Johnny Martin, and W. T. Tsai. 1992. «An Experimental Study of Fault Detection in User Requirements Documents,» *ACM Transactions on Software Engineering and Methodology*, vol 1, no. 2, 188–204.

Schulmeyer, G. Gordon. 1990. *Zero Defect Software*. New York, NY: McGraw-Hill.

Sedgewick, Robert. 1997. *Algorithms in C, Parts 1-4*, 3d ed. Boston, MA: Addison-Wesley.

Sedgewick, Robert. 2001. *Algorithms in C, Part 5*, 3d ed. Boston, MA: Addison-Wesley.

Sedgewick, Robert. 1998. *Algorithms in C++, Parts 1-4*, 3d ed. Boston, MA: Addison-Wesley.

Sedgewick, Robert. 2002. *Algorithms in C++, Part 5*, 3d ed. Boston, MA: Addison-Wesley.

Sedgewick, Robert. 2002. *Algorithms in Java, Parts 1-4*, 3d ed. Boston, MA: Addison-Wesley.

Sedgewick, Robert. 2003. *Algorithms in Java, Part 5*, 3d ed. Boston, MA: Addison-Wesley.

SEI 1995. *The Capability Maturity Model: Guidelines for Improving the Software Process*, Software Engineering Institute, Reading, MA: Addison-Wesley, 1995.

SEI, 2003. «Process Maturity Profile: Software CMM®, CBA IPI and SPA Appraisal Results: 2002 Year End Update,» Software Engineering Institute, April 2003.

Selby, Richard W., and Victor R. Basili. 1991. «Analyzing Error-Prone System Structure.» *IEEE Transactions on Software Engineering* SE-17, no. 2 (February): 141–52.

SEN 1990. «Subsection on Telephone Systems,» *Software Engineering Notes*, April 1990, 11–14.

Shalloway, Alan, and James R. Trott. 2002. *Design Patterns Explained*. Boston, MA: Addison-Wesley.

Sheil, B. A. 1981. «The Psychological Study of Programming.» *Computing Surveys* 13, no. 1 (March): 101–20.

Shen, Vincent Y., et al. 1985. «Identifying Error-Prone Software—An Empirical Study.» *IEEE Transactions on Software Engineering* SE-11, no. 4 (April): 317–24.

Sheppard, S. B., et al. 1978. «Predicting Programmers' Ability to Modify Software.» *TR 78-388100-3*, General Electric Company, May.

Sheppard, S. B., et al. 1979. «Modern Coding Practices and Programmer Performance.» *IEEE Computer* 12, no. 12 (December): 41–49.

Shepperd, M., and D. Ince. 1989. «Metrics, Outlier Analysis and the Software Design Process.» *Information and Software Technology* 31, no. 2 (March): 91–98.

Shirazi, Jack. 2000. *Java Performance Tuning*. Sebastopol, CA: O'Reilly & Associates.

Shlaer, Sally, and Stephen J. Mellor. 1988. *Object Oriented Systems Analysis—Modeling the World in Data*. Englewood Cliffs, NJ: Prentice Hall.

Shneiderman, Ben, and Richard Mayer. 1979. «Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results.» *International Journal of Computer and Information Sciences* 8, no. 3: 219–38.

Shneiderman, Ben. 1976. «Exploratory Experiments in Programmer Behavior.» *International Journal of Computing and Information Science* 5: 123–43.

Shneiderman, Ben. 1980. Software Psychology: Human Factors in Computer and Information Systems. Cambridge, MA: Winthrop.

Shneiderman, Ben. 1987. Designing the User Interface: Strategies for Effective Human-Computer Interaction. Reading, MA: Addison-Wesley.

Shull, et al. 2002. «What We Have Learned About Fighting Defects,» *Proceedings, Metrics 2002*. IEEE; 249–258.

Simon, Herbert. 1996. *The Sciences of the Artificial*, 3d ed. Cambridge, MA: MIT Press.

Simon, Herbert. The Shape of Automation for Men and Management. Harper and Row, 1965.

Simonyi, Charles, and Martin Heller. 1991. «The Hungarian Revolution.» *BYTE*, August, 131–38.

Smith, Connie U., and Lloyd G. Williams. 2002. Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Boston, MA: Addison-Wesley.

Software Productivity Consortium. 1989. *Ada Quality and Style: Guidelines for Professional Programmers*. New York, NY: Van Nostrand Reinhold.

Soloway, Elliot, and Kate Ehrlich. 1984. «Empirical Studies of Programming Knowledge.» *IEEE Transactions on Software Engineering* SE-10, no. 5 (September): 595–609.

Soloway, Elliot, and Sitharama Iyengar, eds. 1986. *Empirical Studies of Programmers*. Norwood, NJ: Ablex.

Soloway, Elliot, Jeffrey Bonar, and Kate Ehrlich. 1983. «Cognitive Strategies and Looping Constructs: An Empirical Study.» *Communications of the ACM* 26, no. 11 (November): 853–60.

Solution Systems. 1987. World-Class Programmers' Editing Techniques: Interviews with Seven Programmers. South Weymouth, MA: Solution Systems.

Sommerville, Ian. 1989. *Software Engineering*, 3d ed. Reading, MA: Addison-Wesley.

Spier, Michael J. 1976. «Software Malpractice—A Distasteful Experience.» *Software—Practice and Experience* 6: 293–99.

Spinellis, Diomidis. 2003. *Code Reading: The Open Source Perspective*. Boston, MA: Addison-Wesley.

SPMN. 1998. *Little Book of Configuration Management*. Arlington, VA; Software Program Managers Network.

Starr, Daniel. 2003. «What Supports the Roof?» *Software Development*. July 2003, 38–41.

Stephens, Matt. 2003. «Emergent Design vs. Early Prototyping,» May 26, 2003, *www.softwarereality.com/design/early_prototyping.jsp*.

Stevens, Scott M. 1989. «Intelligent Interactive Video Simulation of a Code Inspection.» *Communications of the ACM* 32, no. 7 (July): 832–43.

Stevens, W., G. Myers, and L. Constantine. 1974. «Structured Design.» *IBM Systems Journal* 13, no. 2 (May): 115–39.

Stevens, Wayne. 1981. *Using Structured Design*. New York, NY: John Wiley & Sons.

Stroustrup, Bjarne. 1997. *The C++ Programming Language*, 3d ed. Reading, MA: Addison-Wesley.

Strunk, William, and E. B. White. 2000. *Elements of Style*, 4th ed. Pearson.

Sun Microsystems, Inc. 2000. «How to Write Doc Comments for the Javadoc Tool,» 2000. Available from *http://java.sun.com/j2se/javadoc/writingdoccomments/*.

Sutter, Herb. 2000. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. Boston, MA: Addison-Wesley.

Tackett, Buford D., III, and Buddy Van Doren. 1999. «Process Control for Error Free Software: A Software Success Story,» *IEEE Software*, May 1999.

Tenner, Edward. 1997. Why Things Bite Back: Technology and the Revenge of Unintended Consequences. Vintage Books.

Tenny, Ted. 1988. «Program Readability: Procedures versus Comments.» *IEEE Transactions on Software Engineering* SE-14, no. 9 (September): 1271–79.

Thayer, Richard H., ed. 1990. *Tutorial: Software Engineering Project Management*. Los Alamitos, CA: IEEE Computer Society Press.

Thimbleby, Harold. 1988. «Delaying Commitment.» *IEEE Software*, May, 78–86.

Thomas, Dave, and Andy Hunt. 2002. «Mock Objects,» *IEEE Software*, May/June 2002.

Thomas, Edward J., and Paul W. Oman. 1990. «A Bibliography of Programming Style.» *ACM Sigplan Notices* 25, no. 2 (February): 7–16.

Thomas, Richard A. 1984. «Using Comments to Aid Program Maintenance.» *BYTE*, May, 415–22.

Tripp, Leonard L., William F. Struck, and Bryan K. Pflug. 1991. «The Application of Multiple Team Inspections on a Safety-Critical Software Standard,» *Proceedings of the 4th Software Engineering Standards Application Workshop*, Los Alamitos, CA: IEEE Computer Society Press.

U.S. Department of Labor. 1990. «The 1990– 91 Job Outlook in Brief.» *Occupational Outlook Quarterly, Spring*. U.S. Government Printing Office. Document 1990-282-086/20007.

Valett, J., and F. E. McGarry. 1989. «A Summary of Software Measurement Experiences in the Software Engineering Laboratory.» *Journal of Systems and Software* 9, no. 2 (February): 137–48.

Van Genuchten, Michiel. 1991. «Why Is Software Late? An Empirical Study of Reasons for Delay in Software Development.» *IEEE Transactions on Software Engineering* SE-17, no. 6 (June): 582–90.

Van Tassel, Dennie. 1978. *Program Style, Design, Efficiency, Debugging, and Testing*, 2d ed. Englewood Cliffs, NJ: Prentice Hall.

Vaughn-Nichols, Steven. 2003. «Building Better Software with Better Tools,» *IEEE Computer*, September 2003, 12–14.

Vermeulen, Allan, et al. 2000. *The Elements of Java Style*. Cambridge University Press.

Vessey, Iris, Sirkka L. Jarvenpaa, and Noam Tractinsky. 1992. «Evaluation of Vendor Products: CASE Tools as Methodological Companions.» *Communications of the ACM* 35, no. 4 (April): 91–105.

Vessey, Iris. 1986. «Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols.» *IEEE Transactions on Systems, Man, and Cybernetics* SMC-16, no. 5 (September/October): 621–37.

Votta, Lawrence G., et al. 1991. «Investigating the Application of Capture-Recapture Techniques to Requirements and Design Reviews.» *Proceedings of the Sixteenth Annual Software Engineering Workshop, December 4–5, 1991*. Greenbelt, MD: Goddard Space Flight Center. Document SEL-91-006.

Walston, C. E., and C. P. Felix. 1977. «A Method of Programming Measurement and Estimation.» *IBM Systems Journal* 16, no. 1: 54–73.

Ward, Robert. 1989. *A Programmer's Introduction to Debugging C*. Lawrence, KS: R & D Publications.

Ward, William T. 1989. «Software Defect Prevention Using McCabe's Complexity Metric.» *Hewlett-Packard Journal*, April, 64–68.

Webster, Dallas E. 1988. «Mapping the Design Information Representation Terrain.» *IEEE Computer*, December, 8–23.

Weeks, Kevin. 1992. «Is Your Code Done Yet?» *Computer Language*, April, 63–72.

Weiland, Richard J. 1983. The Programmer's Craft: Program Construction, Computer Architecture, and Data Management. Reston, VA: Reston Publishing.

Weinberg, Gerald M. 1983. «Kill That Code!» *Infosystems*, August, 48–49.

Weinberg, Gerald M. 1998. The Psychology of Computer Programming: Silver Anniversary Edition. New York, NY: Dorset House.

Weinberg, Gerald M., and Edward L. Schulman. 1974. «Goals and Performance in Computer Programming.» *Human Factors* 16, no. 1 (February): 70–77.

Weinberg, Gerald. 1988. *Rethinking Systems Analysis and Design*. New York, NY: Dorset House.

Weisfeld, Matt. 2004. *The Object-Oriented Thought Process*, 2d ed. SAMS, 2004.

Weiss, David M. 1975. «Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility.» *Journal of Systems and Software* 1, no. 2 (June): 57–70.

Weiss, Eric A. 1972. «Review of *The Psychology of Computer Programming*, by Gerald M. Weinberg.» *ACM Computing Reviews* 13, no. 4 (April): 175–76.

Wheeler, David, Bill Brykczynski, and Reginald Meeson. 1996. *Software Inspection: An Industry Best Practice*. Los Alamitos, CA: IEEE Computer Society Press.

Whittaker, James A. 2000 «What Is Software Testing? And Why Is It So Hard?» *IEEE Software*, January 2000, 70–79.

Whittaker, James A. 2002. *How to Break Software: A Practical Guide to Testing*. Boston, MA: Addison-Wesley.

Whorf, Benjamin. 1956. *Language, Thought and Reality*. Cambridge, MA: MIT Press.

Wiegers, Karl. 2002. *Peer Reviews in Software: A Practical Guide*. Boston, MA: Addison-Wesley.

Wiegers, Karl. 2003. *Software Requirements*, 2d ed. Redmond, WA: Microsoft Press.

Williams, Laurie, and Robert Kessler. 2002. *Pair Programming Illuminated*. Boston, MA: Addison-Wesley.

Willis, Ron R., et al. 1998. «Hughes Aircraft's Widespread Deployment of a Continuously Improving Software Process,» Software Engineering Institute/Carnegie Mellon University, CMU/SEI-98-TR-006, May 1998.

Wilson, Steve, and Jeff Kesselman. 2000. *Java Platform Performance: Strategies and Tactics*. Boston, MA: Addison-Wesley.

Wirth, Niklaus. 1995. «A Plea for Lean Software,» *IEEE Computer*, February 1995.

Wirth, Niklaus. 1971. «Program Development by Stepwise Refinement.» *Communications of the ACM* 14, no. 4 (April): 221–27.

Wirth, Niklaus. 1986. *Algorithms and Data Structures*. Englewood Cliffs, NJ: Prentice Hall.

Woodcock, Jim, and Martin Loomes. 1988. *Software Engineering Mathematics*. Reading, MA: Addison-Wesley.

Woodfield, S. N., H. E. Dunsmore, and V. Y. Shen. 1981. «The Effect of Modularization and Comments on Program Comprehension.» *Proceedings of the Fifth International Conference on Software Engineering*, March 1981, 215–23.

Wulf, W. A. 1972. «A Case Against the GOTO.» *Proceedings of the 25th National ACM Conference*, August 1972, 791–97.

Youngs, Edward A. 1974. «Human Errors in Programming.» *International Journal of Man-Machine Studies* 6: 361–76.

Yourdon, Edward, and Larry L. Constantine. 1979. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs, NJ: Yourdon Press.

Yourdon, Edward, ed. 1979. *Classics in Software Engineering*. Englewood Cliffs, NJ: Yourdon Press.

Yourdon, Edward, ed. 1982. Writings of the Revolution: Selected Readings on Software Engineering. New York, NY: Yourdon Press.

Yourdon, Edward. 1986a. Managing the Structured Techniques: Strategies for Software Development in the 1990s, 3d ed. New York, NY: Yourdon Press.

Yourdon, Edward. 1986b. *Nations at Risk*. New York, NY: Yourdon Press.

Yourdon, Edward. 1988. «The 63 Greatest Software Books.» *American Programmer*, September.

Yourdon, Edward. 1989a. *Modern Structured Analysis*. New York, NY: Yourdon Press.

Yourdon, Edward. 1989b. *Structured Walk-Throughs*, 4th ed. New York, NY: Yourdon Press.

Yourdon, Edward. 1992. *Decline & Fall of the American Programmer*. Englewood Cliffs, NJ: Yourdon Press.

Zachary, Pascal. 1994. *Showstopper!* The Free Press.

Zahniser, Richard A. 1992. «A Massively Parallel Software Development Approach.» *American Programmer*, January, 34–41.

# Предметный указатель

## К



## Л



## М

## О



## П

## Р



## С



## Т



## У

# Об авторе

Стив Макконнелл — главный разработчик ПО в компании Construx Software, где следит за применением методик разработки. Кроме того, он возглавляет отделение Construction Knowledge Area проекта Software Engineering Body of Knowledge (SWEBOK). Стив работал над программными проектами в Microsoft, Boeing и других компаниях, расположенных около Сиэтла.

Перу Стива принадлежат книги «Rapid Development» (1996), «Software Project Survival Guide» (1998) и «Professional Software Development» (2004). Его книги дважды были удостоены премии Jolt Excellence журнала «Software Development» как лучшие книги года о разработке ПО. Стив также был ведущим разработчиком инструмента SPC Estimate Professional, получившего приз Software Development Productivity. В 1998 году читатели журнала «Software Development» признали Стива одним из трех наиболее влиятельных людей в отрасли разработки ПО наряду с Биллом Гейтсом и Линусом Торвальдсом.

Стив получил степень бакалавра в колледже Уитмена и степень магистра по разработке ПО в Сиэтлском университете. Живет он в городе Беллвью, штат Вашингтон.

Если у вас возникнут какие-либо комментарии или вопросы по поводу этой книги, свяжитесь со Стивом по адресу *stevemcc@construx.com* или посредством сайта *www.stevemcconnell.com*.

**Стив Макконнелл**

# Совершенный код
# Мастер-класс

Перевод с английского под общей редакцией **В. Г. Вшивцева**

Главный редактор **А. И. Козлов**

Подготовлено к печати издательством «Русская редакция»
125362, Москва, ул. Свободы д. 17, а/я 14
тел.: (499) 197-04-22, e-mail: info@rusedit.com, http:www.rusedit.com

РУССКАЯ РЕДАКЦИЯ

Подписано в печать 21.07.2010 г. Дополнительный тираж 1 500 экз.
Формат 70×100/16. Физ. п. л. 56

Отпечатано по технологии CtP в ОАО «Печатный двор» им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15

# СОВЕРШЕННЫЙ КОД

## ПРАКТИЧЕСКОЕ РУКОВОДСТВО ПО РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ


**«Новичок вы или профессионал — эта книга научит вас лучшим подходам к программированию».**

*Джеффри Рихтер*

Более 10 лет первое издание этой книги считалось одним из лучших практических руководств по программированию. Сейчас эта книга полностью обновлена с учетом современных тенденций и технологий и дополнена сотнями новых примеров, иллюстрирующих искусство и науку программирования. Опираясь на академические исследования с одной стороны и практический опыт коммерческих разработок ПО с другой, автор синтезировал из самых эффективных методик и наиболее эффективных принципов ясное прагматичное руководство. Каков бы ни был ваш профессиональный уровень, с какими бы средствами вы ни работали, какова бы ни была сложность вашего проекта — в этой книге вы найдете нужную информацию, она заставит вас размышлять и поможет создать совершенный код.

***Изложенные в книге методики и стратегии помогут вам:***

- проектировать с минимальной сложностью и максимальной продуктивностью;
- извлекать выгоду из групповой разработки;
- применять методики защитного программирования, позволяющие избежать ошибок;
- совершенствовать свой код;
- применять методики конструирования, наиболее подходящие для вашего проекта;
- быстро и эффективно производить отладку;
- своевременно и быстро обнаруживать критические проблемы проекта;
- обеспечивать качество на всех стадиях проекта.

**Об авторе**

***Стив Макконнелл — признанный авторитет и известнейший автор в сообществе разработчиков. Он занимает должность главного разработчика ПО в компании Construx Software и является автором таких популярных и авторитетных книг, как «Rapid Development», «Software Project Survival Guide» и «Professional Software Development».***

*«Стив Макконнелл — один из немногих людей, знающий истинное положение дел в нашей отрасли и способный его разъяснить».*

*Джон Влиссидес, IBM Research*

ISBN 978-5-7502-0064-1

9 785750 200641

Издательство
**Русская Редакция**
Москва
3-я Хорошевская ул., 11
E-mail: info@rusedit.com
Internet: www.rusedit.com
Тел.: (499) 197-0422

Microsoft