O'REILLY®

# Разработка обслуживаемых программ на языке C#

*Джуст Виссер*

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Джуст Виссер

# Разработка обслуживаемых программ на языке C#

Joost Visser

# Building Maintainable Software (C# Edition)

## Ten Guidelines for Future-Proof Code

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY

**Джуст Виссер**

# Разработка обслуживаемых программ на языке C#

## Десять рекомендаций по оформлению современного кода

**Москва, 2017**

УДК 004.457
ББК 32.972.13
В53

Виссер Дж.
В53 Разработка обслуживаемых программ на языке C# / пер. с англ. Р. Н. Рагимова. — М.: ДМК Пресс, 2017. — 192 с.: ил.

ISBN 978-5-97060-446-5

Данное практическое руководство познакомит вас с 10 простыми рекомендациями, помогающими писать программное обеспечение, которое легко поддерживать и адаптировать. Эти тезисы сформулированы на основании анализа сотен реальных систем.

Написанная консультантами компании Software Improvement Group книга содержит ясные и краткие советы по применению рекомендаций на практике. Примеры для этого издания написаны на языке C#, но существует аналогичная книга с примерами на языке Java.

Издание предназначено программистам на C#, желающим научиться писать качественный и хорошо поддерживаемый код.

УДК 004.457
ББК 32.972.13



ISBN 978-1-491-95452-2 (анг.) © 2016 Software Improvement Group, B.V.
ISBN 978-5-97060-446-5 (рус.) © Оформление, издание, перевод, ДМК Пресс, 2017

# Содержание

# Об авторах

**Джуст Виссер** (Joost Visser) — научный руководитель Software Improvement Group (SIG). В этой роли он отвечает за научное обоснование методов и инструментов, предлагаемых компанией SIG для количественной оценки и улучшения программного обеспечения. Кроме того, Джуст занимает должность профессора кафедры крупномасштабных программных систем в университете Неймегена (Нидерланды). Получил докторскую степень в области компьютерных наук в университете Амстердама и опубликовал более 100 статей по таким темам, как обобщенное программирование, трансформация программ, эффективные вычисления, качество программного обеспечения, а также эволюция программного обеспечения. Джуст считает разработку программного обеспечения социально-технической дисциплиной и убежден, что количественная оценка программного обеспечения имеет важное значение для успешности команды разработчиков и владельцев программной продукции.

**Паскаль ван Экк** (Pascal van Eck) присоединился к Software Improvement Group в 2013 году в роли консультанта по общим вопросам качества программного обеспечения. До прихода в SIG в течение 13 лет был доцентом кафедры информационных систем в университете Твенте (Нидерланды). Имеет докторскую степень в области компьютерных наук, полученную в университете Врийе (Амстердам), и опубликовал более 80 статей в таких областях, как архитектура корпоративных приложений, компьютерная безопасность и количественная оценка программного обеспечения. Паскаль является председателем программного комитета голландской национальной конференции по архитектуре цифрового мира.

После получения степени магистра в области программной инженерии в техническом университете Делфта в 2005 году **Роб ван дер Лик** (Rob van der Leek) присоединился к SIG в роли консультанта по качеству программного обеспечения. Работая в SIG, Роб

ощущает себя врачевателем программного обеспечения. Как консультант он, опираясь на свои технические знания в области разработки программного обеспечения и программных технологий, дает клиентам советы по поддержанию их программных систем в хорошей форме. Кроме работы консультантом, Роб занимает ведущее положение в команде внутренних разработок SIG. Эта команда разрабатывает и обслуживает аналитическое программное обеспечение компании. Амбициозной мечтой Робу служит желание оставить IT-индустрию в состоянии, хоть немногим лучшем, чем то, в котором она была, когда он в нее вошел.

**Сильван Ригаль** (Sylvan Rigal) работает в SIG консультантом по вопросам качества программного обеспечения с 2011 года и дает клиентам советы по вопросам управления программным обеспечением с 2008 года. Он помогает клиентам добиться снижения затрат на обслуживание программного обеспечения и повышения его безопасности за счет совершенствования процессов проектирования и разработки программ. Имеет степень магистра в области международного бизнеса Маастрихтского университета (Нидерланды). Как активный член команды безопасности программного обеспечения SIG обучает консультантов анализу рисков безопасности программного обеспечения. Когда Сильван не занят оценкой технического здоровья программ, он совершенствуется в бразильском джиу-джитсу, с удовольствием посещает рестораны Амстердама или путешествует по Азии. Вот примерно в таком порядке.

**Гис Винхолдс** (Gijs Wijnholds) присоединился к Software Improvement Group в 2015 году в качестве консультанта по программному обеспечению в области государственного управления. Помогает клиентам управлять их программными проектами, консультируя по вопросам развития и перевода технических рисков в стратегические решения. Гис получил степень бакалавра в области искусственного интеллекта в Утрехтском университете (Нидерланды) и степень магистра логики в университете Амстердама. Является экспертом по языку программирования Haskell и математической лингвистике.

# Предисловие

В умении обходиться малым виден мастер.
*Иоганн Вольфганг фон Гёте*

После 15 лет консультирования по вопросам качества программного обеспечения мы в Software Improvement Group (SIG) усвоили два аспекта, касающихся обслуживаемости.

Во-первых, неудовлетворительная обслуживаемость является реальной проблемой в практике разработки программного обеспечения — разработчики тратят слишком много времени на обслуживание и исправление старого кода. Это оставляет меньше времени на наиболее плодотворную часть разработки программного обеспечения — написание нового кода. Наш собственный опыт, а также собранные нами данные свидетельствуют от том, что обслуживание исходного кода занимает, по крайней мере, в два раза больше времени, когда обслуживаемость находится на уровне ниже среднего, чем при уровне обслуживаемости выше среднего. С нашей методикой оценки обслуживаемости можно ознакомиться в приложении А.

Во-вторых, недостатки в обслуживаемости в значительной степени вызваны тривиальными просчетами в разработке, повторяемыми снова и снова. Следовательно, наиболее эффективный и действенный способ улучшить обслуживаемость заключается в исключении этих просчетов. Для этого не требуется никакого волшебства или чего-то из ряда вон выходящего. Сочетание относительно простых навыков, знаний плюс дисциплина и соответствующее окружение обеспечивают заметное повышение уровня обслуживаемости.

В компании SIG мы не раз сталкивались с системами, которые по существу являются непригодными для эксплуатации. В этих системах нет возможности исправить ошибки, а их функциональность не модифицируется или не расширяется, поскольку это требует слишком больших временных затрат и связано с огромными рис-

ками. К сожалению, такое явление не редкость в современной IT-индустрии, но ведь это недопустимо.

Именно поэтому мы сформулировали 10 рекомендаций. Мы хотим поделиться знаниями и навыками со всеми практикующими разработчиками программного обеспечения, чтобы помочь им писать обслуживаемый исходный код. Мы уверены, что после знакомства с этими 10 рекомендациями вы как разработчик программного обеспечения сможете писать обслуживаемый исходный код. После этого вам останется только сформировать окружение, в котором вы сможете применить эти навыки с максимальным эффектом, включая общие методики разработки, соответствующий инструментарий и многое другое. Основам среды разработки будет посвящена наша вторая книга *«Building Software Teams»*.

## Тема этой книги: Десять правил разработки обслуживаемого программного обеспечения

Изложенное в следующих главах применимо к любой системе разработки. Рекомендации касаются размеров блоков кода и количества параметров (методов в языке C#), числа точек ветвления, а также других свойств исходного кода. Это давно известные правила, о которых наверняка слышали многие программисты. В главах также приводятся примеры, как правило, в форме шаблонов рефакторинга кода, демонстрирующие применение этих рекомендаций на практике. Все примеры написаны на языке C#, но сами советы не зависят от используемого языка программирования. Восемь советов из 10 заимствованы из выработанных в SIG/TÜViT[1] критериев разработки для достижения уверенной обслуживаемости продукта[2] — комплекса характеристик для системной оценки обслуживаемости исходного кода.

[1] TÜViT является частью TÜV, всемирной организации контроля качества техники немецкого происхождения. Она специализируется на сертификации и консалтинге в области программного обеспечения, в частности в вопросах его безопасности.

[2] Более подробную информацию о критериях обслуживаемости можно найти на странице http://bit.ly/eval_criteria

## Почему следует прочесть эту книгу

Каждая из взятых по отдельности рекомендаций, представленных в этой книге, хорошо известна. Действительно, многие популярные инструменты анализа кода выполняют проверку представленных здесь рекомендаций. Например, инструменты Checkstyle (для Java, http://checkstyle.sourceforge.net/), StyleCop+ (для C#, **Error! Hyperlink reference not valid.**/), Pylint (для Python, http://www.pylint.org/), JSHint (для JavaScript, http://jshint.com/), RuboCop (для Ruby, **Error! Hyperlink reference not valid.**) и PMD (охватывает несколько языков, включая C# и Java, https://pmd.github.io/) проверяют код на соответствие рекомендации, представленной в главе 2. Однако три особенности, перечисленные ниже, выделяют эту книгу из числа других, посвященных разработке программного обеспечения:

*Мы отобрали 10 самых важных рекомендаций на основании собственного опыта*

Инструменты контроля стиля и статического анализа кода обычно слишком сложны. Версия Checkstyle 6.9 выполняет проверку с использованием порядка 150 правил. Все они полезны, но по-разному влияют на обслуживаемость. Мы отобрали 10, имеющих наибольшее влияние на обслуживаемость. Наш выбор обосновывается ниже, во врезке «Почему именно эти десять конкретных рекомендаций?».

*Мы научим применению этих 10 рекомендаций*

Простое перечисление, *что* следует делать программисту и чего не следует, — это одно (а многие руководства ограничиваются только этим). А описание, *как* следовать рекомендациям, — это совсем другое. В этой книге каждая рекомендация сопровождается конкретными примерами создания соответствующего ей кода.

*Мы приведем статистические данные и примеры, взятые из реальных систем*

В компании SIG мы просмотрели *массу* исходного кода, созданного реальными программистами в реальных условиях. Для этого исходного кода характерен компромиссный подход. Мы поделимся результатами своих исследований и покажем,

насколько реальный исходный код соответствует этим рекомендациям.

## Кому адресована эта книга

Эта книга адресована разработчикам программного обеспечения, которые уже умеют программировать на C#. Таких разработчиков можно разбить на две группы. Первая включает разработчиков, которые получили всестороннюю подготовку в области информатики или программной инженерии (например, обучаясь по соответствующей специальности в колледже или университете). Этим разработчикам наша книга поможет закрепить принципы, которым их обучали на вводных курсах программирования.

Вторая группа включает разработчиков, которые приступили к работе, не получив полноценного образования в области информатики или программной инженерии. Здесь имеются в виду разработчики, которые обучались самостоятельно или специализировались, учась в колледже или университете, в совершенно иной области, а затем сменили направление деятельности. Наш опыт показывает, что специалисты из этой группы, как правило, весьма слабо осведомлены в вопросах, выходящих за рамки синтаксиса и семантики используемого ими языка программирования. Именно на эту группу мы ориентировались при написании книги.

---

**Почему именно эти десять конкретных рекомендаций?**

В книге представлено 10 рекомендаций. Первые восемь напрямую связаны с так называемыми *системными свойствами* критериев оценки уверенной обслуживаемости готовых продуктов SIG/TÜViT, на которых основывается показатель обслуживаемости SIG. Из критериев оценки SIG/TÜViT мы выбрали показатели, которые:

- наиболее редко упоминаются;
- не зависят от технологии;
- легко измеряются;
- поддаются оценке в реальных системах корпоративного программного обеспечения.

В результате были отобраны восемь системных свойств, включенных в критерии оценки SIG/TÜVIT. К ним мы добавили две процессуальные рекомендации (касающиеся чистоты кода и автома-

тизации), которые посчитали самыми важными и поддающимися непосредственному контролю.

Специалисты в области информатики и программной инженерии весьма преуспели в определении характеристик исходного кода.

В зависимости от порядка подсчета были зарегистрированы десятки, если не сотни различных характеристик. Поэтому восемь используемых нами системных свойств — далеко не полный перечень характеристик обслуживаемости.

Но мы полагаем, что эти восемь характеристик SIG/TÜViT являются вполне адекватным и достаточным набором для оценки обслуживаемости, поскольку позволяют решить следующие задачи:

*Характеристики, зависящие от технологии*

Некоторые характеристики (например, глубина наследования) применимы только к исходному коду, привязанному к конкретным технологиям (например, только в объектно-ориентированных языках). В то время как доминирующий на практике объектно-ориентированный подход является далеко не единственным. Существует достаточно много исходного кода, не являющегося объектно-ориентированным (вспомните языки Cobol, RPG, C и Pascal), обслуживаемость которого также нужно оценить.

*Характеристики, сильно коррелирующие с другими*

Некоторые характеристики отличаются заметной корреляцией друг с другом. Примером может служить общее число точек ветвления в коде. Эмпирически доказано, что эта характеристика в значительной степени коррелирует с характеристикой, определяющей объем кода. Это значит, что если известно общее количество строк кода в системе (которое легко получить), с высокой точностью можно предсказать количество точек ветвления. Поэтому нет смысла учитывать сложную характеристику, поскольку на ее получение тратятся значительные усилия, а в результате не получается ничего, что нельзя было узнать с помощью более простой характеристики.

*Характеристики, которые невозможно дифференцировать на практике*

Некоторые характеристики хорошо выглядят в теории, но на практике все системы оценки оказываются примерно одинаковыми. Поэтому нет смысла включать их в модели оценки, поскольку полученные результаты будут неотличимы.

## Чего не будет в этой книге

Эта книга использует язык C# (и только C#) для демонстрации и пояснения предлагаемых рекомендаций. Но она не является учебником по языку C#. Предполагается, что читатель, по крайней мере, умеет читать код на C# и знаком с программным интерфейсом его стандартных библиотек. Мы постарались сделать примеры кода простыми, используя только базовые особенности этого языка.

Кроме того, эта книга не рассказывает об идиомах C#, то есть о характерных для C# приемах выражения функциональности в коде. Мы не считаем, что можно обеспечить обслуживаемость кода с помощью определяемых конкретным языком идиом. Наоборот, представленные здесь рекомендации в значительной степени не зависят от языка и, следовательно, не зависят от языковых идиом.

Несмотря на то что книга знакомит с некоторыми шаблонами рефакторинга кода, ее не следует рассматривать как всеобъемлющий каталог таких шаблонов. Существуют книги и сайты, которые с большим основанием могут претендовать на роль таких каталогов. Эта книга посвящена описанию *причин* и *процесса* применения ряда отобранных шаблонов рефакторинга для улучшения обслуживаемости кода. То есть эта книга служит лишь ступенькой на пути к таким каталогам.

## Какую книгу прочесть следующей

Нам известно, что отдельные разработчики не контролируют всех аспектов процесса разработки. Выбор средств разработки, организация контроля качества, порядок развертывания и так далее — все эти аспекты являются важными факторами, влияющими на качество программного обеспечения, но об этом должна заботиться команда разработчиков. Поэтому эти вопросы выходят за рамки данной книги. Они будут освещены в нашей следующей книге «Building Software Teams». В ней мы опишем передовой опыт в этой области и способы оценки его применения на практике.

## О компании Software Improvement Group

Хотя на обложке книге и указано только имя одного автора, в действительности авторов у этой книги гораздо больше. Настоящим автором является SIG, консалтинговая компания по управлению

программным обеспечением. То есть в этой книге объединены коллективный опыт и знания консультантов SIG, которые занимаются оценкой качества программного обеспечения и консультированием в этой области с 2000 года. Мы создали уникальную сертифицированную[1] лабораторию анализа программного обеспечения, выполняющую стандартизированные проверки программного обеспечения на соответствие международным стандартам качества ISO 25010.

Одной из служб компании SIG является наша служба мониторинга рисков программного обеспечения. Клиенты службы загружают исходный код через регулярные промежутки времени (обычно раз в неделю). Этот код автоматически проверяется в лаборатории анализа программного обеспечения. Все необычное, обнаруженное в процессе автоматического анализа, исследуется консультантами SIG и обсуждается с клиентами. На момент написания книги компания SIG проанализировала в общей сложности 7,1 миллиарда строк кода, кроме того, 72,7 миллиона новых строк кода выгружается в SIG еженедельно.

Компания SIG была создана в 2000 году. Ее корни можно проследить до Голландского национального научно-исследовательского института математики и информатики (Centrum voor Wiskunde en Informatica [CWI] на голландском языке). Даже по прошествии 15 лет мы поддерживаем и ценим наши связи с научным сообществом академической разработки программного обеспечения. Консультанты SIG регулярно вносят свой вклад в научные публикации, защитили несколько кандидатских диссертаций, основанных на исследованиях в области разработки и совершенствования модели качества SIG.

## Об этой версии книги

Эта версия книги основана на языке C#. Все примеры кода написаны на C# (и только на C#), в тексте книги часто встречаются названия инструментов и термины, широко используемые только в сообществе C#-разработчиков. Предполагается, что читатель имеет опыт программирования на языке C#. Однако, как уже упоминалось

[1] Имеется в виду сертификация ISO/IEC 17025.

ранее, рекомендации, представленные в данной версии книги, не зависят от используемого языка программирования. Версия для языка Java публикуется издательством O'Reilly (и будет издана издательством «ДМК Пресс») одновременно с этой книгой.

## Рекомендуемые книги

Здесь представлено 10 базовых рекомендаций для достижения высокого уровня обслуживаемости. Даже если это первая прочитанная вами книга, посвященная обслуживаемости, мы надеемся, что она не станет последней. Рекомендуем несколько книг для дальнейшего чтения:

*«Building Software Teams» от Software Improvement Group*

Дополнение к текущей книге, написанное теми же авторами. В то время как данная книга содержит рекомендации разработчикам по созданию обслуживаемого программного продукта, следующая книга освещает продвинутый процесс разработки и его оценку, основанную на подходе «цель — проблема — характеристика». Книга «Building Software Teams» планируется к публикации в 2016 году.

«*Refactoring: Improving the Design of Existing Code», автор Мартин Фаулер (Martin Fowler*)[1]

Эта книга посвящена способам улучшения обслуживаемости (и других качественных характеристик) существующего кода.

*«Clean Code: A Handbook of Agile Software Craftsmanship», автор Роберт С. Мартин (Robert C. Martin) (известный также как Uncle Bob)*[2]

Как и данная книга, книга «Clean Code» посвящена написанию высококачественного исходного кода. Содержит более абстрактные рекомендации.

*«Code Quality: The Open Source Perspective», автор Диомидис Спинел- лис (Diomidis Spinellis)*[3]

---

[1] *Фаулер М., Бек К., Брант Дж., Робертс Д., Апдайк У.* Рефакторинг: улучшение существующего кода. СПб.: Символ-Плюс, 2009. ISBN: 5-93286-045-6. — *Прим. ред.*

[2] *Мартин Р.* Чистый код: создание, анализ и рефакторинг. Библиотека программиста. СПб.: Питер, 2013. ISBN: 978-5-496-00487-9. — *Прим. ред.*

[3] *Спинеллис Д.* Анализ программного кода на примере проектов Open Source. М.: Вильямс,

Подобно этой книге, «Code Quality» предоставляет рекомендации по улучшению качества кода, но, как и в книге «Clean Code», они более абстрактны.

*«Design Patterns: Elements of Reusable Object-Oriented Software», авторы Эрик Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson) и Джон Влиссидес (John Vlissides) (эта группа авторов известна также как «Банда четырех» (Gang of Four))*[1]

Рекомендуем прочесть эту книгу тем разработчикам программного обеспечения, которые хотят стать отличными программными архитекторами.

## Соглашения, принятые в этой книге

В этой книге приняты следующие типографские соглашения:

*Курсив*

Используется для обозначения новых терминов, URL-адресов, адресов электронной почты, имен файлов и расширений имен файлов.

`Моноширинный`

Применяется для оформления листингов программ и программных элементов в тексте, таких как имена переменных, функций, баз данных, типов данных переменных окружения, операторов и ключевых слов.

Так обозначаются советы или рекомендации.

Так обозначаются общие примечания.

Этим значком отмечаются важные замечания.

2014. ISBN: 5-8459-0604-0. — *Прим. ред.*

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. СПб.: Питер, 2011. ISBN: 9785937000231. — *Прим. ред.*

## Обобщенные названия элементов исходного кода

Для иллюстрации рекомендаций по улучшению обслуживаемости в книге используется язык C#, но сами рекомендации не являются характерными исключительно для C#. Они основываются на модели обслуживаемости SIG, не зависящей от используемой технологии и применяемой для примерно ста языков программирования и связанных с ними технологий (таких как JSP).

Языки программирования отличаются своими возможностями и (естественно) синтаксисом. Например, если в языке Java для обозначения константы используется ключевое слово `final`, в C# применяется `readonly`, имеющее тот же смысл. Другой пример: язык C# поддерживает так называемые *разделяемые классы* (partial classes)[1], которые в языке Java (в настоящее время) не поддерживаются.

Помимо различий в синтаксисе, для разных языков программирования в учебниках, пособиях и спецификациях используются разные термины. Например, понятие группы строк кода, которые выполняются как единое целое, присутствует практически в каждом языке программирования. В языках Java и C# это понятие называют *методом.* В языке Visual Basic — *подпрограммой.* В языках JavaScript и C оно известно как *функция.* В языке Pascal — это *процедура.*

Именно поэтому независимая от технологий модель нуждается в обобщенном названии для понятия группы. В табл. П.1 приведены обобщенные названия, используемые в книге.

Следующий список поясняет связи между групповыми конструкциями в табл. П.1 и практическими аспектами программирования на языке C#:

[1] https://msdn.microsoft.com/ru-ru/library/wa80x488.aspx. — Прим, ред.

***Таблица П.1. Обобщенные понятия групп и их представление в C#***

| Обобщенное название | Общее определение | В языке C# |
|---|---|---|
| Блок кода | Наименьшая группа строк, которая может выполняться независимо | Метод или конструктор |
| Модуль | Наименьшая группа блоков кода | Класс верхнего уровня, интерфейс или перечисление |
| Компонент | Высокоуровневая часть системы, определяемая в архитектуре программного обеспе- | (Определение в языке отсутствует) |
| Система | Вся исследуемая кодовая база | (Определение в языке отсутствует) |

*От малого к большему*

Понятия групп в табл. П.1 упорядочены от меньших к большим. Блок кода состоит из операторов, но сам оператор не является группирующей конструкцией.

В языке C#, как и во многих других, существует сложная взаимосвязь между операторами и содержащими их строками в файле .cs. Строка может содержать несколько операторов, но и оператор может располагаться в нескольких строках. Здесь применяется понятие строки кода (Line of Code, LoC), то есть любой строки исходного кода, заканчивающейся символами CR/LF, но не пустой и не содержащей только комментарий.

*Некоторые понятия могут быть не определены в конкретном языке*

Как видно из табл. П.1, некоторые из обобщенных понятий отсутствуют в языке C#. Например, в C# не существует синтаксиса для описания границ системы. Это понятие присутствует только как обобщенный термин, поскольку оно нам будет необходимо. Это не является недостатком синтаксиса C#, поскольку на практике эти границы определяются другими способами.

*Некоторые из общеупотребительных терминов не используются*

Возможно, вас удивило, что табл. П.1 не содержит таких популярных терминов, как субкомпоненты и подсистемы. Причина проста — они не используются в рекомендациях.

*Представлены не все группирующие конструкции языка*

В C# имеется больше группирующих конструкций, часть из которых отсутствует в табл. П.1. Например, для группировки классов и интерфейсов C# поддерживает пространства имен. Он также поддерживает вложенные классы. Они не перечислены в табл. П.1, поскольку не применяются в рекомендациях. Например, нет необходимости разделять обычные и вложенные классы в формулировке рекомендаций, касающихся взаимодействий.

Пространство имен в C# нельзя отождествлять с обобщенным термином «компонент». В малых C#-системах компоненты и пространства имен могут иметь однозначное соответствие. Но в больших C#-системах пространств имен, как правило, гораздо больше, чем компонентов.

*Инструменты сборки никак не отражены в общей терминологии*

При разработке на языке C# интегрированная среда Visual Studio предоставляет дополнительное понятие группы: цели сборки. Тем не менее цели сборки не играют никакой роли в рекомендациях. В некоторых ситуациях можно однозначно соотнести компоненты с целями сборками, но это не является обязательным правилом.

*Компоненты определяются архитектурой системы*

Компоненты не являются понятием языка C#. Компоненты — это не пространства имен, не цели сборки и не решения в Visual Studio. Компоненты — это высокоуровневые строительные блоки, определяемые архитектурой программной системы. Они соответствуют блокам в блок-схеме системы. Более подробно понятие «компонент» будет описано и проиллюстрировано примерами в главе 7.

## Получение примеров кода

Сопроводительные материалы к книге (примеры кода, упражнения и т. д.) можно загрузить на странице: https://github.com/oreillymedia/building_maintainable_software.

Данная книга призвана оказать вам помощь в решении ваших задач. Вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не

нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию необходимо получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: *«Building Maintainable Software: Ten Guidelines for Future-Proof Code by Joost Visser. Copyright 2016 Software Improvement Group B.V., 978-1-4919-5352-5*».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу permissions@oreilly.com.

## Электронная библиотека Safari® Books Online


Safari Books Online — это виртуальная библиотека, содержащая авторитетную информацию в виде книг и видеоматериалов, созданных ведущими специалистами в области технологий и бизнеса.

Профессионалы в области технологии, разработчики программного обеспечения, веб-дизайнеры, а также бизнесмены и творческие работники используют Safari Books Online как основной источник информации для проведения исследований, решения проблем, обучения и подготовки к сертификационным испытаниям.

Библиотека Safari Books Online предлагает широкий выбор продуктов и тарифов для организаций, правительственных и учебных учреждений, а также физических лиц.

Подписчики имеют доступ к поисковой базе данных, содержащей информацию о тысячах книг, видеоматериалов и рукописей от таких издателей, как O'Reilly Media, Prentice Hall Professional, Addison-Wesly Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, и десятков других. За подробной информацией о Safari Books Online обращайтесь по адресу http://www.safaribooksonline.com/.

## Как связаться с нами

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North Sebastopol, CA 95472
800-998-9938 (США или Канада)
707-829-0515 (международный и местный)
707-829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на странице книги: http://bit.ly/modern_php.

Свои пожелания н вопросы технического характера отправляйте по адресу: bookquestions@oreilly.com.

Ищите нас в Facebook: http://facebook.com/oreilly.

Следуйте за нами в Twitter: http://twitter.com/oreillymedia.

Смотрите нас на YouTube: http://www.youtube.com/oreillymedia.

## Благодарности

Мы хотим выразить благодарность за помощь в работе над книгой:

- Яннису Канеллопоулосу (Yiannis Kanellopoulos) (SIG), руководителю нашего проекта, контролирующему все и вся;
- Тобиасу Кейперсу (Tobias Kuipers), инициатору этого проекта;
- Лодевику Бергмансу (Lodewijk Bergmans) (SIG) за помощь в разработке структуры книги;
- Зигеру Любсену (Zeeger Lubsen) (SIG) за тщательную проверку;

- Эду Лоуверсу (Ed Louwers) (SIG) за помощь с иллюстрациями к книге;
- всем сотрудникам SIG (в том числе и бывшим), которые работают и работали над совершенствованием моделей для оценки, сравнительного анализа и интерпретации качества программного обеспечения.

Со стороны издательства O'Reilly:

- Нану Барберу (Nan Barber), рецензенту;
- Киту Конанту (Keith Conant), техническому рецензенту.

И Арье ван Деурсен (Arie van Deursen) за разрешение включить в книгу фрагменты кода из игры *JPacman.*

# Глава 1

# Введение

Кто написал этот фрагмент кода??
Я не мог этого сделать!!

*Любой программист*

Здорово быть разработчиком программного обеспечения. Вам ставят задачи и определяют требования, а вы должны придумать решение и перевести его на понятный компьютеру язык. Это сложная и хорошо вознаграждаемая работа. Но разработка программного обеспечения нередко превращается в весьма кропотливую работу. Если вам регулярно приходится вносить изменения в исходный код, написанный другими (или даже вами), вы уже знаете, что это может быть как очень простым, так и очень сложным занятием. Иногда строки кода, которые необходимо изменить, находятся очень быстро. Изменения полностью изолированы, а тесты подтверждают, что всё работает, как нужно. Но бывают ситуации, когда единственным выходом является использование обходных путей, что создает больше проблем, чем решает их.

Простоту или сложность внесения изменений в программную систему обычно называют *обслуживаемостью*. Обслуживаемость программной системы определяется свойствами ее исходного кода. Эта книга посвящена рассмотрению этих свойств и представляет 10 рекомендаций, которые помогут писать легко изменяемый исходный код.

В данной главе мы проясним, что понимается под обслуживаемостью. Затем обсудим важность обслуживаемости. Это создаст основу для перехода к главной теме книги: как разрабатывать изначально обслуживаемое программное обеспечение. В конце этого введения мы перечислим типичные заблуждения, касающиеся об-

служиваемости, и принципы, лежащие в основе 10 предлагаемых в книге рекомендаций.

## 1.1. Что такое обслуживаемость?

Представьте две различные программные системы, имеющие тождественную функциональность. Для одних и тех же входных данных они вернут одинаковый результат. Одна из этих систем работает быстро, дружественно настроена к пользователю, и вносить изменения в ее исходный код не составляет особого труда. Другая — медленная, сложная в использовании, и в ее исходном коде практически невозможно разобраться, уж не говоря о том, чтобы что-то в нем менять. Несмотря на идентичную функциональность обеих систем, их качество явно отличается.

Обслуживаемость (простота внесения изменений в систему) является одной из качественных характеристик программного продукта. Производительность (скорость вычисления результата) — это совершенно иная характеристика.

Международный стандарт ISO/IEC 25010:2011 (который в этой книге будет упоминаться просто как ISO 25010[1]) различает восемь характеристик программного обеспечения: *обслуживаемость, функциональная пригодность, эффективность работы, совместимость, удобство использования, надежность, безопасность* и *переносимость*. Эта книга посвящена исключительно обслуживаемости.

Несмотря на то что стандарт ISO 25010 не определяет способов оценки качества программного обеспечения, это не значит, что его невозможно количественно оценить. В приложении А описан порядок количественной оценки качества программного продукта, принятый в компании Software Improvement Group (SIG) и соответствующий стандарту ISO 25010.

[1] Полное наименование стандарта: International Standard ISO/IEC 25010. Systems and Software Engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models. First Edition, 2011-03-01.

### Четыре вида обслуживаемости программного обеспечения

Обслуживание программного обеспечения не связано с его поломками или износом. Программное обеспечение не является физической сущностью и, следовательно, не склонно к износу, как это характерно для физического оборудования. Тем не менее большинство программных систем постоянно модернизируется после ввода в эксплуатацию. Это и называется обслуживанием программного обеспечения. Можно выделить следующие четыре вида обслуживания программного обеспечения:

- устранение выявленных ошибок (так называемое *корректирующее обслуживание*);
- адаптация системы к изменениям в окружающей среде, где она функционирует, например при обновлении операционной системы или смене технологий (называется *адаптивным обслуживанием*);
- обеспечение изменения требований пользователей системы и/или других заинтересованных сторон (это *улучшающее обслуживание*);
- повышение качества или предотвращение будущих ошибок *(профилактическое обслуживание).*

## 1.2. Почему так важна обслуживаемость?

Как упоминалось выше, обслуживаемость — это лишь одна из восьми характеристик качества программного продукта, определенных в стандарте ISO 25010. Так почему же обслуживаемость так важна, что заслуживает отдельной книги? На этот вопрос имеются два ответа:

- обслуживаемость или отсутствие таковой оказывает существенное влияние на деловую сторону вопроса;
- обслуживаемость обеспечивает улучшение других характеристик качества.

Оба ответа поясняются в следующих двух разделах.

### Обслуживаемость значительно влияет на деловую сторону вопроса

В области разработки программного обеспечения период обслуживания программной системы обычно длится 10 и более лет. Большую часть этого времени постоянно возникают вопросы, которые нужно решать (корректирующее и адаптивное обслуживание), и появляются новые требования, которые следует удовлетворять (улучшающее обслуживание). Эффективность и результативность решения вопросов и реализация усовершенствований волнуют все заинтересованные стороны.

Сложность обслуживания невысока, когда решение вопросов и реализация усовершенствований даются просто и быстро. Кроме того, если эффективность обслуживания позволяет сократить обслуживающий персонал (число разработчиков), то при этом снижаются затраты на обслуживание. Если число разработчиков не уменьшается, при высокой эффективности обслуживания, у них появляется больше времени на выполнение других задач, таких как создание новых функциональных возможностей. Возможность быстрого усовершенствования означает уменьшение срока выхода на рынок новых продуктов и услуг, поддерживаемых системой. Если решение вопросов и реализация усовершенствований даются с трудом и требуют времени, сроки не выдерживаются, и система может быть признана непригодной к эксплуатации.

Компания SIG собрала эмпирические доказательства, что в системах с обслуживаемостью на уровне выше среднего решение возникающих вопросов и реализация усовершенствований занимают в два раза меньше времени, чем в системах с обслуживаемостью на уровне ниже среднего. Ускорение в два раза считается очень значительным в практике корпоративных систем. Время, необходимое для решения вопросов и внесения улучшений, составляет от нескольких дней до нескольких недель. Важность такого улучшения заключается не в том, что исправляется 5 или 10 ошибок в час, а, например, в том, первой ли компания вышла на рынок с новым продуктом или ее опередили конкуренты на несколько месяцев.

И это именно та разница между обслуживаемостью выше и ниже среднего уровня. Работая в компании SIG, мы сталкивались со

вновь созданными системами, обслуживаемость которых была настолько низка, что делала невозможным изменение системы еще до момента ввода ее в эксплуатацию. Изменения вносили ошибок больше, чем исправляли. Разработка — достаточно длительный процесс, в течение которого бизнес-окружение (а следовательно, и требования пользователей) меняется. Необходимо вносить коррективы, которые приводят к появлению новых ошибок. Чаще всего от таких систем отказываются еще до достижения ими версии 1.0.

### Обслуживаемость обеспечивает улучшение других качественных характеристик

Еще одна причина, почему обслуживаемость является особым аспектом качества программного обеспечения, заключается в том, что она служит стимулятором для других характеристик качества. Если система обладает высоким уровнем обслуживаемости, это облегчает проведение работ для улучшения других ее качеств, например исправление ошибок, связанных с безопасностью. В конечном счете оптимизация программной системы требует внесения изменений в ее исходный код, направленных на улучшение производительности, функциональной пригодности, безопасности или любой другой из семи прочих характеристик, определяемых стандартом ISO 25010.

Порой требуются лишь небольшие, локальные изменения. Но иногда изменения влекут коренную перестройку. Любые изменения сводятся к поиску определенного фрагмента кода, анализу, изучению внутренней логики, определению места в автоматизируемом бизнес-процессе, отслеживанию зависимостей относительно других частей кода, тестированию и перемещению дальше по конвейеру разработки. В любом случае, в системе с высоким уровнем обслуживаемости проще вносить изменения, что ускоряет оптимизацию качества системы. Например, легко обслуживаемый код стабильнее, чем необслуживаемый, поскольку внесение в него изменений влечет меньшее количество случайных побочных эффектов, чем в запутанный код, который трудно анализировать и тестировать.

## 1.3. Три принципа, на которых основаны рекомендации

Если обслуживаемость так важна, тогда как можно ее улучшить? В этой книге представлено 10 рекомендаций, следование которым обеспечит высокую обслуживаемость кода. Они будут представлены и рассмотрены в следующих главах. А в этой мы познакомимся с принципами, лежащими в их основе:

1) рекомендации должны быть простыми;
2) не следует вспоминать об обслуживаемости после окончания разработки, нужно уделять ей должное внимание с самого начала. Здесь важен вклад каждого разработчика;
3) не все отступления от рекомендаций дают одинаковый отрицательный эффект. Чем точнее программная система соответствует рекомендациям, тем выше уровень ее обслуживаемости.

Ниже приведены пояснения к этим принципам.

### Принцип 1: рекомендации должны быть простыми

Многие полагают, что для обеспечения обслуживаемости требуется что-то вроде «серебряной пули», то есть какой-то один способ или принцип, автоматически решающий вопрос обслуживаемости раз и навсегда. Мы придерживаемся противоположной точки зрения: высокий уровень обслуживаемости обеспечивается неуклонным следованием очень простым рекомендациям. Следование рекомендациям гарантирует достаточный, но не максимальный уровень обслуживаемости (что бы под этим не понималось). Приведение исходного кода в соответствие с рекомендациями повышает его обслуживаемость. Но в какой-то момент рост уровня обслуживаемости становится все меньше и меньше, а затраты растут все выше и выше.

### Принцип 2: применение рекомендаций с самого начала и значимость вклада каждого разработчика

Обслуживаемостью надо заниматься с первых дней работы над проектом. Понятно, что трудно оценить влияние отдельного «нарушения» рекомендаций из этой книги на общую обслуживае-

мость системы. Именно поэтому все разработчики должны быть дисциплинированными и следовать рекомендациям по улучшению обслуживаемости системы в целом — индивидуальный вклад каждого имеет большое значение.

Следование рекомендациям, приведенным в этой книге, не только поможет писать легко обслуживаемый код, но и послужит достойным примером коллегам-разработчикам. Это позволит избежать «эффекта разбитых окон», возникающего из-за того, что кто-то расслабился и пошел по пути наименьшего сопротивления. Подающий правильный пример не обязательно должен быть самым опытным, скорее самым дисциплинированным.

Не забывайте, что вы пишете код не только для себя, но и для менее опытных разработчиков, которые придут после вас. Эта мысль поможет вам применять простые решения при программировании.

### Принцип 3: не все отступления от рекомендаций дают одинаковый отрицательный эффект

Изложенные в этой книге рекомендации представляют измеримые пороговые значения в форме абсолютных правил. Например, в главе 2 рекомендуется никогда не писать методы, содержащие более 15 строк кода. Мы прекрасно понимаем, что на практике всегда присутствуют исключения из основного правила. Что, если фрагмент исходного кода нарушает одну или несколько из этих рекомендаций? Многие виды инструментов оценки качества программного обеспечения предполагают, что никакое нарушение недопустимо. Подразумевается, что все нарушения должны быть устранены. На практике исключение всех нарушений не является необходимостью и часто бесполезно. Подход «все или ничего» по отношению к нарушениям способен лишь заставить разработчиков вообще игнорировать их все.

Рассмотрим другой подход. Чтобы сохранить простоту и практичность оценки, мы определяем качество всей базы кода не по количеству нарушений, а по их *профилю качества.* Профиль качества делит оценки на отдельные категории, начиная от совершенного кода до кода с серьезными нарушениями. Используя профили качества, можно отделить умеренные нарушения (например, метод

с 20 строками кода) от серьезных (например, метод с 200 строк кода). После обсуждения в следующем разделе типичных заблуждений, касающихся обслуживаемости, мы поясним порядок использования профилей качества для оценки обслуживаемости систем.

## 1.4. Заблуждения относительно обслуживаемости

В этом разделе обсуждается несколько заблуждений, касающихся обслуживаемости, часто встречающихся на практике.

### Заблуждение: обслуживаемость зависит от языка программирования

*«При разработке системы используется современный язык программирования. Поэтому уровень ее обслуживаемости не может быть хуже, чем у любой другой системы».*

Имеющиеся у компании SIG данные не подтверждают, что применяемая технология (язык программирования) является доминирующим фактором, определяющим обслуживаемость системы. Нам известны примеры систем, написанных на языке C#, как с очень высоким, так и с очень низким уровнем обслуживаемости. В среднем обслуживаемость всех проверенных нами C#-систем совпадает со средним уровнем обслуживаемости, то же самое справедливо и для систем, написанных на языке Java. Это показывает, что на языках C# (или Java) можно создавать отлично обслуживаемые системы, но сам по себе выбор этих языков не гарантирует высокого уровня обслуживаемости. Очевидно, существуют другие факторы, определяющие уровень обслуживаемости.

Для единообразия во всей книге используются фрагменты кода, написанные на языке C#. Однако наши рекомендации применимы не только к языку C#. На основе рекомендаций и показателей, содержащихся в книге, компания SIG провела тестирование систем, написанных на более чем ста языках программирования.

### Заблуждение: обслуживаемость зависит от прикладной области

*«Моя команда разрабатывает встроенное программное обеспечение для автомобильной промышленности. Здесь обслуживаемость оценивается по-своему».*

Мы считаем, что наши рекомендации применимы при разработке всех видов программного обеспечения: встроенного, игрового, научного, таких программных компонентов, как компиляторы и СУБД, а также программного обеспечения для администрирования. Конечно, между этими прикладными областями существуют различия. Например, для создания научного программного обеспечения часто используются языки программирования специального назначения, такие как R, применяемый для статистического анализа. Тем не менее и при использовании языка R имеет смысл делать блоки кода короткими и простыми. Встроенное программное обеспечение должно работать в условиях, где существенное значение имеет предсказуемая производительность, а ресурсы ограничены. Поэтому всякий раз, когда приходится выбирать между производительностью и обслуживаемостью, предпочтение отдается первому в ущерб последнему. Но и в этой прикладной области применимы характеристики стандарта ISO 25010.

### Заблуждение: обслуживаемость гарантирует отсутствие ошибок

*«Вы утверждаете, что система имеет уровень обслуживаемости выше среднего. Но оказалось, что в ней полно ошибок!»*

В соответствии с определениями стандарта ISO 25010 система может иметь высокий уровень обслуживаемости, но при этом у нее могут быть низкими другие характеристики качества. То есть система, обладающая уровнем обслуживаемости выше среднего, может вместе с тем страдать от проблем, связанных с функциональной пригодностью, производительностью, надежностью, и многих других. Высокий уровень обслуживаемости означает лишь, что внесение изменений для уменьшения количества ошибок может быть проделано с высокой степенью эффективности и результативности.

### Заблуждение: обслуживаемость оценивается одной из двух альтернатив

*«Моя команда неоднократно исправляла ошибки в системе. Следовательно, она является обслуживаемой».*

Важное отличие. Под термином «обслуживаемость» понимается простота обслуживания. Согласно определению в стандарте ISO 25010, обслуживаемость исходного кода не характеризуется одним из двух альтернативных значений. Напротив, она оценивается степенью эффективности и результативности внесения изменений. Поэтому правильной постановкой вопроса является не «какие были сделаны изменения (например, исправлены ошибки)», а «сколько усилий было затрачено на исправление ошибок (эффективность) и были ли действительно устранены ошибки (результативность)?».

Основываясь на определении обслуживаемости в стандарте ISO 25010, можно утверждать, что программные системы никогда не бывают ни максимально обслуживаемыми, ни совершенно не обслуживаемыми. В компании SIG мы сталкивались с системами, которые можно считать практически не обслуживаемыми. Эти системы имеют такую низкую степень эффективности и результативности внесения изменений, что их владельцы не могли позволить себе продолжать их эксплуатацию.

## 1.5. Рейтинг обслуживаемости

Теперь мы знаем, что обслуживаемость является качественной характеристикой, определяемой по шкале. Она обозначает степень возможности обслуживать систему. Но как определить простоту или сложность обслуживания? Очевидно, что сложную систему проще будет обслуживать эксперту, чем менее опытному разработчику. В компании SIG при ответе на этот вопрос учитываются параметры эталонных систем индустрии разработки программного обеспечения. Если параметры программной системы ниже, чем средние для эталонных систем, ее трудно обслуживать. Калибровка параметров эталонных систем производится один раз в год. По мере того как в индустрии растет эффективность написания кода (например, при применении новых технологий), среднее значение параметров эталонных систем увеличивается. То, что было нормой

в программной инженерии несколько лет назад, не применимо в настоящее время. Таким образом, показатели эталонных систем отражают состояние улучшения технологий в области программной инженерии.

В компании SIG для обозначения рейтинга систем используются звезды, от 1 (сложное обслуживание) до 5 (простое обслуживание). Распределение систем по этим рейтингам от 1 до 5 звезд составляет 5% — 30% — 30% — 30% — 5%. То есть системам, попавшим в число 5% лучших, присваивается 5 звезд. Безусловно, в этих системах имеются нарушения рекомендаций, но их гораздо меньше, чем в системах с более низким рейтингом.

Рейтинги в виде звезд обеспечивают предсказание реального уровня обслуживаемости. В компании SIG было получено эмпирическое доказательство, что в системах с 4 звездами решение проблем и внесение усовершенствований осуществляются в два раза быстрее, чем в системах с 2 звездами.

Эталонные системы оцениваются на основании характеризующих их профилей качества. На рис. 1.1 приведены три примера профилей качества, основанных на оценке размеров блоков кода (читатель может найти цветные рисунки этого и других профилей качества в репозитории с файлами к этой книге: https://github.com/oreillymedia/building_maintainable_software).


**Рис. 1.1.** Пример трех профилей качества.

На первой диаграмме показан профиль качества по размерам блоков кода популярного сервера непрерывной интеграции с открытым исходным кодом Jenkins версии 1.625. Профиль качества показывает, что 64% всего кода сервера Jenkins находится в методах

длиной не более 15 строк (соответствует рекомендации). Профиль также сообщает, что 18% кода включено в методы с длиной от 16 до 30 строк и 12% — в методы, содержащие от 31 до 60 строк. Как видите, система Jenkins несовершенна. Серьезное нарушение рекомендации о размерах блоков кода наблюдается в 6% очень длинных блоков кода (более 60 строк кода).

На второй диаграмме показан профиль качества системы с рейтингом в 2 звезды. Обратите внимание, что более трети кода находится в блоках с длиной более 60 строк. Обслуживание этой системы станет весьма трудоемкой работой.

И наконец, на третьей диаграмме показано распределение блоков кода по размерам для системы с 4 звездами. Сравните эту диаграмму с первой. Можно сказать, система Jenkins соответствует рекомендации, касающейся размеров блоков кода, на 4 звезды, поскольку проценты в каждой из категорий ниже, чем у 4-звездной системы.

В конце каждой главы, посвященной одной из рекомендаций, будут представлены примеры диаграмм профилей качества для соответствующей рекомендации, используемых в компании SIG для оценки обслуживаемости. В частности, для каждой рекомендации будут приведены граничные точки и максимальный процент для категорий с рейтингом от 4 звезд или выше (лучшие 35%).

## 1.6. Обзор рекомендаций по улучшению обслуживаемости

В следующих главах все рекомендации будут рассмотрены отдельно, а сейчас просто перечислим их. Рекомендуем читать главы последовательно, продолжив чтение с главы 2.

*Пишите короткие блоки кода (глава 2)*

Короткие блоки кода (то есть методы и конструкторы) легче анализировать, тестировать и многократно использовать.

*Пишите простые блоки кода (глава 3)*

Блоки кода с меньшим количеством ветвлений проще анализировать и тестировать.

*Не повторяйте один и тот же код (глава 4)*

Всегда и везде следует избегать дублирования исходного кода, поскольку при необходимости изменений их придется произвести во всех копиях. Кроме того, дублирование приводит к ошибкам при откате назад.

*Стремитесь к уменьшению размеров интерфейсов (глава 5)*

Блоки кода (методы и конструкторы) с меньшим числом параметров проще тестировать и многократно использовать.

*Разделяйте задачи на модули (глава 6)*

В слабо связанные модули (классы) легче вносить изменения, и такое разделение улучшает модульность системы.

*Избегайте тесных связей между элементами архитектуры (глава 7)*

Слабая связанность компонентов верхнего уровня облегчает внесение изменении и придает системе модульность.

*Стремитесь к сбалансированности архитектуры компонентов (глава 8)*

Хорошо сбалансированная архитектура, характеризующаяся не слишком большим и не слишком малым числом компонентов примерно одинакового размера, обеспечивает достаточный уровень модульности и облегчает внесение изменений на основании изоляции задач.

*Следите за размером базы кода (глава 9)*

Огромные системы трудно обслуживать, поскольку для этого требуется анализировать, изменять и тестировать большие объемы кода. Кроме того, эффективность обслуживания в пересчете на каждую строку кода в большой системе значительно ниже, чем в малой.

*Автоматизируйте конвейер разработки и тестирования (глава 10)*

Автоматизированные тесты (то есть тесты, запускаемые без ручного вмешательства) позволяют почти мгновенно получить сведения об эффективности произведенных изменений. Ручные тесты не поддерживают масштабирования.

*Пишите чистый код (глава 11)*

Наличие в коде бесполезных артефактов, таких как незаконченные или неиспользуемые фрагменты, затрудняет его понимание новыми членами команды. Это снижает эффективность обслуживания.

# Глава 2

# Пишите короткие блоки кода

Любой дурак сможет написать код,
понятный компьютеру.
Хорошие программисты пишут код, в
котором могут разобраться люди.
*Мартин Фаулер (Martin Fowler)*

Рекомендация:
- **ограничивайте длину блоков кода 15 строками;**
- при этом или **изначально не пишите блоки кода длиной более 15 строк**, или **разделяйте длинные блоки на несколько более коротких, чтобы каждый содержал не более 15 строк**;
- это улучшает обслуживаемость, поскольку **небольшие блоки кода проще понять, тестировать и многократно использовать**.

Блоки — это наименьшие фрагменты кода, которые можно изолированно обслуживать и выполнять. В языке C# под блоками кода понимаются методы и конструкторы. Блок кода всегда выполняется как одно целое. Нельзя выполнить только часть строк в блоке. Таким образом, наименьшим фрагментом кода, который можно повторно использовать и тестировать, является блок.

Рассмотрим фрагмент кода, приведенный ниже. Если в URL-адресе указать идентификатор клиента, этот код сгенерирует список всех его банковских счетов вместе с их сальдо. Список возвращается в виде строки в формате JSON, включающей также общее сальдо по всем счетам. Код проверяет правильность номеров банковских счетов с помощью контрольных сумм и пропускает недопустимые номера. Врезка «Проверка контрольной суммы номеров банковских счетов на кратность 11» поясняет использование контрольной суммы.

### Проверка контрольной суммы номеров банковских счетов на кратность 11

Проверка на кратность 11 используется для проверки нидерландских девятизначных номеров банковских счетов. Контрольная сумма вычисляется как взвешенная сумма девяти цифр, составляющих номер счета. Роль весовых коэффициентов играют номера позиций цифр в номере банковского счета, справа налево. Крайняя левая цифра имеет весовой коэффициент 9, а крайняя правая цифра — весовой коэффициент 1. Номер банковского счета признается допустимым, только если полученная контрольная сумма кратна 11. Контрольная сумма позволяет обнаруживать ошибочные номера, в которых одна из цифр номера банковского счета указана неправильно.

В качестве примера рассмотрим номер банковского счета **12.34.56.789.** Основываясь на цифрах номера банковского счета, выделенных жирным, рассчитаем его контрольную сумму, двигаясь слева направо: (**1** × 9) + (**2** × 8) + (**3** × 7) + (**4** × 6) + (**5** × 5) + (**6** × 4) + (**7** × 3) + (8 × 2) + (9 × 1) = 165. Контрольная сумма правильная, поскольку 165 = 15 × 11.

```
public void DoGet(HttpRequest req, HttpResponse resp)
{
  resp.ContentType = "application/json";
  string command = "SELECT account, balance " +
    "FROM ACCTS WHERE id=" + req.Params[
      ConfigurationManager.AppSettings["request.parametername"]];
  SqlDataAdapter dataAdapter = new SqlDataAdapter(command,
    ConfigurationManager.AppSettings["handler.serverstring"]);
  DataSet dataSet = new DataSet();
  dataAdapter.Fill(dataSet, "ACCTS");
  DataTable dataTable = dataSet.Tables[0];
  try
  {
    float totalBalance = 0;
    int rowNum = 0;
    resp.Write("{\"balances\":[");
    while (dataTable.Rows.GetEnumerator().MoveNext())
    {
      rowNum++;
      DataRow results = (DataRow)dataTable.Rows.GetEnumerator().Current;
      // Предполагается, что на входе имеется 9-значный номер счета,
      // который требуется проверить с помощью контрольной суммы:
      int sum = 0;
      for (int i = 0; i < ((string)results["account"]).Length; i++)
      {
```

```
        sum = sum + (9 - i) *
          (int)Char.GetNumericValue(((string)results["account"])[i]);
      }
      if (sum % 11 == 0)
      {
        totalBalance += (float)results["balance"];
        resp.Write($"{{\"{results["account"]}\":{results["balance"]}}}");
      }
      if (rowNum == dataTable.Rows.Count)
      {
        resp.Write("],\n");
      }
      else
      {
        resp.Write(",");
      }
    }
    resp.Write($"\"total\":{totalBalance}}}\n");
  }
  catch (SqlException e)
  {
    Console.WriteLine($"SQL exception: {e.Message}");
  }
}
```

Чтобы разобраться в этом коде, необходимо пояснить большое количество деталей. Во-первых, здесь предполагается использовать ADO. NET-соединение. Далее следует цикл `while`, выполняющий обход записей, возвращаемых SQL-запросом, вычисляющий контрольную сумму в цикле `for` и проверяющий ее. Кроме того, здесь также выполняются JSON-форматирование и обработка HTTP-запросов и ответов.

Цикл `for` в середине блока кода реализует вычисление контрольных сумм. Хотя идея достаточно проста, тем не менее такой код требует тестирования. Это легче сказать, чем сделать, поскольку этот код можно проверить только вызовом метода `DoGet`. Для этого потребуется сначала создать объекты `HttpServletRequest` и `HttpServletResponse`. При этом понадобится настроить тестовую базу данных и заполнить ее тестовыми номерами счетов. После вызова метода `DoGet` вы получите строку в формате JSON, скрытую в объекте `HttpServletResponse`. Для проверки общего сальдо нужно будет извлечь это значение из строки в формате JSON.

Кроме того, код вычисления контрольной суммы сложно будет повторно использовать. Этот код можно выполнить, только вызвав метод `DoGet`. Следовательно, любой код, где потребуется повторно использовать код вычисления контрольной суммы, должен будет иметь базу данных, доступную через SQL-запросы, хранящую проверяемые номера банковских счетов.

Длинные блоки, как правило, сложно тестировать, многократно использовать, и в них трудно разобраться. Основная причина большого размера метода `DoGet` заключается в том, что он выполняет (по крайней мере) четыре разных действия: обрабатывает HTTP-запрос GET, извлекает номера счетов из базы данных, выполняет требуемую бизнес-логику и возвращает результаты в формате, пригодном для передачи, в данном случае это формат JSON. Всего метод `DoGet` содержит 46 строк кода. Более короткий блок просто не справится с таким количеством обязанностей.

Всякая непустая строка, содержащая не только комментарий, считается строкой кода. Подсчет строк кода начинается со строки, содержащей первую открывающую фигурную скобку.

## 2.1. Мотивация

Короткие блоки кода проще тестировать, анализировать и многократно использовать.

### Короткие блоки кода проще тестировать

Блоки кода инкапсулируют прикладную логику системы, и часто требуется приложить немалые усилия, чтобы проверить правильность логики приложения. Это связано с тем, что компилятор C# не обнаруживает автоматически ошибки, связанные с логикой приложения, не делают этого ни редакторы, ни интегрированные среды разработки (например, Visual Studio). Их может выявить только анализ кода. Вообще, короткий блок может выполнить лишь одну-единственную функцию, а длинные блоки способны взять на себя ответственность за выполнение нескольких функций. Блок, отвечающий за что-то одно, гораздо проще проверить, поскольку он решает одну неделимую задачу. Это позволяет применить к нему изолированный и простой тест (специфичный для каждого блока). Более подробно тестирование будет рассмотрено в главе 10.

### Короткие блоки кода проще анализировать

Чем короче блок, тем меньше времени займут чтение и анализ его внутреннего устройства. Это может быть не очевидно при написании нового кода, но явно проявляется при внесении изменений в написанный ранее код. А такая ситуация не редкость и возникает уже на следующий день после запуска проекта в эксплуатацию.

### Короткие блоки кода проще повторно использовать

Блок кода должен использоваться, по меньшей мере, в одном месте (в противном случае он считается неиспользуемым). Блок кода можно использовать многократно, вызывая его из разных методов. Короткие блоки кода лучше приспособлены для повторного использования, чем длинные. Длинные блоки, как правило, включают массу специфических деталей или вполне конкретную комбинацию функциональных возможностей. В результате они имеют более специальное назначение, чем короткие блоки. Это затрудняет их многократное использование, поскольку вероятность того, что специальная функциональность длинного блока понадобится где-то еще, не велика. Короткие блоки, напротив, оказываются полезными во многих случаях. Это облегчает их повторное использование, поскольку высока вероятность, что они потребуются где-то еще. Кроме того, многократное использование одного и того же кода уменьшает общий объем базы кода (более подробно об этом рассказывается в главе 9).

Копирование и вставка блоков кода в других местах — это совсем не то, что имеется в виду под многократным использованием. Такой вид повторного использования ведет к дублированию кода, которого следует избегать везде и всегда (более подробно об этом в главе 4).

## 2.2. Как применять рекомендацию

Следовать этой рекомендации не трудно, если владеть соответствующей методикой, но это требует определенной дисциплины. В этом разделе рассматриваются две, наиболее важные с нашей точки зрения методики. При написании нового блока никогда не позволяйте ему превысить предел в 15 строк. То есть задолго до приближения к 15 строкам следует начинать продумывать использо-

вание дополнительных возможностей. *На самом ли деле* все это должно быть помещено в один блок кода или можно что-то выделить в отдельный блок? Когда размер блока, несмотря на все ваши усилия, выходит за предел в 15 строк, поищите возможность сократить его.

### При написании нового блока кода

Предположим, что разрабатывается класс, реализующий уровень в игре CsPacMan, код которой используется во многих примерах в книге. Общее описание этой игры вы найдете во врезке «Об игре CsPacMan». Этот класс предоставляет общедоступные методы запуска и остановки игры, `Start` и `Stop`, вызываемые при нажатии кнопок в пользовательском интерфейсе. Уровень поддерживает список *наблюдателей,* то есть классов, которые уведомляются о завершении уровня.

Минимальная версия метода `Start` проверяет, запущена ли игра. Если запущена, метод завершается, ничего не делая, в противном случае устанавливает значение локальной переменной `inProgress`, предназначенной для хранения состояния игры:

```
public void Start()
{
  if (inProgress)
  {
    return;
  }
  inProgress = true;
}
```

Сейчас блок кода содержит только семь строк. В данный момент можно добавить модульный тест для этого блока. При использовании методики TDD (разработка, управляемая тестированием) на этот момент уже должен иметься модульный тест. Более подробно модульное тестирование рассматривается в главе 10.

---

### Об игре CsPacMan

В нескольких главах этой книги используется исходный код простой игры в стиле Pacman (рис. 2.1). Авторы игры называют этот код *JPacman Framework.* Фреймворк JPacman был создан профессором

Ари ван Дерсеном (Arie van Deursen) и его командой из технического университета Делфта для обучения идеям тестирования. Исходный код фреймворка JPacman открыт на условиях лицензии Apache 2.0 и доступен для загрузки на GitHub (http://bit.ly/jpacman). Для этой книги фреймворк JPacMan был переведен на язык C# и назван CsPacman. Его исходный код также доступен на GitHub (http://bit.ly/cspacman).

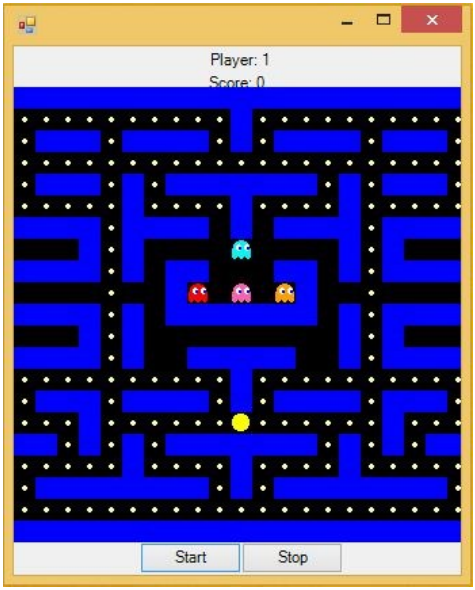


**Рис. 2.1** ❖ CsPacman, игра в стиле Pacman

### При добавлении в блок новых функциональных возможностей

При добавлении в систему новых возможностей размеры блоков начнут расти. Но рекомендация требует строго ограничивать их размер. Например, добавим в метод Start рассылку уведомлений всем наблюдателям об изменении текущего состояния. Для этого включим код, сообщающий всем наблюдателям, что уровень проигран, если игрок уничтожен, или выигран, если игрок собрал все гранулы:

```
public void Start()
{
  if (inProgress) {
```

```
    return;
  }
  inProgress = true;
  // Уведомление наблюдателей об уничтожении игрока:
  if (!IsAnyPlayerAlive())
  {
    for (LevelObserver o : observers)
    {
      o.LevelLost();
    }
  }
  // Уведомление наблюдателей о съедении всех гранул:
  if (RemainingPellets() == 0)
  {
    for (LevelObserver o : observers)
    {
    o.LevelWon();
    )
  }
)
```

Добавление кода для оповещения наблюдателей увеличивает размер блока кода до 21 строки (до 23, если учесть две строки с комментариями). После тестирования вновь добавленного кода, вероятно, можно подумать о реализации следующих функциональных возможностей. Но сначала нужно провести реорганизацию, чтобы привести код в соответствие с рекомендацией этой главы.

### Два метода рефакторинга для приведения кода в соответствие с рекомендацией

В этом разделе рассматриваются два метода рефакторинга кода для приведения его в соответствие с рекомендацией и уменьшения размеров блоков кода.

#### *Метод извлечения*

Одним из методов рефакторинга кода, который подходит в данном случае, является метод извлечения. В следующем фрагменте демонстрируется применение этого метода к фрагменту, приведенному выше:

```
public void Start()
{
```

```
  if (inProgress)
  {
    return;
  }
  inProgress = true;
  UpdateObservers();
}

private void UpdateObservers()
{
  // Уведомить наблюдателей об уничтожении игрока:
  if (!IsAnyPlayerAlive())
  {
    foreach (LevelObserver o in observers)
    {
      o.LevelLost();
    }
  }
  // Уведомить наблюдателей, что съедены все гранулы:
  if (RemainingPellets() == 0)
  {
    foreach (LevelObserver o in observers)
    {
      o.LevelWon();
    }
  }
}
```

Как видите, блок кода (метод Start), разросшийся ранее до 21, теперь уменьшился до 8 строк, что значительно меньше предела в 15 строк. При этом был добавлен новый блок кода (метод `UpdateObservers`). Однако он содержит 16 строк, что превышает предел в 15 строк (в ближайшее время это будет исправлено). Имеются и дополнительные преимущества. Запуск или возобновление уровня — не единственные действия, требующие уведомлять наблюдателей; им также необходимо сообщать о любых перемещениях в игре (игрока или одного из призраков). Теперь это легко осуществить, просто вызвав `UpdateObservers` из метода `Move` управления движением игрока или призраков.

Вновь созданный метод по-прежнему отвечает за две функциональные возможности, как это указано в комментариях. Можно продолжить рефакторинг кода, применив метод извлечения дважды:

```
private void UpdateObservers()
{
  UpdateObserversPlayerDied();
  UpdateObserversPelletsEaten();
}

private void UpdateObserversPlayerDied()
{
  if ((IsAnyPlayerAlive()) {
    foreach (LevelObserver o in observers)
    {
      o.LevelLdst();
    }
  }
}

private void UpdateObserversPelletsEaten ()
{
  if (RemainingPellets() == 0) {
    foreach (LevelObserver o in observers)
    {
      o.LevelWon();
    }
  }
}
```

В комментариях нет больше необходимости, их заменяют имена новых методов. Использование коротких блоков делает исходный код самоочевидным, поскольку роль комментариев берут на себя имена методов. Но все имеет свою цену: общее количество строк кода увеличилось с 16 до 25.

Написание обслуживаемого кода всегда связано с поиском компромисса между различными рекомендациями. При делении блока увеличивается общее количество строк кода. Это противоречит рекомендации об уменьшении объема базы кода (рассматривается в главе 9). Тем не менее длина и сложность блоков кода уменьшились, что упрощает их тестирование и изучение. То есть обслуживаемость улучшилась. Сохранение небольшого размера базы кода является хорошей идеей, но преимущества от работы с короткими блоками кода значительно перевешивают недостаток, связанный с увеличением общего объема кода, тем более что в этом случае оно незначительное.

О том, что написание обслуживаемого кода всегда основывается на компромиссе, свидетельствует и выбор, сделанный авторами JPacman. В исходном коде, доступном в GitHub, метод извлечения применен только один раз, в результате чего получилась версия метода `UpdateObservers` с 16 строками кода. Авторы кода JPacman решили не разделять метод `UpdateObservers` на методы `UpdateObserversPlayerDied` и `UpdateObserversPelletsEaten`.

***Замена методов объектами***

В примере выше не возникло никаких сложностей с применением метода извлечения. Причина заключается в том, что извлекаемые строки кода не нуждались в локальных переменных и не возвращали никаких значений. Но иногда требуется выделить метод, которому необходим доступ к локальным переменным. Конечно, выделенному методу всегда можно передать локальные переменные через параметры. Но это может привести к образованию длинного списка параметров, что само по себе также является проблемой (более подробно об этом в главе 5). В отношении возврата значений все обстоит еще хуже, поскольку в языке C# метод может вернуть только одно значение. В таких случаях можно применить второй метод рефакторинга, называемый *заменой методов методами объектов.*

Код JPacman содержит фрагмент, к которому можно применить этот прием рефакторинга. Рассмотрим следующий метод класса `BoardFactory`, содержащий 21 строку кода:

```
public Board CreateBoard(Square[,] grid)
{
  Debug.Assert(grid != null);

  Board board = new Board(grid);

  int width = board.Width;
  int height = board.Height;
  for (int x = 0; x < width; x++)
  {
    for (int y = 0; y < height; y++)
    {
      Square square = grid[x, y];
      foreach (Direction dir in Direction.Values)
      {
```

```
      int dirX = (width + x + dir.DeltaX) % width;
      int dirY = (height + y + dir.DeltaY) % height;
      Square neighbour = grid[dirX, dirY];
      square.Link(neighbour, dir);
    }
   }
  }

  return board;
}
```

Четыре строки в теле внутреннего цикла `for` являются кандидатами для применения метода извлечения. Но в этих четырех строках используются шесть локальных переменных, `width`, `height`, `x`, `y`, `dir`, `square`, и одна псевдопеременная, параметр `grid`. Чтобы применить метод извлечения, придется передавать в извлекаемый метод семь параметров:

```
private void SetLink(Square square, Direction dir, int x, int y,
  int width, int height, Square[,] grid)
{
  int dirX = (width + x + dir.DeltaX) % width;
  int dirY = (height + y + dir.DeltaY) % height;
  Square neighbour = grid[dirX, dirY];
  square.Link(neighbour, dir);
}
```

После внесения изменений метод `CreateBoard` будет выглядеть так:

```
public Board CreateBoard(Square[,] grid)
{
  Debug.Assert(grid != null);

  Board board = new Board(grid);

  int width = board.Width;
  int height = board.Height;
  for (int x = 0; x < width; x++)
  {
    for (int y = 0; y < height; y++)
    {
      Square square = grid[x, y];
      foreach (Direction dir in Direction.Values)
      {
        SetLink(square, dir, x, y, width, height, grid);
      }
    }
```

```
  }

  return board;
}
```

А теперь попытаемся применить вместо него прием замены методов объектами. В этом случае требуется создать новый класс, который возьмет на себя роль улучшенного метода `CreateBoard`:

```
class BoardCreator
    {
        private Square[,] grid;
        private Board board;
        private int width;
        private int height;

        internal BoardCreator(Square[,] grid)
        {
            Debug.Assert(grid != null);
            this.grid = grid;
            this.board = new Board(grid);
            this.width = board.Width;
            this.height = board.Height;
        }

        internal Board Create()
        {
            for (int x = 0; x < width; x++)
            {
                for (int y = 0; y < height; y++)
                {
                    Square square = grid[x, y];
                    foreach (Direction dir in Direction.Values)
                    {
                        SetLink(square, dir, x, y);
                    }
                }
            }
            return this.board;
        }

        private void SetLink(Square square, Direction dir, int x, int y)
        {
            int dirX = (width + x + dir.DeltaX) % width;
            int dirY = (height + y + dir.DeltaY) % height;
            Square neighbour = grid[dirX, dirY];
            square.Link(neighbour, dir);
```

```
        }
    }
```

В этом классе три локальные переменные (`board`, `width` и `height`) и параметр (`grid`) метода `createBoard` превратились в поля (`private`), доступные всем методам нового класса. Следовательно, их не нужно передавать в виде параметров. Четыре строки в теле внутреннего цикла `for` теперь помещены в новый метод `SetLink`, имеющий четыре параметра вместо семи.

Это почти все. Осталось только заменить оригинальный метод `CreateBoard`:

```
public Board CreateBoard(Square[,] grid)
{
  return new BoardCreator(grid).Create();
}
```

У нас не только получилось уместить метод в 15 строк и избежать создания методов с длинными списками параметров, а сам код стало легче читать, тестировать и повторно использовать.

## 2.3. Типичные возражения против коротких блоков кола

Хотя написание коротких блоков *выглядит* просто, разработчикам программного обеспечения часто бывает тяжело добиться этого на практике. Ниже приводятся типичные возражения против применения рассматриваемой в этой главе рекомендации.

### Возражение: увеличение количества блоков кода плохо сказывается на производительности

*«Написание коротких блоков кода означает увеличение их количества и, следовательно, более частые вызовы методов. Этого не следует делать».*

Теоретически производительность должна пострадать при увеличении количества блоков кода. Эго связано с увеличением числа вызовов методов (по сравнению с их числом при меньшем количестве более длинных блоков кода). Каждый из вызовов требует от .Net выполнения некоторой дополнительной работы. Но на практике это редко вызывает проблемы. В худшем случае речь идет о

микросекундах. Если блок кода не вызывается сотни тысяч раз в цикле, потери производительности при вызове метода незначительны. Кроме того, компилятор языка C# *очень* хорошо справляется с оптимизацией вызовов методов.

При разработке программного обеспечения корпоративного уровня в очень специфичных случаях приходится поступиться обслуживаемостью, чтобы избежать нанесения ущерба производительности. Примером могут послужить редко встречающиеся алгоритмы, вызывающие методы сотни и тысячи раз. Вероятно, это один из тех случаев в жизни программиста, когда и волки должны быть сыты, и овцы целы. Никто не утверждает, что нет никаких проблем с производительностью при разработке корпоративного программного обеспечения, но они редко бывают вызваны чрезмерным количеством вызовов методов.

Не жертвуйте обслуживаемостью в пользу производительности, если не было проведено тщательное тестирование, доказавшее, что действительно имеются проблемы с производительностью и код требует оптимизации.

### Возражение: разделение кода ухудшает читаемость

*«Код становится труднее читать при распределении его по нескольким блокам*».

Это противоречит утверждениям психологов, согласно которым люди способны удерживать в рабочей памяти около семи элементов, поэтому при чтении кода, содержащего больше семи строк, они не способны удерживать их в голове. Исключение здесь, наверное, можно сделать для первоначального автора фрагмента исходного кода в момент его написания (но не через неделю).

Пишите код, который легко читается и понятен вашим последователям (в том числе и вам, по прошествии определенного времени).

### Рекомендация препятствует надлежащему форматированию кода

*«Следование вашей рекомендации ведет к неправильному форматированию исходного кода».*

Не пытайтесь при следовании рекомендации решать проблему за счет форматирования. Речь идет о размещении нескольких операторов или нескольких фигурных скобок в одной строке. Это затрудняет чтение кода, снижая уровень его обслуживаемости. Не поддавайтесь искушению поступать таким образом.

Обсудим главную цель рекомендации. Мы просто не можем не ограничивать длину блоков кода. Это было бы сродни отказу от ограничения скорости при движении транспорта, по причине того, что такое ограничение не позволяет прибыть в пункт назначения вовремя. Мы вполне можем приехать вовремя, соблюдая при этом ограничения скорости, просто нужно выйти из дома немного раньше. Это правило применимо и к блокам кода. Наш опыт показывает, что 15 правильно отформатированных строк кода вполне достаточно для написания полезных блоков кода.

В качестве доказательства в табл. 2.1 приводятся некоторые данные, полученные для обычной системы на Java 2 Enterprise Edition, состоящей из исходных файлов па языке Java, а также нескольких XSD- и XSLT-файлов. В настоящее время эта система используется в компании SIG для создания отчетов. Ее Java-часть насчитывает около 28 000 строк кода (система среднего размера). Из этих 28 000 строк чуть более 17 000 размещены в блоках. Всего имеется чуть больше 3000 блоков.

Из 3220 блоков кода в этой системе 3071 (95,4%) содержит не более 15 строк, в то время как 149 блоков (4,6%) содержат больше строк. Это демонстрирует возможность писать короткие блоки кода, по крайней мере, для подавляющего их большинства.

***Таблица 2.1 ❖ Распределение блоков кода по размерам в реальной корпоративной системе***

| Строк в блоках кода | Число блоков кода (в абсолютном выражении) | Число блоков кода (в %) | Число строк кода (в абсолютном выражении) | Число строк кода (в %) |
|---|---|---|---|---|
| 1-15 | 3071 | 95,4% | 14 032 | 81,3% |
| 16 и более | 149 | 4,6% | 3221 | 18,7% |
| **Итого** | 3220 | 100% | 17 253 | 100% |

Примите соглашение о форматировании кода в команде. Создавайте короткие блоки, соответствующие этому соглашению.

### Этот блок кода невозможно разделить

*«Мой блок в самом деле невозможно разделить».*

Иногда разделить метод действительно сложно. Возьмем, к примеру, правильно отформатированный оператор `switch`. Для каждого варианта `case` оператора `switch` нужны, по крайней мере, одна строка для выполнения полезного действия п строка для оператора `break`. То есть в 15 строк никак нельзя уместить более четырех вариантов `case` и нельзя разделить оператор `switch`. В главе 3 приводится несколько специальных рекомендаций, касающихся исключительно операторов `switch`.

Это правда, что некоторые операторы просто невозможно разделить. Типичным примером в корпоративном программном обеспечении являются SQL-запросы. Рассмотрим следующий фрагмент (взятый из реальной системы, анализ которой был проведен авторами книги):

```
public static void PrintDepartmentEmployees(string department)
{
  Query q = new Query();
  foreach (Employee e in q.AddColumn("FamilyName")
    .AddColumn("Initials")
    .AddColumn("GivenName")
    .AddColumn("AddressLine1")
    .AddColumn("ZIPcode")
    .AddColumn("City")
    .AddTable("EMPLOYEES")
    .AddWhere($"EmployeeDep='{department}'")
    .Execute())
  {
    Console.WriteLine($@"<div name='addressDiv'>
      {e.FamilyName}, {e.Initials}<br />" +
      "{e.AddressLine1}<br />{e.ZipCode}{e.City}</div>");
  }
}
```

Этот фрагмент содержит 16 строк кода. Но в нем всего три оператора. Второй оператор содержит выражение, занимающее девять строк. Его не получится выделить ни методом извлечения, ни методом замены методов объектами, по крайней мере не напрямую.

Однако выражение в девять строк, начинающееся с `q.AddColumn("FamilyName")`, можно выделить в новый метод. Но прежде чем сделать это (и увидеть, что вновь созданный метод в дальнейшем превысит 15 строк, когда запрос еще более усложнится), имеет смысл переосмыслить архитектуру. Разумно ли создавать SQL-запрос по частям, как в этом фрагменте? Нужна ли здесь HTML-разметка? Возможно, здесь больше подойдет решение, основанное на шаблонах ASP или Razor.

То есть если вы столкнулись с блоком, который, кажется, невозможно разделить, не следует оставлять его в прежнем виде и переходить к другой программной задаче, возможно, имеет смысл обсудить этот вопрос с членами или руководителем команды.

Когда модернизация кода возможна, но выглядит бессмысленной, попробуйте переосмыслить архитектуру системы.

### Разделение блоков кода не дает заметных преимуществ

*«Разделение кода на методы `DoSomethingOne`, `DoSomethingTwo`, `DoSomethingThree` не дает никаких преимуществ по сравнению с тем же кодом в одном длинном методе `DoSomething`».*

На самом деле они появятся, если выбрать для методов более подходящие имена, чем `DoSomethingOne`, `DoSomethingTwo` и т. д. В каждом отдельном коротком блоке легче разобраться, чем в длинном методе `DoSomething`. И, что еще более важно, можно даже не вдаваться во все детали, если имена каждого из методов будут указывать на назначение этих блоков. Кроме того, длинные методы `DoSomething` обычно сочетают в себе несколько задач. Это означает, что повторно использовать метод `DoSomething` можно будет только тогда, когда потребуется точно такая же их комбинация. Скорее всего, потребуется использовать один из методов `DoSomethingOne`, `DoSomethingTwo` и т. д.

Помещайте код в короткие блоки кода (не более 15 строк кода) с тщательно подобранными именами, описывающими их назначение.

## 2.4. Дополнительные сведения

Дополнительные методы рефакторинга приводятся в главах 3, 4 и 5. Тестированию методов посвящена глава 10.

### Как в SIG устанавливается рейтинг по размерам блоков

Размер (длина) блоков кода (методов и конструкторов в языке C#) является одним из восьми критериев SIG/TÜViT-оценки свойств системы для достижения уверенной обслуживаемости продукта. При оценке размеров каждый блок кода помещается в одну из четырех категорий риска, в зависимости от количества строк. В табл. 2.2 перечислены четыре категории риска, используемые в версии 2015 критериев оценки SIG/TÜViT.

Критерии (количество строк) в табл. 2.2 носят конъюнктивный характер: система должна соответствовать всем четырем из них. Например, если 6,9% всех строк кода находится в методах с длиной более 60 строк, системе все еще можно присвоить 4 звезды. Но при этом в методах с длиной 30 до 60 строк может располагаться не более 22,3% — 6,9% = 15,4% от всех строк кода. С другой стороны, если система не содержит методов длиннее 60 строк, тогда 22,3% всех строк кода может находиться в методах с длиной от 30 до 60 строк.

***Таблица 2.2 ❖ Минимальные границы для системы с рейтингом в 4 звезды по размерам блоков кода (версия 2015 критериев оценки SIG/TÜViT)***

| Строк кода в методах... | Разрешенный процент для размера блоков в системе с 4 звездами |
|---|---|
| ...более 60 | 6,9% |
| ...более 30 | 22,3% |
| ...более 15 | 43,7% |
| 15 и менее | 56,3% |

На рис. 2.2 изображены примеры профилей качества трех систем:

- слева — диаграмма для системы с открытым исходным кодом, в данном случае Jenkins;
- в центре — диаграмма для анонимной системы, используемой SIG в качестве эталона системы с рейтингом в 4 звезды по размерам блоков кода;
- справа — диаграмма, отображающая граничные точки характеристик для достижения качества, оцениваемого в 4 звезды.


**Рис. 2.2** ❖ Три профиля качества по размерам блоков кода

# Глава 3

# Пишите простые блоки кода

Любая задача делится на несколько маленьких задач.
*Мартин Фаулер (Marlin Fowler)*

Рекомендация:
• **блок не должен содержать больше 4 точек ветвления;**
• для этого **делите сложные блоки на несколько более простых** и вообще избегайте сложных блоков кода;
• это улучшает обслуживаемость, поскольку уменьшение количества точек ветвления **упрощает модификацию и тестирование кода.**

Сложность представляет собой часто оспариваемую характеристику качества. Код, кажущийся сложным постороннему или начинающему разработчику, может выглядеть простым для разработчика, хорошо знакомого с ним. Сложность во многом зависит от того, кто просматривает код. Существует, однако, граница, за которой код представляется настолько сложным, что его изменение становится чрезвычайно рискованной и очень трудоемкой задачей, не говоря уже о дальнейшем тестировании изменений. Чтобы сохранить код обслуживаемым, следует установить предел его сложности. Еще одна причина оценки сложности кода заключается в необходимости определения минимального количества тестов, достаточных для уверенности в предсказуемой работе системы. Перед тем как указать предел сложности кода, следует научиться оценивать его сложность.

Обычным способом объективной оценки сложности кода является подсчет количества всевозможных путей выполнения фрагмента кода. То есть чем больше таких маршрутов можно выделить, тем сложнее фрагмент кода. Число таких путей можно определить,

подсчитав число *точек ветвления.* Точка ветвления — это оператор, где может быть выбрано несколько направлений продолжения выполнения, в зависимости от конкретного состояния. Примерами точек ветвления в языке C# являются операторы `if` и `switch` (полный их список приводится ниже). Точки ветвления можно подсчитать во всей базе кода, классах, пакетах или блоках кода. Число точек ветвления в блоке кода равно минимальному числу путей, необходимых для охвата всех ветвей, созданных этими точками ветвления. Это называется *охватом ветвей.* Однако при учете всех путей, от первой строки до конечного оператора, может возникать комбинаторный эффект. Причина в том, что *может* иметь значение порядок прохождения ветвей. Все возможные комбинации ветвей называются *путями выполнения,* то есть они определяют максимальное количество путей через блок.

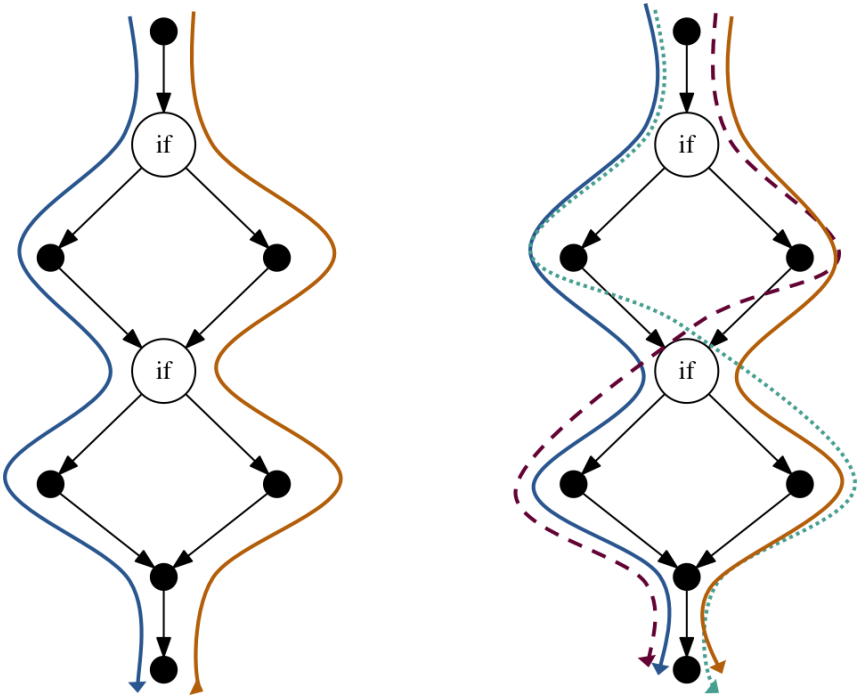


**Рис. 3. 1** ❖ Охват ветвей и охват путей выполнения

Рассмотрим блок, содержащий два последовательных оператора `if`. Рисунок 3.1 отображает управление потоком через блок и демонстрирует разницу между охватом ветвей и охватом путей выполнения.

Предположим, что в точке слева от первого оператора `if` вносятся изменения в базу данных, а в точке справа от второго оператора `if` выполняется чтение данных из этой базы. Такие побочные эффекты требуют учитывать пути в форме «зигзагов» (показаны пунктиром на рис. 3.1).

Таким образом, число точек ветвления определяет число путей, охватывающих все ветви, созданные точками ветвления. Это минимальное число путей, и оно может быть равно нулю (для блоков без точек ветвления). Число путей выполнения является максимальным и может стать очень большим из-за комбинаторного эффекта. Какое из них следует выбрать?

Ответ прост: определите число точек ветвления и прибавьте к нему единицу. Это называется *цикломатической сложностью,* или *сложностью по Маккейбу* (McCabe). Следовательно, рекомендация «блок не должен содержать больше 4 точек ветвления» равноценна рекомендации: «предельная сложность кода по Маккейбу не должна превышать 5». Это число определяет минимальное количество тестов, необходимое для охвата блока, в котором каждый путь имеет часть, не входящую в другие пути. Чтобы проще было понять идею цикломатической сложности (сложности по Маккейбу), рассмотрим блок кода без точек ветвления. Согласно определению, его цикломатическая сложность равна единице (число точек ветвления плюс единица). Это понятно и очевидно: блок без точек ветвления имеет один путь выполнения, и для его проверки потребуется, по крайней мере, один тест.

Для полноты картины отметим, что для блоков кода с одной точкой выхода цикломатическая сложность, или сложность по Маккейбу, равна числу точек ветвления плюс единица. Но как обстоит дело с более сложными блоками, содержащими несколько точек выхода? Пусть вас это не беспокоит, важно только ограничение числа точек ветвления.

Минимальное число тестов, охватывающих все независимые пути выполнения, равно количеству точек ветвления плюс единица.

А теперь рассмотрим следующий пример. Для данной страны метод `getFlagColors` возвращает правильные цвета ее флага:

```
public IList<Color> GetFlagColors(Nationality nationality)
{
    List<Color> result;
    switch (nationality)
    {
        case Nationality.DUTCH:
            result = new List<Color> { Color.Red, Color.White, Color.Blue };
            break;
        case Nationality.GERMAN:
            result = new List<Color> { Color.Black, Color.Red, Color.Yellow };
            break;
        case Nationality.BELGIAN:
            result = new List<Color> { Color.Black, Color.Yellow, Color.Red };
            break;
        case Nationality.FRENCH:
            result = new List<Color> { Color.Blue, Color.White, Color.Red };
            break;
        case Nationality.ITALIAN:
            result = new List<Color> { Color.Green, Color.White, Color.Red };
            break;
        case Nationality.UNCLASSIFIED:
        default:
            result = new List<Color> { Color.Gray };
            break;
    }
    return result;
}
```

Оператор `switch` в теле метода должен обрабатывать все перечисленные в нем страны и возвращать правильные цвета их флагов. При перечислении пяти стран, а также нераспознанной страны по умолчанию число изолированных путей (контролируемых потоков через ветви) равно шести.

На первый взгляд метод `GetFlagColors` может показаться вполне безобидным. Действительно, метод легко читается и выполняется ожидаемым образом. Тем не менее для тестирования этого метода потребуются шесть уникальных вариантов тестов (по одному для каждой страны и еще один для варианта по умолчанию). Написание автоматизированных тестов для метода `GetFlagColors` может выгля-

деть излишним, но предположим, что разработчику потребовалось срочно добавить в метод флаг Люксембурга (очень похожего на флаг Нидерландов):

```
...
        case Nationality.DUTCH:
            result = new List<Color> { Color.Red, Color.White, Color.Blue };
        case Nationality.LUXEMBOURGER:
            result = new List<Color> { Color.Red, Color.White, Color.LightBlue };
            break;
        case Nationality.GERMAN:
....
```

Впопыхах разработчик скопировал вызов конструктора для голландского флага и изменил в нем последний аргумент, указав нужный цвет. К сожалению, разработчик позабыл об операторе `break`, и теперь жители Нидерландов увидят флаг Люксембурга на своей странице!

Этот пример выглядит надуманным, но из практики нам хорошо знакомы подобные случаи, вызванные сложностью кода. Такие простые ошибки являются причиной многих «тривиальных» сбоев, которые можно легко предотвратить.

Чтобы понять, почему сложный код вызывает проблемы при обслуживании, важно понимать, что код, изначально вполне прямолинейный, как правило, усложняется с течением времени. Рассмотрим следующий фрагмент, позаимствованный из реализации сервера Jenkins. Этот фрагмент содержит 20 ветвей. Представьте себе процесс изменения и тестирования этого метода:

```
/**
* Получить пользователя по его идентификатору или создать
* нового, если затребовано.
* @return
*      Существующий или вновь созданный пользователь.
*      Может быть {@code null} , если искомый пользователь не существует
*      и {@code create} имеет значение false.
*/
private static User getOrCreate(string id, string fullName, bool create)
{
    string idkey = idStrategy().keyFor(id);

    byNameLock.readLock().doLock();
    User u;
    try
```

```
{
    u = byName.get(idkey);
}
finally
{
    byNameLock.readLock().unlock();
}
FileInfo configFile = getConfigFileFor(id);
if (!configFile.Exists && !Directory.Exists(configFile.Directory.FullName))
{
    // проверить, не используется ли устаревшая структура
    // с информацией о пользователе, и обновить, если это
    // можно сделать безопасно.
    FileInfo[] legacy = getLegacyConfigFilesFor(id);
    if (legacy != null && legacy.Length > 0)
    {
        foreach (FileInfo legacyUserDir in legacy)
        {
            XmlFile legacyXml = new XmlFile(XmlFile.XSTREAM,
                                    new FileInfo(Path.Combine(
                                    legacyUserDir.FullName, "config.xml")));
            try
            {
                object o = legacyXml.read();
                if (o is User)
                {
                    if (idStrategy().equals(id, legacyUserDir.Name)
                        && !idStrategy()
                        .filenameOf(legacyUserDir.Name)
                        .Equals(legacyUserDir.Name))
                    {
                        try
                        {
                            File.Move(legacyUserDir.FullName,
                                configFile.Directory.FullName);
                        }
                        catch (IOException)
                        {
                            LOGGER.log(Level.WARNING,
                                "Failed to migrate user record from {0} " +
                                "to {1}", new Object[] {legacyUserDir,
                                    configFile.Directory.FullName
                                    });
                        }
                        break;
                    }
                }
```

```
                else
                {
                    LOGGER.log(Level.FINE,
                        "Unexpected object loaded from {0}: {1}",
                        new object[] { legacyUserDir, o });
                }
            }
            catch (IOException e)
            {
                LOGGER.log(Level.FINE,
                    string.Format(
                        "Exception trying to load user from {0}: {1}",
                        new Object[] { legacyUserDir, e.Message }),
                    e);
            }
        }
    }
}
if (u == null && (create || configFile.Exists))
{
    User tmp = new User(id, fullName);
    User prev;
    byNameLock.readLock().doLock();
    try
    {
        prev = byName.putIfAbsent(idkey, u = tmp);
    }
    finally
    {
        byNameLock.readLock().unlock();
    }
    if (prev != null)
    {
        u = prev; // если значение уже есть в словаре, использовать его
        if (LOGGER.isLoggable(Level.FINE)
            && !fullName.Equals(prev.getFullName()))
        {
            LOGGER.log(Level.FINE,
                "mismatch on fullName (`" + fullName + "' vs. `"
                + prev.getFullName() + "') for `" + id + "'",
                new Exception());
        }
    }
    else if (!id.Equals(fullName) && !configFile.Exists)
    {
        // JENKINS-16332: так как иногда fullName может оказаться
        // невозможно получить из id, а различный код может
```

```
            // хранить только id, мы должны сохранить fullName
            try
            {
                u.save();
            }
            catch (IOException x)
            {
                LOGGER.log(Level.WARNING, null, x);
            }
        }
    }
    return u;
}
```

## 3.1. Мотивация

Основываясь на примерах, приведенных в предыдущем разделе, можно сформулировать две причины, определяющие важность написания простых блоков кода:

- в простом блоке проще разобраться, чем в сложном, и поэтому его проще изменять;
- простые блоки проще тестировать.

### Простые блоки проще изменять

Блоки кода с высокой сложностью трудно понять, что приводит к проблемам при их изменении. Первый пример из первого раздела не слишком сложен, но это пока в нем не обрабатывается, скажем, 15 и более стран. Второй пример реализует множество вариантов поиска и создания пользователей. Разобраться в нем будет довольно сложно. Длительное время, необходимое для достижения понимания кода, затрудняет внесение изменений.

### Простые блоки проще тестировать

Веской причиной поддержания простоты является упрощение их тестирования. Если в блоке имеются шесть ветвей выполнения, понадобится минимум шесть тестов для их охвата. Вспомним метод `GetFlagColors`, для которого требуются шесть тестов, по одному для каждой из стран и еще один для варианта по умолчанию, чтобы предотвратить тривиальные ошибки.

## 3.2. Как применять рекомендацию

Как уже упоминалось в начале главы, блок кода не должен содержать больше четырех точек ветвления. Точками ветвления в C# являются следующие операторы и инструкции:

```
if;
case;
?, ??;
&&, ||;
while;
for, foreach;
catch.
```

Итак, как ограничить количество точек ветвления? Этот вопрос главным образом относится к правильному определению причин высокой сложности кода. Во многих случаях сложный блок кода состоит из нескольких блоков, а его сложность вычисляется как сумма сложностей его частей. В других случаях сложность возникает как результат вложенности операторов `if-then-else`, что делает код еще более трудно читаемым с увеличением уровня вложенности. Еще одна причина: наличие длинной цепочки операторов `if-then-else` или длинного оператора `switch`, как, например, в методе `GetFlagColors`.

Каждый из этих случаев отличается собственными проблемами и требует отдельного решения. Первый случай, когда блок кода состоит из нескольких блоков, практически независимых друг от друга, является хорошим кандидатом на оптимизацию с помощью *метода извлечения*. Этот способ аналогичен методу, описанному в главе 2. Но как быть, когда имеешь дело с другими случаями высокой сложности?

### Цепочки условий

В цепочке операторов `if-then-else` решение принимается в каждом из условных операторов. Наиболее простым случаем является тот, в котором условия являются взаимоисключающими, то есть каждое из них применимо только для конкретной ситуации. Это особенно характерно для оператора `switch`, как в методе `GetFlagColors`.

Существует много способов упрощения этого вида сложности, и выбор решения во многом зависит от конкретной ситуации. Приведем два альтернативных способа уменьшения сложности для метода `GetFlagColors`. Первый основан на добавлении структуры данных `Map`, отображающей страны в объекты `Flag`. Он позволит уменьшить сложность метода `GetFlagColors` с 7 до 2 по Маккейбу.

```
private static Dictionary<Nationality, IList<Color>> FLAGS =
    new Dictionary<Nationality, IList<Color>>();

static FlagFactoryWithMap()
{
    FLAGS[Nationality.DUTCH] = new List<Color>{ Color.Red, Color.White,
        Color.Blue };
    FLAGS[Nationality.GERMAN] = new List<Color>{ Color.Black, Color.Red,
        Color.Yellow };
    FLAGS[Nationality.BELGIAN] = new List<Color>{ Color.Black, Color.Yellow,
        Color.Red };
    FLAGS[Nationality.FRENCH] = new List<Color>{ Color.Blue, Color.White,
        Color.Red };
    FLAGS[Nationality.ITALIAN] = new List<Color>{ Color.Green, Color.White,
        Color.Red };
}

public IList<Color> GetFlagColors(Nationality nationality)
{
    IList<Color> colors = FLAGS[nationality];
    return colors ?? new List<Color> { Color.Gray };
}
```

Второй, более продвинутый способ уменьшения сложности, основывается на применении шаблона рефакторинга, который отделяет функциональность различных флагов с помощью введения разных классов флагов. Это можно сделать с помощью модели *замены условий полиморфизмом*, то есть каждый флаг получит собственный класс, реализующий общий интерфейс. Полиморфная модель поведения в C# обеспечивает такую функциональность во время выполнения. Начнем изменения с определения общего интерфейса `IFlag`:

```
public interface IFlag
{
    IList<Color> Colors { get; }
}
```

и класса флагов для различных стран, таких как Нидерланды:

```
public class DutchFlag : IFlag
{
    public IList<Color> Colors
    {
        get
        {
            return new List<Color> { Color.Red, Color.White, Color.Blue };
        }
    }
}
```

и Италия:

```
public class ItalianFlag : IFlag
{
    public IList<Color> Colors
    {
        get
        {
            return new List<Color> { Color.Green, Color.White, Color.Red };
        }
    }
}
```

Теперь метод `GetFlagColors` сократился в размерах, и уменьшилась возможность появления в нем ошибок:

```
private static readonly Dictionary<Nationality, IFlag> FLAGS =
    new Dictionary<Nationality, IFlag>();

static FlagFactory()
{
    FLAGS[Nationality.DUTCH] = new DutchFlag();
    FLAGS[Nationality.GERMAN] = new GermanFlag();
    FLAGS[Nationality.BELGIAN] = new BelgianFlag();
    FLAGS[Nationality.FRENCH] = new FrenchFlag();
    FLAGS[Nationality.ITALIAN] = new ItalianFlag();
}

public IList<Color> GetFlagColors(Nationality nationality)
{
    IFlag flag = FLAGS[nationality];
    flag = flag ?? new DefaultFlag();
    return flag.Colors;
}
```

Такая модернизация кода предлагает наиболее гибкую реализацию. Например, она обеспечивает возможность расширения иерархии классов флагов новыми типами флагов и их изолированного тестирования. Недостаток такого подхода заключается в необходимости добавлять определения классов. Разработчик должен сделать выбор между расширяемостью и лаконичностью.

### Вложенность

Предположим, что блок содержит глубоко вложенные условия, как это показано в следующем примере. Получив корневой узел бинарного дерева и целое число, метод `CalculateDepth` определяет, содержит ли дерево это число. Метод либо возвращает уровень глубины найденного числа в дереве, либо возбуждает исключение `TreeException`:

```
public static int CalculateDepth(BinaryTreeNode<int> t, int n)
{
    int depth = 0;
    if (t.Value == n)
    {
        return depth;
    }
    else
    {
        if (n < t.Value)
        {
            BinaryTreeNode<int> left = t.Left;
            if (left == null)
            {
                throw new TreeException("Value not found in tree!");
            }
            else
            {
                return 1 + CalculateDepth(left, n);
            }
        }
        else
        {
            BinaryTreeNode<int> right = t.Right;
            if (right == null)
            {
                throw new TreeException("Value not found in tree!");
            }
```

```
            else
            {
                return 1 + CalculateDepth(right, n);
            }
        }
    }
}
```

Для улучшения читаемости кода можно избавиться от вложенного условия, идентифицировав конкретные случаи и вставив операторы `return`. В терминах рефакторинга это называется *заменой вложенных условий граничными операторами.* В результате получится следующий метод:

```
public static int CalculateDepth(BinaryTreeNode<int> t, int n)
{
    int depth = 0;
    if (t.Value == n)
    {
        return depth;
    }
    if ((n < t.Value) && (t.Left != null))
    {
        return 1 + CalculateDepth(t.Left, n);
    }
    if ((n > t.Value) && (t.Right != null))
    {
        return 1 + CalculateDepth(t.Right, n);
    }
    throw new TreeException("Value not found in tree!");
}
```

Блок кода получился более понятным, но его сложность не уменьшилась. Чтобы уменьшить сложность, следует выделить вложенные условия в отдельные методы, например:

```
public static int CalculateDepth(BinaryTreeNode<int> t, int n)
{
    int depth = 0;
    if (t.Value == n)
    {
        return depth;
    }
    else
    {
        return TraverseByValue(t, n);
    }
```

```
}

private static int TraverseByValue(BinaryTreeNode<int> t, int n)
{
    BinaryTreeNode<int> childNode = GetChildNode(t, n);
    if (childNode == null)
    {
        throw new TreeException("Value not found in tree!");
    }
    else
    {
        return 1 + CalculateDepth(childNode, n);
    }
}

private static BinaryTreeNode<int> GetChildNode(
    BinaryTreeNode<int> t, int n)
{
    if (n < t.Value)
    {
        return t.Left;
    }
    else
    {
        return t.Right;
    }
}
```

Сложность кода *уменьшилась*. Мы получили два преимущества: методы стали более понятными и за счет изоляция проще в тестировании, поскольку появилась возможность писать модульные тесты для проверки функций по отдельности.

## 3.3. Типичные возражения против создания простых блоков кода

Естественно, по мере увеличения блоков кода они усложняются. Можно утверждать, что увеличение сложности неизбежно или что уменьшение сложности блоков кода не поможет улучшить обслуживаемость системы. Обсудим эти возражения.

### Возражение: высокая сложность неизбежна

*«Наша предметная область настолько сложна, что высокая сложность кода неизбежна».*

При работе в сложной области, такой как оптимизация логистических задач, визуализация в реальном времени или другой, требующей реализации сложной прикладной логики, естественно считать, что сложность предметной области влияет на сложность реализации, и этого нельзя избежать.

Мы выступаем против такого утверждения. Сложность предметной области не означает сложности технической реализации. На разработчика ложится ответственность за такое упрощение задачи, которое позволит написать простой код. Даже если система в целом выполняет сложные функции, это не означает, что низкоуровневые блоки кода тоже должны быть сложными. В тех случаях, когда система должна обрабатывать множество условий и исключений (например, конкретные требования законодательства), одним из решений может стать реализация простого процесса и модели исключений.

Действительно, чем сложнее предметная область, тем больше усилий должен приложить разработчик, чтобы получить простое техническое решение. Но это возможно! Мы видели много хорошо обслуживаемых систем, решающих сложные бизнес-задачи, и уверены, что решить сложную задачу и обеспечить ее обслуживаемость можно только через простой код.

### Возражение: разделение методов не уменьшает сложности

*«Замена одного метода, с уровнем сложности 15 по Маккейбу, тремя, с уровнем сложности 5, означает, что общая сложность по Маккейбу по-прежнему равна 15 (и остаются те же 15 ветвей выполнения). То есть это ничего не меняет».*

Конечно, разделение методов не уменьшает общую сложность системы по Маккейбу. Но с точки зрения удобства обслуживания это дает дополнительные преимущества: простоту тестирования и понимания кода. И, как уже упоминалось, дополнительные модульные тесты упрощают выявление причин ошибок.

Размещайте код в простых блоках (содержащих не более четырех точек ветвления) с тщательно подобранными именами, описывающими их назначение и особенности.

## 3.4. Дополнительные сведения

Приемы рефакторинга для разделения блоков кода были описаны в главе 2.

### Как в компании SIG устанавливается рейтинг сложности

Сложность (по Маккейбу) блоков кода (методов и конструкторов в языке C#) является одним из восьми критериев SIG/TÜViT оценки системных свойств для достижения уверенной обслуживаемости продукта. Чтобы определить рейтинг сложности, все блоки кода в системе распределяются по четырем категориям риска. В табл. 3.1 перечислены четыре категории рисков в версии 2015 критериев оценки SIG/TÜViT.

Критерии (строки) в табл. 3.1 имеют конъюнктивный характер: база кода должна соответствовать всем четырем из них. Например, если 1,5% всех строк кода находится в методах со сложностью по Маккейбу более 25, такой системе еще можно присвоить 4 звезды. Но в этом случае не более 10,0% — 1,5% = 8,5% всех строк может находиться в методах со сложностью по Маккейбу от 10 до 25.

***Таблица 3.1 ❖ Минимальные границы для системы с рейтингом в 4 звезды по сложности блоков кода (версия 2015 критериев оценки SIG/TÜViT)***

| Сложность блоков по Маккейбу... | Допустимый процент блоков в 4-звездной системе |
|---|---|
| ... выше 25 | 1,5% |
| ... выше 10 | 10,0% |
| ... выше 5 | 25,2% |
| ... не более 5 | 74,8% |

На рис. 3.2 изображены примеры профилей качества трех систем:

- слева — диаграмма для системы с открытым исходным кодом, в данном случае Jenkins;
- в центре — диаграмма для анонимной системы, используемой компанией SIG в качестве эталона 4-звездной системы;
- справа — диаграмма, отображающая граничные точки характеристик для достижения качества, оцениваемого в 4 звезды.


**Рис. 3.2** ❖ Три профиля качества по сложности блоков кода

# Глава 4

# Не повторяйте один и тот же код

Первое место среди недостатков
занимает дублирование кода.

*Кент Бек (Kent Beck)*
*и Мартин Фаулер (Martin Fowler),*
*глава «Bad Smells in Code»*[1]

Рекомендация:
• **не дублируйте код**;
• пишите **многократно используемый, обобщенный код и/или вызывайте существующие методы**;
• это улучшит обслуживаемость, поскольку **при дублировании кода исправлять ошибки придется в нескольких местах**, это неэффективно и чревато новыми ошибками.

Копирование существующего кода выглядит заманчивым решением. Зачем писать что-то заново, когда это уже существует? Но дело в том, что копирование кода приводит к дублированию, а дублирование вызывает проблемы. Согласно цитате выше, существует даже мнение, что дублирование кода является самой главной проблемой, снижающей качество программного обеспечения.

Рассмотрим систему управления банковскими счетами. В этой системе денежные переводы между счетами представляют объекты класса `Transfer` (здесь не показан). Банк применяет проверку счетов, предоставляемую классом `CheckingAccount`:

```
public class CheckingAccount
{
    private int transferLimit = 100;
```

[1] *Фаулер М., Век К., Брант Дж., Робертс Д., Апдайк У.* Рефакторинг: улучшение существующего кода. СПб.: Символ-Плюс, 2009. ISBN: 5-93286-045-6. — *Прим. ред.*

```
    public Transfer MakeTransfer(String counterAccount, Money amount)
    {
        // 1. Проверка превышения лимита:
        if (amount.GreaterThan(this.transferLimit))
        {
            throw new BusinessException("Limit exceeded!");
        }
        // 2. Проверка 9-значного номера банковского счета на кратность 11:
        int sum = 0;
        for (int i = 0; i < counterAccount.Length; i++)
        {
            sum = sum + (9 - i) * (int)Char.GetNumericValue(
                counterAccount[i]);
        }
        if (sum % 11 == 0)
        {
            // 3. Поиск контрсчета и создание объекта перевода:
            CheckingAccount acct = Accounts.FindAcctByNumber(counterAccount);
            Transfer result = new Transfer(this, acct, amount);
            return result;
        }
        else
        {
            throw new BusinessException("Invalid account number!");
        }
    }
}
```

Получив номер счета, куда переводятся деньги (в виде строки), метод `MakeTransfer` создает объект `Transfer`. Он сначала проверяет, не превышает ли сумма перевода установленного лимита. В этом примере лимит указывается явно. Затем проверяет, соответствует ли номер счета получателя контрольной сумме (порядок проверки контрольной суммы описан во врезке «Проверка контрольной суммы номеров банковских счетов на кратность 11» в главе 2). После этого извлекается объект, представляющий счет получателя, и создается и возвращается объект `Transfer`.

Теперь предположим, что банк ввел новый вид счетов — сберегательные счета. Для сберегательных счетов сумма перевода не ограничивается, но деньги с них можно перевести только на один конкретный (фиксированный) расчетный счет. Идея в том, что владелец счета только один раз выбирает конкретный расчетный счет для присоединения его к сберегательному счету.

Необходимо определить класс, представляющий новый вид счета. Предположим, что для этого мы просто скопировали п переименовали существующий класс, немного подправив его реализацию:

```
public class SavingsAccount
{
    public CheckingAccount RegisteredCounterAccount { get; set; }

    public Transfer makeTransfer(string counterAccount, Money amount)
    {
        // 1. Проварка 9-значного номера банковского счета на кратность 11:
        int sum = 0; ❶
        for (int i = 0; i < counterAccount.Length; i++)
        {
            sum = sum + (9 - i) * (int)Char.GetNumericValue(
                counterAccount[i]);
        }
        if (sum % 11 == 0)
        {
            // 2. Поиск контрсчета и создание объекта перевода:
            CheckingAccount acct = Accounts.FindAcctByNumber(counterAccount);
            Transfer result = new Transfer(this, acct, amount); ❷
            // 3. Проверка контрсчета на соответствие зарегистрированному:
            if (result.CounterAccount.Equals(this.RegisteredCounterAccount))
            {
                return result;
            }
            else
            {
                throw new BusinessException("Counter-account not registered!");
            }
        }
        else
        {
            throw new BusinessException("Invalid account number!!");
        }
    }
}
```

❶ Начало повторяющегося кода.
❷ Конец повторяющегося кода.

Оба класса находятся в одной базе кода. Скопировав существующий класс, мы добавили дублирующий код. В настоящий момент имеются два совершенно одинаковых фрагмента кода (по

восемь строк). Такие фрагменты называются *клонами,* или *дубликатами.*

Теперь предположим, что в реализации проверки на кратность 11 (в цикле `for`, выполняющем перебор символов в строке `CounterAccount`) обнаружилась ошибка. Эту ошибку придется исправить в обоих дубликатах. Такая дополнительная работа снижает эффективность обслуживания. Кроме того, если исправление будет выполнено только в одном из дубликатов, поскольку о другом просто забыли, ошибка будет исправлена лишь наполовину.

Не поддавайтесь искушению выигрыша времени от копирования и вставки кода. При каждой последующей корректировке вам потребуется просматривать все дубликаты.

Программирование должно основываться на поиске *общих* решений *конкретных задач.* Более общим решением является повторное использование (путем вызова) существующего универсального метода или придание существующему методу большей универсальности.

### Виды дублирования

Под *дубликатами,* или *клонами,* понимаются идентичные фрагменты кода длиной не менее шести строк. В это число не входят пустые строки и комментарии в соответствии с определением «строк кода» (в главе 1). То есть чтобы строки рассматривались как дубликаты, они должны быть *совершенно* одинаковыми. Такие клоны называются *клонами первого вида.* Не имеет значения, где находятся дубликаты. Два клона могут располагаться в одном методе, в различных методах одного класса или в различных методах разных классов в одной базе кода. Клоны могут пересекать границы метода. Например, если в базе кода дважды встречается приведенный ниже фрагмент, он считается одним клоном из шести строк кода, а не двумя клонами из трех строк:

```
public void SetGivenName(string givenName)
{
    this.givenName = givenName;
}

public void SetFamilyName(string familyName)
```

```
{
    this.familyName = familyName;
}
```

Следующие два метода не считаются дубликатами, даже при том, что отличаются только литеральными значениями и именами идентификаторов:

```
public void SetPageWidthInInches(float newWidth)
{
    float cmPerInch = 2.54f;
    this.pageWidthInCm = newWidth * cmPerInch;
    // И еще несколько строк.
}

public void SetPageWidthInPoints(float newWidth)
{
    float cmPerPoint = 0.0352777f;
    this.pageWidthInCm = newWidth * cmPerPoint;
    // И еще несколько строк (таких же, как в SetPageWidthlnlnches).
}
```

Два фрагмента кода, которые синтаксически одинаковы (в отличие от текстуальной одинаковости), называются *клонами второго вида.* Клоны второго вида отличаются друг от друга только пробелами, комментариями, именами идентификаторов и значениями литералов. Любой клон первого вида одновременно является клоном второго вида, но некоторые клоны второго вида не являются клонами первого вида. Методы `SetPageWidthlnlnches` и `SetPageWidthlnPoints` являются клонами второго вида, но не клонами первого вида.

Рекомендация, представленная в этой главе, касается только клонов первого вида, и на это есть две причины:

- обслуживаемость улучшается в основном при удалении клонов первого вида;
- клоны первого вида легче обнаружить и распознать (как человеку, так и компьютеру, поскольку выявление клонов второго вида потребует более полного анализа).

Предел в шесть строк кода может показаться несколько произвольным, поскольку в других книгах и средствах анализа используются другие значения. Согласно нашему опыту, предел в шесть строк обеспечивает правильный баланс между выявлением слиш-

ком большого и слишком малого количества клонов. Например, метод `ToString` может занимать три-четыре строки, и эти строки могут присутствовать во многих предметных объектах. Такие клоны можно игнорировать, поскольку они не являются тем, что мы ищем, а именно преднамеренным копированием функциональных возможностей.

## 4.1. Мотивация

Чтобы понять преимущества кода с минимумом дубликатов, мы обсудим влияние дублирования на обслуживаемость системы.

### Код с дубликатами сложнее анализировать

Обнаружив проблему, вы наверняка захотите ее исправить. Но для этого потребуется определить ее местонахождение. Вызывая существующий метод, вы легко сможете найти его исходный код. Но если код был прежде скопирован куда-то еще, проблема может оказаться совсем в другом месте. И единственный способ узнать об этом — использовать инструмент обнаружения клонов. Одним из таких инструментов является CPD, входящий в комплект средств анализа исходного кода PMD (https://pmd.github.io/). Некоторые версии Visual Studio оснащены встроенными инструментами выявления клонов.

Фундаментальная проблема дублирования заключается в том, что неизвестно, существует ли еще одна копия анализируемого кода, сколько еще таких копий имеется и где они расположены.

### В дублированный код сложно вносить изменения

Любой код может содержать ошибки. Но если ошибка содержится в дублируемом коде, она будет повторена несколько раз. Поэтому исправлять дублированный код гораздо сложнее, ибо исправления придется повторить несколько раз. Это в первую очередь требует точного понимания, какие исправления следует сделать и в дубликатах! Именно поэтому дублирование является типичным источником так называемых *регрессивных ошибок*, приводящих к тому, что код, нормально работающий до определенного момента, вдруг внезапно перестает работать (поскольку дубликат был забыт).

Та же проблема касается обычных изменений. Если код дублируется, изменения потребуется внести в несколько мест, а наличие большого числа дубликатов превращает внесение изменений в процесс с непредсказуемым результатом.

## 4.2. Как применять рекомендацию

Чтобы избежать ошибок, связанных с дублированием, никогда не используйте код повторно посредством копирования и вставки существующих фрагментов. Лучше поместите его в отдельный метод, чтобы при необходимости его можно было вызвать в другом месте. Вот почему рассмотренный в предыдущих главах метод извлечения кода является рабочей лошадкой, которая решает также проблему дублирования.

В примере, приведенном в начале главы, имеется дублирующий код, проверяющий контрольную сумму, который является очевидным кандидатом на извлечение. Для устранения дублирования методом извлечения дубликат (или его часть) выделяется в новый метод, который затем вызывается в нужных местах.

В главе 2 извлеченный метод стал локальным методом класса. Но это не сработает, если дубликаты находятся в разных классах, как, например, в классах `CheckingAccount` и `SavingsAccount`. Для решения данной проблемы можно создать служебный класс и поместить извлекаемый метод в него. В этом примере у нас уже есть соответствующий класс (`Accounts`), куда можно добавить новый статический метод `IsValid`:

```
public static bool IsValid(string number)
{
    int sum = 0;
    for (int i = 0; i < number.Length; i++)
    {
        sum = sum + (9 - i) * (int)Char.GetNumericValue(number[i]);
    }
    return sum % 11 == 0;
}
```

Этот метод вызывается в классе `CheckingAccount`:

```
public class CheckingAccount
{
    private int transferLimit = 100;
```

```
    public Transfer MakeTransfer(string counterAccount, Money amount)
    {
        // 1. Проверка превышения лимита:
        if (amount.GreaterThan(this.transferLimit))
        {
            throw new BusinessException("Limit exceeded!");
        }
        if (Accounts.IsValid(counterAccount))
        { ❶
            // 2. Поиск контрсчета и создание объекта перевода:
            CheckingAccount acct = Accounts.FindAcctByNumber(counterAccount);
            Transfer result = new Transfer(this, acct, amount); ❷
            return result;
        }
        else
        {
            throw new BusinessException("Invalid account number!");
        }
    }
}
```

❶ Начало короткого клона (три строки).
❷ Конец короткого клона (три строки).

И в классе `SavingsAccount`:

```
public class SavingsAccount
{
    public CheckingAccount RegisteredCounterAccount { get; set; }

    public Transfer MakeTransfer(string counterAccount, Money amount)
    {
        // 1. Проверка контрольной суммы 9-значного номера банковского счета
        // на кратность 11:
        if (Accounts.IsValid(counterAccount))
        { ❶
            // 2. Поиск контрсчета и создание объекта перевода:
            CheckingAccount acct = Accounts.FindAcctByNumber(counterAccount);
            Transfer result = new Transfer(this, acct, amount); ❷
            if (result.CounterAccount.Equals(this.RegisteredCounterAccount))
            {
                return result;
            }
            else
            {
                throw new BusinessException("Counter-account not registered!");
            }
```

```
        }
        else
        {
            throw new BusinessException("Invalid account number!!");
        }
    }
}
```

❶ Начало короткого клона (три строки).
❷ Конец короткого клона (три строки).

Задача решена: согласно определению дубликатов в начале этой главы, клоны исчезли (так как повторяющийся фрагмент содержит менее шести строк). Но остались следующие вопросы:

- несмотря на то что клоны исчезли, в двух классах все еще имеется повторяемая логика;
- извлеченный фрагмент потребовалось поместить в третий класс только потому, что в языке C# любой метод должен принадлежать классу (или структуре). Класс, куда добавлен извлеченный метод, рискует превратиться в сборную солянку из не связанных между собой методов. Создание *большого класса не сулит ничего хорошего* и ведет к созданию *тесных связей.* Наличие больших классов нежелательно, поскольку это сигнализирует о наличии ряда не связанных между собой функций. А это, как правило, приводит к образованию тесных связей, требующих знания деталей реализации большого класса для взаимодействия с ним. (Более подробно эта проблема обсуждается в главе 6.)

Рассматриваемый в следующем разделе прием рефакторинга помогает решить эти проблемы.

### Извлечение суперкласса

В предыдущих фрагментах кода сосуществуют два отдельных класса для расчетного и сберегательного счетов. Они функционально связаны между собой. Однако они никак не связаны в языке C# (это всего лишь два разных класса, наследующих общий класс `System.Object`). Оба имеют общую функциональность (проверка контрольной суммы), которая становится дубликатом при создании класса `SavingsAccount` путем копирования и вставки (с последую-

щим изменением) класса CheckingAccount. Можно сказать, что расчетный счет — это особый вид (обобщенного) банковского счета, как и сберегательный, который также является особым видом (обобщенного) банковского счета. Язык C# (как и другие объектно-ориентированные языки) имеет возможность представлять связь между чем-то обобщенным и чем-то более конкретным с помощью наследования классов.

Метод *извлечения суперкласса* использует эту особенность, извлекая фрагмент кода не просто в метод, а в новый класс, который становится суперклассом оригинального класса. То есть суть применения этого метода заключается в создании нового класса Account:

```
public class Account
{
    public virtual Transfer MakeTransfer(string counterAccount, Money amount)
    {
        // 1. Проверка контрольной суммы 9-значного номера банковского счета
        // на крат-ность 11:
        int sum = 0; ❶
        for (int i = 0; i < counterAccount.Length; i++)
        {
            sum = sum + (9 - i) * (int)Char.
                GetNumericValue(counterAccount[i]);
        }
        if (sum % 11 == 0)
        {
            // 2. Look up counter account and make transfer object:
            CheckingAccount acct = Accounts.FindAcctByNumber(counterAccount);
            Transfer result = new Transfer(this, acct, amount); ❷
            return result;
        }
        else
        {
            throw new BusinessException("Invalid account number!");
        }
    }
}
```

❶ Начало извлеченного кода.
❷ Конец извлеченного кода.

Новый суперкласс Account содержит общую логику двух особых видов счетов. Теперь следует превратить классы CheckingAccount и

SavingsAccount в подклассы этого нового суперкласса. Ниже представлена новая версия класса CheckingAccount:

```
public class CheckingAccount : Account
{
    private int transferLimit = 100;

    public override Transfer MakeTransfer(string counterAccount, Money amount)
    {
        if (amount.GreaterThan(this.transferLimit))
        {
            throw new BusinessException("Limit exceeded!");
        }
        return base.MakeTransfer(counterAccount, amount);
    }
}
```

Класс CheckingAccount объявляет свойство transferLimit и переопределяет метод MakeTransfer. Его метод MakeTransfer сначала проверяет превышение лимита переводимой суммы и затем вызывает метод MakeTransfer суперкласса для создания фактического перевода.

Новая версия класса SavingsAccount действует аналогично:

```
public class SavingsAccount : Account
{
    public CheckingAccount RegisteredCounterAccount { get; set; }

    public override Transfer MakeTransfer(string counterAccount, Money amount)
    {
        Transfer result = base.MakeTransfer(counterAccount, amount);
        if (result.CounterAccount.Equals(this.RegisteredCounterAccount))
        {
            return result;
        }
        else
        {
            throw new BusinessException("Counter-account not registered!");
        }
    }
}
```

Класс SavingsAccount объявляет свойство RegisteredCounterAccount и метод CheckingAccount, переопределяющий метод MakeTransfer. В его методе MakeTransfer не нужно проверять лимит (поскольку сберегательные счета не имеют лимита). Вместо этого он сразу

вызывает метод MakeTransfer суперкласса для создания перевода. Затем он проверяет, выполняется ли перевод на зарегистрированный счет.

Все функциональные возможности теперь находятся именно там, где они и должны быть. Функциональность создания перевода, одинаковая для всех счетов, находится в классе Account, в то время как специфические проверки выполняются в соответствующих им подклассах. Дублирование полностью устранено.

И наконец, следует отметить, что метод MakeTransfer суперкласса Account выполняет два действия. И хотя проблема дублирования, возникшая после копирования и вставки метода CheckingAccount, решена, внесем еще одно изменение, поместив проверку контрольной суммы на кратность 11 в отдельный метод, чтобы еще больше упростить обслуживание нового класса Account:

```
public class Account
{
    public Transfer MakeTransfer(string counterAccount, Money amount)
    {
        if (IsValid(counterAccount))
        {
            CheckingAccount acct = Accounts.FindAcctByNumber(counterAccount);
            return new Transfer(this, acct, amount);
        }
        else
        {
            throw new BusinessException("Invalid account number!");
        }
    }

    public static bool IsValid(string number)
    {
        int sum = 0;
        for (int i = 0; i < number.Length; i++)
        {
            sum = sum + (9 - i) * (int)Char.GetNumericValue(number[i]);
        }
        return sum % 11 == 0;
    }
}
```

Класс Account — именно то место, где должен находиться извлеченный метод IsValid.

## 4.3. Типичные возражения против исключения дублирования

В этом разделе рассматриваются типичные возражения против необходимости избегать дублирования кода. Как показывает наш опыт, многие разработчики приводят следующие аргументы в пользу дублирования фрагментов из других баз кода: это «неизбежно», и эта часть кода «никогда не изменится».

### Копирование фрагментов из другой базы кода допустимо

*«Копирование кода из другого кода не является проблемой, поскольку она не создает дубликата в базе кода текущей системы»*.

С технической точки зрения, действительно в базе кода текущей системы дубликаты не создаются. Копирование кода из другой системы кажется полезным, если код реализует точно такую же задачу. Однако это способствует появлению проблем в следующих ситуациях:

*Другая (оригинальная) база кода по-прежнему обслуживается*

Копия ничего не выиграет при улучшении оригинальной базы кода. Поэтому предпочтительнее не копировать, а импортировать необходимые функции (то есть добавлять другую базу кода в путь к классам системы).

*Другая база кода больше не обслуживается, и выполняется ее переделка*

В этом случае определенно не следует копировать код. Обычно переделка бывает вызвана проблемами с обслуживаемостью или сменой технологий. В случае проблем с обслуживаемостью не стоит копировать код, поскольку копируемый код (часто) сложно обслуживать. В случае смены технологий ограничения старой технологии переносятся в новую базу кода, что, например, делает невозможным использование абстракций, необходимых для эффективного повторного использования функциональных возможностей.

### При незначительных изменениях дублирование неизбежно

*«Нам нужно внести лишь незначительные изменения, поэтому дублирование кода неизбежно».*

Действительно, системы часто содержат варианты одних и тех же функций с небольшими различиями. Например, некоторые функции несколько отличаются для разных операционных систем, версий (по соображениям обратной совместимости) или групп клиентов. Однако это не означает, что без дублирования не обойтись. Нужно найти те части кода, которые являются общими для всех вариантов, и поместить их в общий суперкласс, как это делается в примерах, представленных в этой главе. Старайтесь изменить модель кода так, чтобы она стала четкой, изолированной и тестируемой.

### Этот код никогда не изменится

*«Этот код никогда не изменится, поэтому от его дублирования не будет никакого вреда».*

Если вы абсолютно уверены, что код никогда не изменится, его дублирование не создаст проблем (в том числе и для обслуживания). Но для этого вы должны быть абсолютно уверены, что рассматриваемый код не содержит никаких ошибок, которые потребуется исправить. Кроме того, реальность такова, что причин внесения изменений в системы очень много, и любая из них может привести к необходимости изменения тех частей, в которых, казалось, никогда не придется что-либо менять:

- *функциональные требования* к системе могут измениться из-за смены пользователей, модели поведения или способов организации бизнеса;
- в *организации* может смениться собственник, ответственность, подход к разработке, процесс разработки или законодательные требования;
- может смениться *технологическая основа,* на которую опирается ваша система, например: операционная система, библиотеки, структуры или интерфейсы других приложений;

- сам *код* может измениться из-за ошибок, оптимизации или даже вследствие косметических улучшений.

Вот почему утверждение, что код никогда не изменится, как правило, является необоснованным. То есть дублирование кода надо воспринимать как допущение определенного риска, с последствиями которого в дальнейшем придется иметь дело.

Ваш код *будет* меняться. Это на самом деле неизбежно.

### Дублирование всех файлов допустимо в целях создания их резервных копий

*«Мы храним копии всех файлов кода в качестве резервных копий. Каждая из резервных копий, естественно, является дубликатом всех прочих версий*».

Мы рекомендуем хранить резервные копии, но не применять подход, на котором основано это возражение (хранение внутри базы кода). Системы контроля версий, такие как Microsoft TFS, SVN или Git, обеспечивают гораздо лучший механизм резервного копирования. Если они недоступны, поместите файлы резервных копий в каталог рядом с корневым каталогом проекта, а не внутри. Почему? Потому что рано или поздно вы запутаетесь в том, какой вариант файлов является правильным.

### Модульные тесты защитят меня

*«Модульные тесты отследят, если с дубликатом что-то пойдет не так».*

Это верно, но только если дубликаты находятся в одном методе и модульный тест метода охватывает оба клона. Если дубликаты разбросаны по нескольким методам, это верно, только если инструмент анализа предупреждает об изменениях дубликатов. В противном случае модульные тесты не подадут сигнала о том, что изменился только один из дубликатов. Следовательно, нельзя полагаться лишь на тесты (выявление симптомов) вместо исключения главной причины проблем (устранения дублирования кода). Не следует полагать, что возможные проблемы будут устранены позже в процессе разработки, если их можно избежать прямо сейчас.

### Дублирование строковых литералов неизбежно и совершенно безвредно

*«Мне нужны длинные строковые литералы с большим количеством повторов в них. Дублирование их неизбежно и не принесет вреда, поскольку это просто литералы».*

Это разновидность одного из возражений, приведенных в главе 2 (раздел «Этот блок кода невозможно разделить»). Мы часто сталкиваемся с программами на C#, содержащими длинные SQL-запросы, XML- или HTML-документы в виде строковых литералов. Иногда такие литералы являются полными клонами, но чаще повторяются только их части. Например, мы видели SQL-запросы, содержащие больше ста строк кода, различающиеся только порядком сортировки (`order by asc` вместо `order by desc`). Этот тип дублирования не безобиден, несмотря на то что чисто технически не имеет отношения к логике языка C#. И он не является неизбежным. Этого типа дублирования можно избежать с помощью следующих простых способов:

- извлеките метод, который с помощью объединения строк и параметров будет обрабатывать разные варианты;
- используйте механизм шаблонов для получения HTML-разметки из малых неповторяющихся фрагментов, которые хранятся в отдельных файлах.

## 4.4. Дополнительные сведения

Исключение дублирования ведет к уменьшению размеров базы кода, более подробно об этом рассказывается в главе 9. Просмотрите в главе 2 описание метода извлечения для разделения блоков кода, помогающего сделать их более удобными для повторного использования.

---

### Как в компании SIG устанавливается рейтинг дублирования

Объем дублирования является одним из восьми критериев SIG/TÜViT оценки системных свойств для достижения уверенной обслуживаемости продукта. При оценке рейтинга дублирования учитывается только наличие клонов первого типа (то есть тексту-

ально идентичных), содержащих не менее шести строк кода, за исключением клонов, состоящих исключительно из операторов `import`. Затем клоны подразделяются на две категории риска: избыточные и неизбыточные, как показано ниже. Возьмем для примера фрагмент из 10 строк кода, который повторяется в коде трижды. Другими словами, существует группа из трех клонов, каждый из которых содержит 10 строк кода. Теоретически два из них можно удалить, и они считаются технически избыточными. Следовательно, 10 + 10 = 20 строк кода относятся к категории избыточных. Один клон классифицируется как неизбыточный, и, следовательно, 10 строк кода классифицируются как неизбыточные. Для получения рейтинга в 4 звезды не более 4,6% от общего количества строк должно классифицироваться как избыточные. Это демонстрирует табл. 4.1.

***Таблица 4.1 ❖ Минимальные границы для системы с рейтингом в 4 звезды по дублированию кода (версия 2015 критериев оценки SIG/TÜViT)***

| Строки кода, оцениваемые как... | Допустимый процент для систем с 4 звездами |
|---|---|
| ... неизбыточные | Не менее 95,4% |
| ... избыточные | Не более 4,6% |

На рис. 4.1 изображены примеры профилей качества трех систем:

- слева — диаграмма для системы с открытым исходным кодом, в данном случае Jenkins;
- в центре — диаграмма для анонимной системы, используемой SIG в качестве эталона 4-звездной системы;
- справа — диаграмма, отображающая граничные точки характеристик для достижения качества, оцениваемого в 4 звезды.

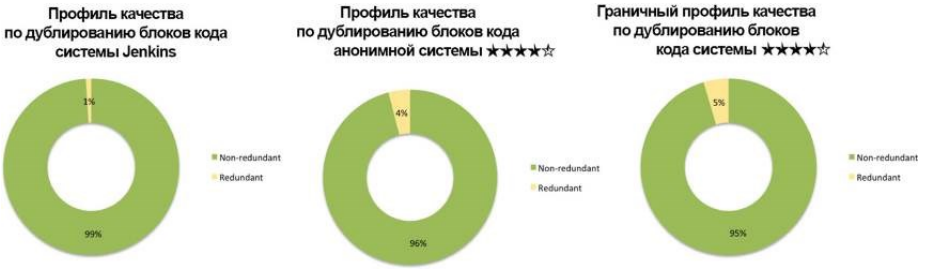


**Рис. 4.1 ❖** Три профиля качества по дублированию кода

# Глава 5

# Стремитесь к уменьшению размеров интерфейсов

Связки данных, встречающихся совместно,
надо превращать в самостоятельный класс.

*Мартин Фаулер*

Рекомендация:

- **передавайте в блоки кода не более 4 параметров**.
- при необходимости **объединяйте параметры в объекты**.
- это улучшит обслуживаемость, поскольку уменьшение числа параметров **упростит понимание и повторное использование** блоков кода.

Программисты ежедневно сталкиваются с ситуациями, когда создание длинных списков параметров кажется неизбежными. В порыве скорее достичь цели можно добавить еще несколько параметров в метод, чтобы заставить его обрабатывать исключительные случаи. Но в долгосрочной перспективе такой подход ведет к появлению методов, которые трудно обслуживать и повторно использовать. Чтобы сохранить код легко обслуживаемым, избегайте длинных списков параметров или *интерфейсов блоков кода* большого размера, ограничивая число параметров.

Типичным примером блока с большим количеством параметров является метод `render` класса `BoardPanel` из JPacman. Этот метод рисует квадрат и его обитателей (призраки и гранулы) в прямоугольной области, определяемой параметрами `x`, `y`, `w` и `h`.

```
/// <summary>
/// Отображает единственный прямоугольник в графическом контексте
/// прямоугольной формы.
///
```

```
/// <param name="square">Отображаемый квадрат.</param>
/// <param name="g">Графический контекст.</param>
/// <param name="x">Координата x начала рисования.</param>
/// <param name="y">Координата y начала рисования.</param>
/// <param name="w">Ширина квадрата (в пикселях).</param>
/// <param name="h">Высота квадрата (в пикселях).</param>
private void Render(Square square, Graphics g, int x, int y, int w, int h)
{
    square.Sprite.Draw(g, x, y, w, h);
    foreach (Unit unit in square.Occupants)
    {
        unit.Sprite.Draw(g, x, y, w, h);
    }
}
```

Число параметров метода превышает допустимый предел 4. Больше всего неприятностей могут доставить последние четыре аргумента, имеющие один и тот же тип `int`, затрудняющие понимание метода и способствующие появлению большого количества ошибок. В конце дня, после длительной работы с кодом, даже опытный разработчик может запутаться в параметрах `x`, `y`, `w` и `h`, допустив ошибки, которые не сможет определить компилятор и, возможно, даже модульные тесты.

Поскольку переменные `x`, `y`, `w` и `h` связаны между собой (они определяют координаты его начальной точки, ширину и высоту) и метод `render` не использует их по отдельности, имеет смысл сгруппировать их в объект класса `Rectangle`. Следующие фрагменты кода демонстрируют класс `Rectangle` и измененный метод `render`:

```
public class Rectangle
{
    public Point Position { get; set; }

    public int Width { get; set; }

    public int Height { get; set; }

    public Rectangle(Point position, int width, int height)
    {
        this.Position = position;
        this.Width = width;
        this.Height = height;
    }
```

```
}

/// <summary>
/// Отображает единственный прямоугольник в графическом контексте
/// прямоугольной формы.
///
/// <param name="square">Отображаемый квадрат.</param>
/// <param name="g">Графический контекст.</param>
/// <param name="r">Координаты и размеры квадрата.</param>
private void Render(Square square, Graphics g, Rectangle r)
{
    Point position = r.Position;
    square.Sprite.Draw(g, position.X, position.Y, r.Width, r.Height);
    foreach (Unit unit in square.Occupants)
    {
        unit.Sprite.Draw(g, position.X, position.Y, r.Width, r.Height);
    }
}
```

Теперь метод `render` принимает три параметра вместо шести. Кроме того, в системе имеется готовый к использованию класс `Rectangle`. Это позволяет сократить интерфейс метода `draw`:

```
private void Render(Square square, Graphics g, Rectangle r)
{
    Point position = r.Position;
    square.Sprite.Draw(g, r);
    foreach (Unit unit in square.Occupants)
    {
        unit.Sprite.Draw(g, r);
    }
}
```

Только что проделанное изменение является примером применения шаблона рефакторинга *предоставление параметров в виде объекта*. Сокращение длинных списков параметров, продемонстрированное в предыдущем примере, улучшает удобочитаемость кода. В следующем разделе поясняется, как краткие интерфейсы способствуют улучшению обслуживаемости системы.

## 5.1. Мотивация

Как уже было упомянуто во введении, имеются веские причины сокращения размеров интерфейсов и применения объектов для передачи ряда параметров. Методы с интерфейсами небольшого

размера проще и понятнее. Кроме того, такие методы легче использовать и изменять, поскольку они не требуют большого количества входных данных.

### Интерфейсы небольшого размера упрощают понимание и повторное использование кода

По мере роста базы кода основные классы превращаются в прикладной программный интерфейс, который становится основой для прочего кода системы. Для уменьшения общего объема базы кода (более подробно об этом в главе 9) и увеличения скорости разработки большое значение имеет ясный н компактный интерфейс методов основных классов. Предположим, что нужно сохранить объект `ProductOrder` в базе данных, какой метод вы выберете — `ProductOrderDao.store(ProductOrder order)` или `ProductOrderDao.store( ProductOrder order, String databaseUser, String databaseName, boolean validateBeforeStore, boolean closeDbConnection)`?

### В методы с компактным интерфейсом проще вносить изменения

Раздутые интерфейсы не только делают методы трудно читаемыми, но и, как правило, указывают на чрезмерность обязанностей, возложенных на них (особенно когда не получается сгруппировать параметры в объекты). В этом смысле размер интерфейсов зависит от размера и сложности блоков кода. Совершенно очевидно, что в методы с большими интерфейсами сложнее вносить изменения. Если имеется метод, скажем, с восемью параметрами и в теле метода выполняется масса действий, что затрудняет его анализ, имеет смысл разделить этот метод на несколько частей. В результате появится несколько методов, каждый из которых отвечает за единственное действие и имеет небольшое число параметров! Теперь станет гораздо проще вносить изменения в любой из этих методов, поскольку легко можно найти тот фрагмент кода, который следует изменить.

## 5.2. Как применять рекомендацию

Теперь у вас не должно остаться сомнений в удобстве компактных интерфейсов. Но насколько компактными они должны быть? С практической точки зрения верхняя граница в четыре параметра выглядит вполне разумной: метод с четырьмя параметрами все еще выглядит понятным, а с пятью параметрами уже становится трудно читаемым, и на него явно возложено слишком много обязанностей.

Как добиться компактности интерфейсов? Для начала рассмотрим способы преобразования больших интерфейсов, не забывая, что они не являются проблемой сами по себе, а скорее выступают в роли *видимого признака* наличия настоящей проблемы, которая заключена в модели данных или в особенностях кода. То есть большой размер интерфейса можно рассматривать как симптом, что модель данных нуждается в улучшении.

Обычно большие интерфейсы — не главная проблема, а скорее признак наличия более глубинной проблемы с обслуживаемостью.

Пусть имеется метод `DoBuildAndSendMail` с девятью параметрами, предназначенный для создания и отправки сообщений по электронной почте. Анализ списка параметров не проясняет, что происходит в теле метода:

```
public void DoBuildAndSendMail(MailMan m, string firstName, string lastName,
    string division, string subject, MailFont font, string message1,
    string message2, string message3)
{
    // Сформировать адрес электронной почты
    string mId = $"{firstName[0]}.{lastName.Substring(0, 7)}" +
        $"@{division.Substring(0, 5)}.compa.ny";
    // Сформировать сообщение, указав тип содержимого и исходный текст
    MailMessage mMessage = FormatMessage(font,
        message1 + message2 + message3);
    // Отправить сообщение
    m.Send(mId, subject, mMessage);
}
```

Метод `DoBuildAndSendMail` явно имеет слишком много обязанностей, поскольку формирование адреса электронной почты имеет мало общего с отправкой сообщения. Кроме того, передача тела

сообщения в пяти параметрах способна сбить с толку кого угодно! Предлагаем пересмотреть код метода:

```
public void DoBuildAndSendMail(MailMan m, MailAddress mAddress,
    MailBody mBody)
{
    // Сформировать электронное письмо
    Mail mail = new Mail(mAddress, mBody);
    // Послать электронное письмо
    m.SendMail(mail);
}

public class Mail
{
    public MailAddress Address { get; set; }
    public MailBody Body { get; set; }

    public Mail(MailAddress mAddress, MailBody mBody)
    {
        this.Address = mAddress;
        this.Body = mBody;
    }
}

public class MailBody
{
    public string Subject { get; set; }
    public MailMessage Message { get; set; }

    public MailBody(string subject, MailMessage message)
    {
        this.Subject = subject;
        this.Message = message;
    }
}

public class MailAddress
{
    public string MsgId { get; private set; }

    public MailAddress(string firstName, string lastName,
        string division)
    {
        this.MsgId = $"{firstName[0]}.{lastName.Substring(0, 7)}" +
            $"@{division.Substring(0, 5)}.compa.ny";
    }
}
```

Теперь метод `DoBuildAndSendMail` стал значительно проще. Конечно, сейчас перед вызовом метода требуется сформировать электронный адрес и текст сообщения. Но если требуется отправить одно и то же сообщение на несколько адресов, сообщение придется создать лишь один раз. То же относится к случаю, когда нужно отправить кучу сообщений на один и тот же адрес. В конечном итоге мы разделили задачи между несколькими, вновь созданными, элегантными и хорошо структурированными классами.

Представленные в этой главе примеры группируют все параметры в объекты. Такие объекты часто называют *объектами передачи данных,* или *объектами параметров.* Эти новые объекты представляют значимые понятия предметной области. Начальная точка, ширина и высота определяют прямоугольник, поэтому группировка их в класс прямоугольника `Rectangle` выглядит вполне естественно. Аналогично имя, фамилия и отдел составляют адрес, поэтому объединение их в класс `MailAddress` также не лишено смысла. Очень вероятно, что эти классы будут часто использоваться в базе кода, поскольку являются удобными обобщениями, а не только потому, что с их помощью можно уменьшить число параметров.

А как поступить, если имеется ряд параметров, никак не связанных между собой? Их также можно собрать в объект, но такой объект, скорее всего, можно будет использовать только один раз. В подобных случаях часто применяется другой подход, демонстрируемый в следующем примере.

Предположим, что создается библиотека для рисования диаграмм на холсте `System.Drawing.Graphics`, например гистограмм и круговых диаграмм. Чтобы нарисовать красивую диаграмму, требуется не так много сведений, например нужны размер области для рисования, конфигурация оси категорий, конфигурация оси значений, собственно набор данных и т. д. Ниже показан один из способов передачи этой информации в библиотеку:

```
public static void DrawBarChart(Graphics g,
    CategoryItemRendererState state,
    Rectangle graphArea,
    CategoryPlot plot,
    CategoryAxis domainAxis,
    ValueAxis rangeAxis,
```

```
    CategoryDataset dataset)
{
    // ..
}
```

Этот метод содержит семь параметров, на три больше, чем рекомендовано. Кроме того, в любом вызове метода `DrawBarChart` необходимо указать значения всех семи параметров. А что, если библиотека рисования диаграмм должна подставлять значения по умолчанию везде, где это только возможно? Реализовать это можно путем переопределения методов, например определив версию метода `DrawBarChart` с двумя параметрами:

```
public static void DrawBarChart(Graphics g, CategoryDataset dataset)
{
    Charts.DrawBarChart(g,
        CategoryItemRendererState.DEFAULT,
        new Rectangle(new Point(0, 0), 100, 100),
        CategoryPlot.DEFAULT,
        CategoryAxis.DEFAULT,
        ValueAxis.DEFAULT,
        dataset);
}
```

Этот метод можно использовать в случаях, когда значения по умолчанию должны использоваться для всех параметров, для которых они определены. Однако это лишь частный случай. Для охвата всех ситуаций потребуется определить еще несколько подобных версий методов. И версия с семью параметрами по-прежнему останется среди них.

Другим решением проблемы является использование шаблона *замены методов методами объекта,* уже описанного в главе 2. Там этот шаблон применялся для сокращения размеров методов, но его можно использовать и для уменьшения числа параметров.

Чтобы применить шаблон замены методов методами объекта в этом примере, определим класс `BarChart`:

```
public class BarChart
    {
        private CategoryItemRendererState state =
            CategoryItemRendererState.DEFAULT;
        private Rectangle graphArea = new Rectangle(new Point(0, 0), 100, 100);
        private CategoryPlot plot = CategoryPlot.DEFAULT;
        private CategoryAxis domainAxis = CategoryAxis.DEFAULT;
```

```
        private ValueAxis rangeAxis = ValueAxis.DEFAULT;
        private CategoryDataset dataset = CategoryDataset.DEFAULT;

        public BarChart Draw(Graphics g)
        {
            // ..
            return this;
        }

        public ValueAxis GetRangeAxis()
        {
            return rangeAxis;
        }

        public BarChart SetRangeAxis(ValueAxis rangeAxis)
        {
            this.rangeAxis = rangeAxis;
            return this;
        }

        // Другие методы чтения и записи значений.

}
```

Статический метод `DrawBarChart` в оригинальной версии мы заменили нестатическим методом `Draw`. Шесть из семи параметров метода `DrawBarChart` были преобразованы в закрытые свойства класса `BarChart`, получившие соответствующие методы чтения и записи. Все свойства имеют значения по умолчанию. Параметр `g` (типа `System.Drawing.Graphics`) решено сохранить как параметр метода `Draw`. Это разумное решение, поскольку в любом случае придется использовать объект `Graphics`, для которого не существует никакого подходящего значения по умолчанию. Но и в этом нет необходимости, можно просто преобразовать `g` в седьмое закрытое свойство и добавить для него методы чтения и записи.

Кроме того, все методы чтения и записи возвращают `this` для создания так называемого *потокового интерфейса.* Это позволяет составлять цепочки из вызовов методов записи, например:

```
private void ShowMyBarChart()
{
    Graphics g = this.CreateGraphics();
    BarChart b = new BarChart()
        .SetRangeAxis(myValueAxis)
```

```
        .SetDataset(myDataset)
        .Draw(g);
}
```

Здесь, перед вызовом метода `Draw`, мы установили диапазоны значений по осям, набор данных, холст `g` и использовали значения по умолчанию для других свойств класса `BarChart`. Такой подход позволяет использовать большее или меньшее число значений по умолчанию, без необходимости определять дополнительные перегруженные версии метода `Draw`.

## 5.3. Типичные возражения против сокращения размеров интерфейсов

Для уменьшения размеров интерфейсов требуется определенное время. Ниже рассматриваются типичные возражения, оспаривающие полезность приложения усилий в этом направлении.

### Возражение: объекты параметров требуют определения конструкторов с большим количеством параметров

*«Объект параметров, созданный мной, имеет конструктор с большим количеством параметров».*

Итак, в процессе рефакторинга метода с целью сократить число параметров был создан некоторый объект. Может случиться так, что теперь конструктор этого объекта имеет слишком много параметров. Обычно это означает, что можно выявить более тонкие различия внутри самого объекта. Помните первый пример рефакторинга метода `render`? Там параметры, определяющие прямоугольник, были объединены в объект, конструктор которого имел не четыре, а три параметра, поскольку параметры `x` и `y` были объединены в объект `Point`. Это говорит о том, что нужно думать о структуре добавляемого объекта и его взаимодействии с другими частями кода, а не об отказе от него.

### Преобразование интерфейсов большого размера не улучшает ситуацию

*«После внесения изменений в один метод приходится передавать слишком много параметров в другой».*

Не всегда легко сократить интерфейс. Обычно для этого нужно изменить несколько методов. Как правило, следует продолжить разделять возложенные на методы обязанности, предоставляя доступ к простым параметрам, только когда они используются отдельно. Например, модернизированный метод `render` вынужден извлекать все параметры из объекта `Rectangle`, чтобы передать их в вызов метода `Draw`. Поэтому было бы лучше провести также рефакторинг метода `Draw`, чтобы параметры `x`, `y`, `w` и `h` извлекались только внутри него. В этом случае метод `render` мог бы просто передавать объект `Rectangle` в метод `draw`, поскольку ему не требуется изменять свойства этого объекта перед началом рисования!

### Фреймворки или библиотеки предоставляют интерфейсы с длинными списками параметров

*«Интерфейс используемого фреймворка включает девять параметров. Как реализовать такой интерфейс, не нарушая рекомендацию об интерфейсах блоков кода?»*

Иногда фреймворки или библиотеки определяют интерфейсы или классы с методами, содержащими длинные списки параметров. Реализация или переопределение этих методов неизбежно приводит к появлению длинных списков параметров в коде. Такие нарушения невозможно предотвратить, но можно смягчить их отрицательное влияние. Для смягчения последствий таких нарушений, вызванных применением сторонних фреймворков или библиотек, следует изолировать эти нарушения, например, с помощью оберток или адаптеров. Также стоит подумать о выборе другого фреймворка или библиотеки, но это может сильно повлиять на другие части системы.

## 5.4. Дополнительные сведения

Методы, имеющие множество обязанностей, как правило, являются объемными и сложными. Поэтому стоит вспомнить рекомендации по упрощению блоков кода, приведенные в главах 2 и 3.

### Как в компании SIG устанавливается рейтинг по размерам интерфейсов блоков кода

Размер интерфейсов блоков кода является одним из восьми критериев SIG/TÜViT оценки системных свойств для достижения уверенной обслуживаемости продукта. При оценке рейтинга по размерам интерфейсов все блоки кода помещаются в одну из четырех категорий риска, в зависимости от количества параметров. В табл. 5.1 перечислены эти четыре категории в соответствии с версией 2015 критериев оценки SIG/TÜViT.

***Таблица 5.1 ❖ Минимальные границы для системы с рейтингом в 4 звезды по размерам интерфейсов блоков кода (версия 2015 критериев оценки SIG/TÜViT)***

| Строк кода в методах с... | Допустимый процент для интерфейсов для систем с 4 звездами |
|---|---|
| ... более чем семью параметрами | Не более 0,7% |
| ... пятью и более параметрами | Не более 2,7% |
| ... тремя и более параметрами | Не более 13,8% |
| ... не более чем двумя параметрами | Не менее 86,2% |

На рис. 5.1 изображены примеры профилей качества трех систем:

- слева — диаграмма для системы с открытым исходным кодом, в данном случае Jenkins;
- в центре — диаграмма для анонимной системы, используемой SIG в качестве эталона 4-звездной системы;
- справа — диаграмма, отображающая граничные точки характеристик для достижения качества, оцениваемого в 4 звезды.

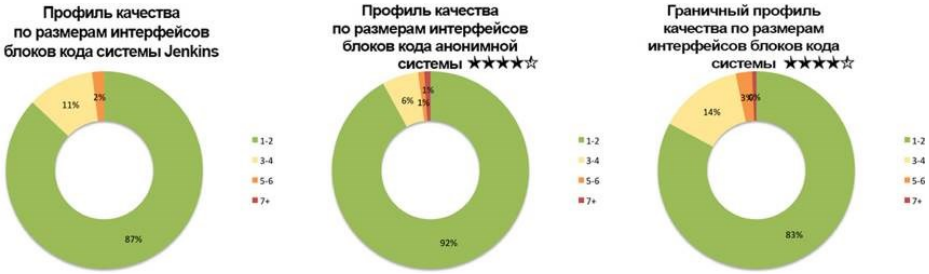


**Рис. 5.1** ❖ Три профиля качества по размерам интерфейсов блоков кода

# Глава 6

# Разделяйте задачи на модули

В сложной, тесно связанной системе
несчастные случаи неизбежны.

*Чарльз Перроу (Charles Perrow),*
*теория обычных аварий в одной фразе*

Рекомендация:

- **избегайте модулей большого размера, чтобы исключить образование тесных связей между ними**;
- для этого **разделяйте сферы ответственности отдельных модулей и скрывайте детали реализации с помощью интерфейсов**;
- это улучшает обслуживаемость, поскольку модификацию слабо связанного кода **гораздо проще выполнять и контролировать**.

Все рекомендации, представленные в предыдущих главах, называются *рекомендациями для блоков кода*, поскольку все они касаются упрощения сопровождения отдельных блоков кода (методов и конструкторов). Рекомендация, представленная в этой главе, относится уже к уровню модулей.

Следует напомнить, что понятию модуля в объектно-ориентированных языках, таких как язык C#, соответствует класс.

Эта рекомендация относится к взаимоотношениям между классами. Она направлена на установление между ними гибких связей.

Рассмотрим реальный пример, демонстрирующий тесную связь между классами, и обсудим, почему это приводит к проблемам обслуживания. Этим примером послужит класс `UserService` из уровня служб веб-приложения, который в процессе разработки

начал разрастаться и рос до тех пор, пока не стал нарушать рекомендацию, приведенную в этой главе.

В первой итерации разработки класс `UserService` содержал лишь три метода:

```
public class UserService
    {
        public User LoadUser(string userId)
        {
            // ...
        }

        public bool DoesUserExist(string userId)
        {
            // ...
        }

        public User ChangeUserInfo(UserInfo userInfo)
        {
            // ...
        }
    }
    // end::UserSerice[]
}
```

Здесь серверная часть веб-приложения реализует интерфейс REST для клиентской части этой и других систем.

Интерфейс REST обеспечивает упрощенный порядок предоставления веб-служб. Это популярный способ предоставления функциональных возможностей сторонним системам. Ниже представлен класс, реализующий пользовательские операции с использованием класса `UserService`:

```
public class UserController : System.Web.Http.ApiController
{

    private readonly UserService userService = new UserService();

    // ...

    public System.Web.Http.IHttpActionResult GetUserById(string id)
    {
        User user = userService.LoadUser(id);
        if (user == null)
        {
```

```
            return NotFound();
        }
        return Ok(user);
    }
}
```

Во второй итерации разработки класс `UserService` остался без изменений. В третьей итерации были реализованы новые требования, что позволило пользователю регистрироваться для получения определенных уведомлений. Для удовлетворения этих требований в класс `UserService` были добавлены три новых метода:

```
public class UserService
    {
        public User LoadUser(string userId)
        {
            // ...
        }

        public bool DoesUserExist(string userId)
        {
            // ...
        }

        public User ChangeUserInfo(UserInfo userInfo)
        {
            // ...
        }

        public List<NotificationType> GetNotificationTypes(User user)
        {
            // ...
        }

        public void RegisterForNotifications(User user, NotificationType type)
        {
            // ...
        }

        public void UnregisterForNotifications(User user, NotificationType type)
        {
            // ...
        }
    }
    // end::UserSerice[]

}
```

Доступ к этим новым возможностям был реализован в виде отдельного класса:

```
public class NotificationController : System.Web.Http.ApiController
{
    private readonly UserService userService = new UserService();

    // ...

    public System.Web.Http.IHttpActionResult Register(string id,
        string notificationType)
    {
        User user = userService.LoadUser(id);
        userService.RegisterForNotifications(user,
            NotificationType.FromString(notificationType));
        return Ok();
    }

    [System.Web.Http.HttpPost]
    [System.Web.Http.ActionName("unregister")]
    public System.Web.Http.IHttpActionResult Unregister(string id,
        string notificationType)
    {
        User user = userService.LoadUser(id);
        userService.UnregisterForNotifications(user,
            NotificationType.FromString(notificationType));
        return Ok();
    }
}
```

В четвертой итерации были реализованы новые требования, обеспечивающие поиск, блокировку и вывод списка всех заблокированных пользователей (последнее требование руководства реализовано для целей отчетности). Все эти требования привели к добавлению новых методов в класс `UserService`.

```
public class UserService
    {
        public User LoadUser(string userId)
        {
            // ...
        }

        public bool DoesUserExist(string userId)
        {
            // ...
        }
```

```
        public User ChangeUserInfo(UserInfo userInfo)
        {
            // ...
        }

        public List<NotificationType> GetNotificationTypes(User user)
        {
            // ...
        }

        public void RegisterForNotifications(User user, NotificationType type)
        {
            // ...
        }

        public void UnregisterForNotifications(User user, NotificationType type)
        {
            // ...
        }

        public List<User> SearchUsers(UserInfo userInfo)
        {
            // ...
        }

        public void BlockUser(User user)
        {
            // ...
        }

        public List<User> GetAllBlockedUsers()
        {
            // ...
        }
    }
    // end::UserSerice[]

}
```

После завершения этой итерации класс вырос до внушительных размеров. На этот момент класс `UserService` стал наиболее часто используемой службой в системе. Три клиентских представления (страницы профиля, уведомлений и поиска) пользуются классом `UserService` через три службы REST API. Число входящих вызовов

из других классов *(нагрузка по входу*) возросло до 50. Размер класса превысил 300 строк кода.

Подобные классы страдают *синдромом большого класса,* упоминавшимся в главе 4. Их код содержит слишком много функциональных возможностей, а также зависит от деталей реализации окружающего кода. Как результат этот класс образует *тесные связи.* Он используется во множестве мест и *зависит* от деталей реализации внешнего кода. Например, он использует классы слоя данных для управления профилями пользователей, систему оповещения, поиска и блокировки других пользователей.

Под тесной связью подразумевается, что при изменении одной части системы возникает необходимость изменять другую. Она определяется не только непосредственными вызовами, но и соединениями через конфигурационные файлы, структуру базы данных или даже через функциональное назначение (с точки зрения бизнес-логики).

Проблема таких классов заключается в том, что они становятся горячей точкой при обслуживании. Все функции, связанные (даже отдаленно) с пользователями, в конечном итоге сходятся в классе `UserService`. Это пример неправильного *разделения обязанностей.* Даже разработчикам все сложнее разобраться в классе `UserService`, он разрастается и становится неуправляемым. Менее опытных разработчиков команды этот класс пугает, и они боятся вносить в него изменения.

Необходимо понимать значение двух принципов тесной связи между классами.

- Тесная связь является проблемой исходного кода на уровне класса. Все методы класса `UserService` соответствуют рекомендациям, представленным в предыдущих главах. Но *объединение* методов в классе `UserService` *делает класс* `UserService` *тесно связанным с классами, использующими его.*
- Понятия тесной и слабой связи являются относительными. Влияние тесной связи на обслуживаемость определяется числом вызовов методов и размером класса. То есть чем больше обращений к тесно связанному классу, тем меньше должен быть его размер. Если классы надежно изолированы,

число вызовов может быть большим, но связь при этом будет слабой, поскольку в ней участвует меньший объем кода.

## 6.1. Мотивация

Большим преимуществом маленьких классов является слабая связь между ними. Слабая связь обеспечивает большую гибкость на уровне классов, что упрощает дальнейшее внесение изменений. Под «гибкостью» здесь понимается возможность внесения изменений при ограниченности риска появления неожиданных последствий. То есть слабая связь позволяет разработчикам работать с изолированными частями кода, не оказывая воздействия на остальной код. Третье преимущество, которое нельзя недооценивать, заключается в простоте изучения базы кода для менее опытных разработчиков.

В следующих разделах мы рассмотрим преимущества наличия в системе только небольших и слабо связанных классов.

#### Небольшие слабо связанные модули позволяют разработчикам иметь дело с надежно изолированными частями системы

Когда класс тесно связан с другими классами, внесение изменений в его реализацию оказывает влияние на всю систему. Например, изменение интерфейса общедоступного метода приводит к изменению всех вызовов этого метода. Помимо необходимости приложения больших усилий, это увеличивает риск появления ошибок или несоответствий где-то в отдаленных местах системы.

#### Небольшие слабо обязанные модули упрощают навигацию по коду

Правильное разделение задач не только делает код гибким, упрощая его последующее изменение, но и облегчает анализ, поскольку классы инкапсулируют данные и реализацию для решения одной-единственной задачи. Подобно тому как можно было давать понятные имена методам, ответственным за что-то одно, классы, если они используются для достижения какой-то одной цели, также можно называть так, чтобы было сразу ясно, для чего они предна-

значены. Уверенность в том, что один класс отвечает только за что-то одно, носит название *принципа персональной ответственности.*

### Небольшие слабо связанные модули делают все области кода более понятными новым разработчикам

Классы, нарушающие *принцип персональной ответственности,* становятся тесно связанными и с течением времени накапливают в себе массу кода. Подобно классу `UserService`, представленному во введении к главе, такие классы пугают малоопытных разработчиков, и даже опытные разработчики не слишком уверенно себя чувствуют при внесении изменений в их реализацию. Базу кода с большим количеством классов и нечеткими границами ответственности очень трудно адаптировать к новым требованиям.

## 6.2. Как применять рекомендацию

В целом эта рекомендация предписывает сохранять классы маленькими (возлагая на них решение одной, конкретной задачи) и ограничивать число обращений к классу из кода вне самого класса. Далее описываются три приема, помогающих избежать образования тесной связи между классами.

### Разделение классов по решаемым задачам

Разработка классов, совместно реализующих функциональность программной системы, является очень важным этапом моделирования и проектирования объектно-ориентированных систем. В типичных программных проектах классы изначально являются логическими сущностями, которые реализуют единственную функциональность, но с течением времени на них могут возлагаться дополнительные обязанности. Для предотвращения синдрома больших классов разработчикам следует принимать меры для разделения класса, когда ему вменено несколько обязанностей.

Для демонстрации разделим класс `UserService` из введения к главе на три отдельных класса. Ниже приводятся два вновь созданных класса и измененный класс `UserService`:

```
public class UserNotificationService
{
    public IList<NotificationType> GetNotificationTypes(User user)
    {
        // ...
    }

    public void Register(User user, NotificationType type)
    {
        // ...
    }

    public void Unregister(User user, NotificationType type)
    {
        // ...
    }
}

public class UserBlockService
{
    public void BlockUser(User user)
    {
        // ...
    }

    public IList<User> GetAllBlockedUsers()
    {
        // ...
    }
}

public class UserService
{
    public User LoadUser(string userId)
    {
        // ...
    }

    public bool DoesUserExist(string userId)
    {
        // ...
    }

    public User ChangeUserInfo(UserInfo userInfo)
    {
        // ...
    }
```

```
    public IList<User> SearchUsers(UserInfo userInfo)
    {
        // ...
    }
}
```

После перераспределения ответственности между классами REST API реализация системы получилась более слабо связанной. Например, классу `UserService` не требуется знать, как устроена система уведомлений или логика блокирования пользователей. В дальнейшем разработчики, вероятнее всего, будут размещать новые функциональные возможности в отдельных классах, а не добавлять их в класс `UserService`.

### Сокрытие подробностей реализации за интерфейсами

Также слабую связанность можно обеспечить за счет сокрытия особенностей и деталей реализации с помощью высокоуровневого интерфейса. Рассмотрим класс, реализующий управление цифровой камерой и способной делать снимки как со вспышкой, так и без нее:

```
public class DigitalCamera
{
    public Image TakeSnapshot()
    {
        // ...
    }

    public void FlashLightOn()
    {
        // ...
    }

    public void FlashLightOff()
    {
        // ...
    }
}
```

Предположим, что этот код используется в приложении для смартфона, как показано ниже:

```
public class SmartphoneApp
```

```
{
    private static DigitalCamera camera = new DigitalCamera();

    public static void Main(string[] args)
    {
        // ...
        Image image = camera.TakeSnapshot();
        // ...
    }
}
```

Спустя какое-то время появились более продвинутые цифровые камеры. Кроме съемки фотографий, они могут записывать видео, содержат таймер, могут осуществлять приближение и отдаление. В результате в класс `DigitalCamera` была добавлена поддержка новых возможностей:

```
public class DigitalCamera
{
    public Image TakeSnapshot()
    {
        // ...
    }

    public void FlashLightOn()
    {
        // ...
    }

    public void FlaslLightOff()
    {
        // ...
    }

    public Image TakePanoramaSnapshot()
    {
        // ...
    }

    public Video Record()
    {
        // ...
    }

    public void SetTimer(int seconds)
    {
        // ...
```

```
    }

    public void ZoomIn()
    {
        // ...
    }

    public void ZoomOut()
    {
        // ...
    }
}
```

Из приведенного примера нетрудно понять, что расширенная версия класса `DigitalCamera` получилась гораздо больше первоначальной, обеспечивающей меньше функциональных возможностей.

Приложение для смартфона по-прежнему использует только три первоначальных метода. Однако, поскольку имеется лишь один класс `DigitalCamera`, приложение вынуждено использовать этот класс большего размера. Это ведет к образованию более тесной связи, чем необходимо. Если один или несколько дополнительных методов класса `DigitalCamera` изменится, придется просмотреть код всего приложения, чтобы убедиться, что такая модернизация ни на что не повлияла. И хотя приложение не использует ни одного нового метода, они остаются доступными для него.

Для ослабления связи можно использовать интерфейс, определяющий ограниченный список функциональных возможностей камеры как для обычных, так и для продвинутых камер:

```
public interface ISimpleDigitalCamera
{
    Image TakeSnapshot();

    void FlashLightOn();

    void FlashLightOff();
}

public class DigitalCamera : ISimpleDigitalCamera
{
    // ...
}

public class SmartphoneApp
```

```
{
    private static ISimpleDigitalCamera camera = SDK.GetCamera();

    public static void Main(string[] args)
    {
        // ...
        Image image = camera.TakeSnapshot();
        // ...
    }
}
```

Такое изменение приводит к ослаблению связи за счет более высокой степени инкапсуляции. Другими словами, классы, которые используют только основные функциональные возможности цифровых камер, теперь ничего не знают обо всех возможностях продвинутой цифровой камеры. Класс `SmartphoneApp` получает доступ лишь к интерфейсу `SimpleDigitalCamera`. Это гарантирует, что приложение `SmartphoneApp` не сможет использовать методов, предназначенные только для продвинутых камер.

Кроме того, система получилась более модульной: она структурирована так, что изменение одного класса оказывает минимальное влияние на другие классы. Это, в свою очередь, улучшает модифицируемость: систему стало проще изменять и уменьшился риск появления ошибок, вызванных изменениями.

### Замена пользовательского кода библиотеками или фреймворками от сторонних производителей

Третьей причиной, обычно приводящей к образованию тесной связи между модулями, являются классы с общей или служебной функциональностью. Классическим примером могут служить классы с именами `StringUtils` и `FileUtils`. Поскольку эти классы обеспечивают общие функциональные возможности, они используются повсеместно в базе кода. Часто такой тесной связи сложно избежать. Тем не менее старайтесь ограничивать размеры классов и периодически пересматривать библиотеки и фреймворки (с открытым исходным кодом) на предмет замены ими пользовательской реализации. CommonLibrary.NET (https://commonlibrarynet.codeplex.com/) — популярная библиотека часто востребованных функций. В некоторых случаях служебный код можно заменить

новыми возможностями языка C# или кодом из общей библиотеки компании.

## 6.3. Типичные возражения против разделения задач

Ниже рассматриваются типичные возражения против рекомендации, которой посвящена глава.

### Возражение: слабые связи вступают в конфликт с возможностью повторного использования

*«Тесная связь — это побочный эффект многократного использования кода, получается, что эта рекомендация противоречит другим*».

Да, многократное использование кода может увеличить число вызовов метода. Однако это не должно приводить к образованию тесных связей по двум причинам:

- многократное использование кода не обязательно должно сопровождаться вызовом методов из множества мест в коде. Улучшение архитектуры программного обеспечения, например с помощью наследования или сокрытия реализации за интерфейсами, позволяет использовать код многократно без образования тесных связей, поскольку интерфейс скрывает детали реализации;
- придание коду более обобщенного характера для расширения круга решаемых им задач не означает образования тесных связей. Действительно, обобщенный код будет вызываться из большего количества мест, чем конкретный. Но для реализации служебных функций часто требуется небольшой объем кода. В результате может иметься много входящих зависимостей, но все они будут ссылаться на код небольшого размера.

### Возражение: интерфейсы языка C# не предназначены для ослабления связей

*«Не имеет смысла использовать интерфейсы языка C# для исключения тесных связей».*

Действительно, использование интерфейсов дает отличную возможность улучшить инкапсуляцию путем сокрытия реализации, но нет смысла определять по интерфейсу для каждого класса. Определять интерфейс имеет смысл, только если он реализуется не менее чем двумя классами в приложении. Рассмотрите возможность разделения класса, если единственной причиной определения интерфейса для него является ограничение объема кода, доступного другим классам.

### Возражение: высокая нагрузка на служебные классы неизбежна

*«Служебный код всегда будет вызываться из множества мест в системе».*

Это действительно так. На практике даже системы с высоким уровнем обслуживаемости содержат небольшое количество кода, который является настолько универсальным, что используется из множества мест (например, функции регистрации или ввода/вывода). Универсальный, многократно используемый код должен быть небольшим, но без некоторой его части просто нельзя обойтись. Однако если функция действительно широко используется в системе, должны существовать реализующие ее фреймворки или библиотеки, позволяющие использовать ее в готовом виде.

### Возражение: не все слабо связанные решения улучшают обслуживаемость

*«Фреймворки, реализующие инверсию управления (IoC), обеспечивают слабую связанность, но при этом затрудняют обслуживание кода».*

Инверсия управления (Inversion of Control, IoC) — это принцип проектирования, помогающий добиться ослабления связей. Существуют фреймворки, способные помочь в этом. Применение принципа IoC сделает систему более гибкой для расширения и уменьшит количество связей между фрагментами кода.

Утверждение, что такие фреймворки затрудняют обслуживание, свидетельствует о недостаточной опытности разработчиков. Следовательно, когда такое утверждение оказывается обоснованным,

это объясняется проблемой выбранного фреймворка, а не технологии IoC.

К решению использовать фреймворк для реализации технологии IoC следует подходить с осторожностью. Подобно всем инженерным решениям, оно представляет собой компромисс, который не всегда оправдан. Использование фреймворка такого типа только для достижения слабой связи практически никогда не бывает оправдано.

### Как в компании SIG устанавливается рейтинг связанности модулей

Связь между модулями является одним из восьми критериев SIG/TÜViT оценки системных свойств для достижения уверенной обслуживаемости продукта. При определении рейтинга по степени связанности модулей рассчитывается нагрузка на каждый метод. Каждый модуль (класс в языке C#) затем помещается в одну из четырех категорий риска, в зависимости от итоговой нагрузки на все методы класса. В табл. 6.1 перечислены эти четыре категории риска в соответствии с версией 2015 критериев оценки SIG/TÜViT. Таблица отражает максимальный объем кода, который может находиться в категориях риска для системы с рейтингом в 4 звезды. Например, не более 21,8% кода может находиться в категории умеренного риска, и то же самое относится к другим категориям.

***Таблица 6.1 ❖ Категории риска по связям модулей (версия 2015 критериев оценки SIG/TÜViT)***

| Нагрузка на вход в категории | Допустимый процент в системе с 4 звездами |
|---|---|
| 51 + | Не более 6,6% |
| 21—50 | Не более 13,8% |
| 11—20 | Не более 21,6% |
| 1—10 | Не содержит |

На рис. 6.1 изображены примеры профилей качества трех систем:

- слева — диаграмма для системы с открытым исходным кодом, в данном случае Jenkins. Следует отметить, что здесь Jenkins не полностью соответствует требованиям к системе с рейтингом в 4 звезды из-за большого объема кода в категории с высоким риском (красного цвета);
- в центре — диаграмма для анонимной системы, используемой SIG в качестве эталона 4-звездной системы;
- справа — диаграмма, отображающая граничные точки характеристик для достижения качества, оцениваемого в 4 звезды.

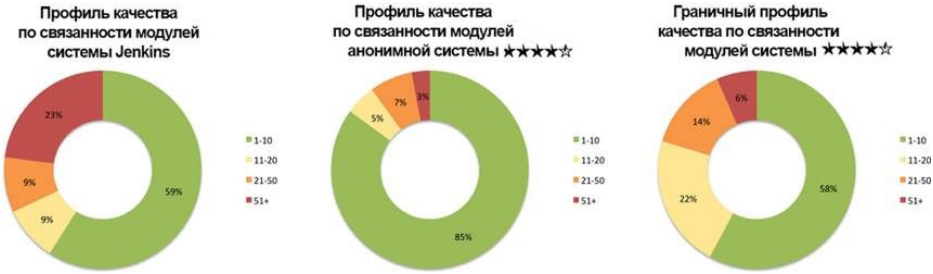


**Рис. 6.1** ❖ Три профиля качества по связанности модулей

# Глава 7

# Избегайте тесных связей между элементами архитектуры

Существуют два подхода к разработке программного обеспечения: один из них заключается в его упрощении, пока не станет очевидным отсутствие недостатков, а другой — в усложнении, пока его недостатки не станут неочевидными.

*C. A. P. Хоар (C. A. R. Hoare)*

Рекомендация:
- **добивайтесь слабой связи между компонентами верхнего уровня**;
- для этого **минимизируйте относительный объем экспортируемого кода, доступного для использования в модулях других компонентов**;
- это улучшает обслуживаемость, поскольку **независимые компоненты проще в обслуживании**.

Четкое представление об архитектуре программного обеспечения очень важно для создания и обслуживания программ. Ясная архитектура позволяет понять, что делает система, как она это делает и как она организована (то есть как сгруппированы ее компоненты). Она отражает высокоуровневую структуру системы, так сказать, ее «костяк». Хорошая архитектура облегчает поиск нужного исходного кода и дает возможность разобраться во взаимодействиях компонентов (высокого уровня).

Эта глава посвящена зависимостям на уровне компонентов. Компоненты являются элементами самого верхнего уровня системы. Они определяют программную архитектуру системы, по-

этому их границы должны быть четко очерчены в самом начале процесса разработки. Поскольку на архитектуру программного обеспечения может оказывать влияние предметная область, иногда нет возможности напрямую управлять ею. Однако за реализацию архитектуры программного обеспечения всегда отвечает разработчик.

Компоненты должны быть слабо связанными, то есть они должны быть надежно разделены за счет ограничения количества точек доступа и объема информации, используемой компонентами совместно. В этом случае детали реализации их методов будут скрыты (или инкапсулированы), что улучшит модульность системы.

Звучит знакомо? Да, это все тот же общий принцип проектирования, направленный на ослабление связей между модулями, рассматривавшийся в главе 6. К связям между компонентами применимы те же рассуждения, но уже на более высоком уровне. Связанность модулей определяется объемом функций, экспортируемых отдельными модулями (классами) остальному коду. Связанность компонентов определяется количеством модулей, экспортируемых одним компонентом (группой модулей) другому.


Если модуль вызывается другим модулем в том же компоненте, такой вызов считается внутренним на уровне компонентов, но на уровне модулей он влияет на *связанность модулей.*

В этой главе мы будем называть слабо связанные компоненты независимыми. Противоположностью независимых компонентов являются зависимые компоненты, которые в значительной степени зависят от особенностей работы других компонентов. Взаимосвязанность такого рода затрудняет контроль влияния изменений в одном компоненте на другие компоненты. Это усложняет тестирование, поскольку требует предвидения или моделирования воздействия на другие компоненты.

## 7.1. Мотивация

Обслуживание системы упрощается, когда изменения в компоненте не оказывают влияния за его пределами. Для демонстрации пре-

имуществ слабо связанных компонентов рассмотрим различные виды зависимости, изображенные на рис. 7.1.


**Рис. 7.1** ❖ Слабо связанный компонент (слева) и тесно связанный компонент (справа)

Слева показан слабо связанный компонент. Многие вызовы между его модулями являются внутренними (в рамках компонента). Остановимся более подробно на внутренних и внешних зависимостях.

Вызовы, улучшающие обслуживаемость:

- внутренние вызовы приветствуются. Поскольку модули, вызывающие друг друга, являются частью одного компонента, они должны реализовывать тесно связанные функции. Их внутренняя логика не видна снаружи;
- исходящие вызовы также приветствуются. Делегирование задач другим компонентам создает внешние зависимости. В целом делегирование различных задач другим компонентам не вызывает возражений. Делегирование может производиться из любого компонента, и нет смысла ограничиваться только модулями компонента.


Обратите внимание, что вызовы, исходящие для одного компонента, являются входящими для другого.

Вызовы, оказывающие негативное влияние на обслуживаемость:

- входящие вызовы экспортируют функциональные возможности другим компонентам через интерфейсы. Участвующий в этом объем кода должен быть небольшим. И наоборот, код компонента должен быть максимально инкапсулирован, то есть он не должен напрямую вызываться из других компонентов. Это позволяет скрыть информацию о деталях реализации. Кроме того, изменение кода, включенного во входящие зависимости, потенциально оказывает влияние на другие компоненты. Включение во входящие зависимости небольшого процента кода ослабляет негативное влияние изменений на другие компоненты;
- транзитный код является источником повышенного риска, и его следует избегать. Транзитный код получает входящие вызовы и делегирует их другим компонентам. Транзитный код выполняет действие, обратное сокрытию подробностей реализации, предоставляя клиентам доступ к делегатам (их реализации). Он подобен службе поддержки, которая не в состоянии ответить па поставленный вопрос н переадресует клиента к службе другой компании. Теперь успешное получение ответа зависит от работы уже двух служб. Для кода это означает, что обязанности плохо распределены между компонентами. Это затрудняет отслеживание путей прохождения запросов, а также тестирование и внесение изменений, поскольку тесные связи способствуют их отражению на других компонентах.

Справа на рис. 7.1 показан компонент с высоким уровнем зависимостей. От компонента зависит множество модулей других компонентов, из-за чего он оказывается тесно связанным. Вносимые в него изменения трудно изолировать, поскольку сложно контролировать последствия таких изменений.

Обратите внимание, что независимость распространяется на *баланс компонентов*. Баланс компонентов достигается при сбалансированности количества и относительного размера компонентов. Подробнее об этом рассказывается в главе 8.

Для демонстрации развития связей между компонентами с течением времени рассмотрим появление переплетений в системе, кажущееся вполне естественным. Переплетения связей появляются

из-за спешки при внесении изменений в код, низкой дисциплины разработки или по другим причинам, препятствующим последовательному применению выбранной архитектуры. Рисунок 7.2 иллюстрирует ситуацию, с которой часто приходится сталкиваться на практике. Изначально система имеет четкую архитектуру с однонаправленными зависимостями, но со временем они запутываются и переплетаются. В данном случае показано переплетение между слоями, но подобные ситуации характерны и для компонентов.


**Рис. 7.2** ❖ Задуманная и реализованная архитектура

### Слабая зависимость между компонентами обеспечивает изолированность его обслуживания

Слабая зависимость подразумевает возможность изолированного внесения изменений. Это возможно, когда подавляющее большинство составляют внутренние, или исходящие, вызовы. Изолированное обслуживание уменьшает объем работы, поскольку изменения в коде не влияют ни на что, находящееся за его пределами.

Обратите внимание, что рассуждения об изоляции применимы на более низком уровне. Например, система, состоящая из небольших, простых классов, предполагает правильное разделение задач, но не гарантирует его. Для уверенности в этом следует проанализировать сами зависимости (о чем рассказывалось в главе 6).

### Слабая зависимость компонентов способствует разделению ответственности за обслуживание

Если все компоненты независимы друг от друга, это упрощает распределение ответственности за их обслуживание между отдельными командами. Это следует из возможности изолированного внесения изменений. Изоляция является необходимым условием эффективного разделения работы между членами одной команды или между разными командами разработчиков.

Напротив, если компоненты тесно переплетены, их нельзя изолировать и разделить обязанности по обслуживанию между командами, поскольку последствия внесения изменений повлияют на работу других команд. Это, помимо усложнения процесса тестирования, может привести к непредсказуемым последствиям. То есть зависимости способны привести к противоречиям, увеличить время на согласование действий разработчиков и время, впустую потраченное одними разработчиками на ожидание завершения внесения изменений другими.

### Слабая зависимость компонентов упрощает тестирование

Код со слабой зависимостью от других компонентов (модули содержат в основном внутренние и исходящие вызовы) проще тестировать. Внутренние вызовы можно исследовать и протестировать в пределах компонента. Для тестирования исходящих вызовов можно не создавать фиктивных функций или функций-заглушек, поскольку все необходимое предоставляется другими компонентами (при условии, что работа над этими компонентами завершена).

Более подробно модульное тестирование рассматривается в главе 10.

## 7.2. Как применять рекомендацию

Рекомендация, представленная в этой главе, направлена на обеспечение слабой связанности между компонентами. На практике достичь этого поможет соблюдение следующих правил при реализации интерфейсов и запросов между компонентами:

- ограничивайте размеры модулей, составляющих интерфейс компонента;
- определяйте максимально абстрактные интерфейсы компонентов. Это ограничит число видов запросов, пересекающих границы компонента, что позволит избежать запросов, которые «слишком много знают» о деталях реализации;
- избегайте транзитного кода — он серьезно затрудняет тестирование. Другими словами, избегайте интерфейсных модулей, передающих вызовы другим компонентам. Если транзитный код уже имеется, проанализируйте соответствующие модули и организуйте прямой вызов других компонентов.

### Абстрактная фабрика как шаблон проектирования

Независимость компонентов отражает архитектуру программной системы. Но эта книга не об архитектуре программного обеспечения. В данном разделе рассматривается только один шаблон проектирования, который мы часто применяем на практике для ограничения объема интерфейсного кода, экспортируемого компонентом: абстрактная фабрика. Слабо связанная система в основном ориентирована на договоренности, а не на детали реализации.

Многие шаблоны проектирования и архитектурные стили способны помочь сохранить слабые связи между компонентами. Примером может служить применение фреймворка внедрения зависимостей (реализующего принцип инверсии управления). Более подробную информацию о других шаблонах можно найти в книгах, посвященных шаблонам проектирования и архитектуре программного обеспечения (см. список «Рекомендуемые книги»).

Суть шаблона проектирования «Абстрактная фабрика» заключается в сокрытии (или инкапсуляции) конкретного «продукта» за универсальным интерфейсом «фабрики продуктов». В данном контексте под продуктами понимаются сущности, которые могут присутствовать в нескольких формах. Примерами могут служить алгоритмы кодирования/декодирования аудиофайлов различных форматов или виджеты пользовательского интерфейса, поддерживающие темы оформления внешнего вида. В следующем примере шаблон «Абстрактная фабрика» используется для сокрытия осо-

бенностей платформ облачного хостинга за небольшим фабричным интерфейсом.

Предположим, что в системе имеется компонент PlatformServices, реализующий управление службами облачного хостинга. Компонентом PlatformServices поддерживаются два конкретных облачных провайдера: Amazon AWS и Microsoft Azure (в дальнейшем планируется добавить еще несколько).

Для запуска и остановки серверов и резервирования места под хранилище необходимо реализовать следующий интерфейс доступа к платформе облачного хостинга:

```
public interface ICloudServerFactory
{
    ICloudServer LaunchComputeServer();

    ICloudServer LaunchDatabaseServer();

    ICloudStorage CreateCloudStorage(long sizeGb);
}
```

На основании этого интерфейса создадим два конкретных фабричных класса: для AWS и Azure:

```
public class AWSCloudServerFactory : ICloudServerFactory
{
    public ICloudServer LaunchComputeServer()
    {
        return new AWSComputeServer();
    }

    public ICloudServer LaunchDatabaseServer()
    {
        return new AWSDatabaseServer();
    }

    public ICloudStorage CreateCloudStorage(long sizeGb)
    {
        return new AWSCloudStorage(sizeGb);
    }
}

public class AzureCloudServerFactory : ICloudServerFactory {
    public ICloudServer LaunchComputeServer() {
        return new AzureComputeServer();
    }
```

```
    public ICloudServer LaunchDatabaseServer() {
        return new AzureDatabaseServer();
    }

    public ICloudStorage CreateCloudStorage(long sizeGb) {
        return new AzureCloudStorage(sizeGb);
    }
}
```

Обратите внимание, что фабрики вызывают конкретные классы для AWS и Azure (которые, в свою очередь, обращаются к прикладному программному интерфейсу AWS и Azure), но возвращают универсальные интерфейсы для доступа к серверам и хранилищам.

Теперь код за пределами компонента `PlatformServices` может использовать лаконичный интерфейс `ICloudServerFactory`, как показано ниже:

```
public class ApplicationLauncher
{

    public static void Main(string[] args)
    {
        ICloudServerFactory factory;
        if (args[1].Equals("-azure"))
        {
            factory = new AzureCloudServerFactory();
        }
        else
        {
            factory = new AWSCloudServerFactory();
        }
        ICloudServer computeServer = factory.LaunchComputeServer();
        ICloudServer databaseServer = factory.LaunchDatabaseServer();
```

Интерфейс `ICloudServerFactory` компонента `PlatformServices` экспортирует небольшое число методов для использования другими компонентами системы, благодаря чему сохраняется слабая связь других компонентов с этим.

## 7.3. Типичные возражения против устранения тесных связей компонентов

В этом разделе рассматриваются возражения против уменьшения зависимости между компонентами из-за сложности исправления самих компонентов или специальных требований системы.

### Возражение: зависимости между компонентами невозможно смягчить из-за тесного переплетения компонентов

*«Мы не можем устранить зависимости между компонентами из-за их тесной взаимосвязанности».*

Переплетенные между собой компоненты превращаются в проблему при обслуживании. Для начала следует проанализировать модули на наличие транзитного кода, который затрудняет тестирование и исследование особенностей функционирования.

Четкое разделение ответственности компонентов упрощает анализ и тестирование модулей, находящихся внутри. Например, чрезмерное число входящих вызовов указывает, что модуль имеет слишком много обязанностей и его можно разделить. Разделение упрощает анализ и тестирование. Более подробные сведения об этом приводятся в главе 6.

### Возражение: нет времени на исправление

*«Обслуживающая команда понимает важность слабых зависимостей между компонентами, но не хватает времени, чтобы исправить это».*

Мы с пониманием относимся к этой проблеме. Сроки разработки должны выдерживаться, времени на модернизацию кода может не хватить, или руководитель может счесть это «технической эстетикой». В таком случае важно принять компромиссное решение. Несомненно, нужно исключить проблемы, которые реально затрудняют обслуживание. То есть проблема зависимостей должна решаться, когда команда обнаружит, что она отрицательно влияет на тестирование, возможность анализа или стабильность. Это решение можно подкрепить оценкой степени усложнения обслуживания тесно связанных компонентов.

Например, транзитный код выполняет сложные переходы, с трудом поддающиеся тестированию. Возможно, вы найдете более элегантное решение, требующее меньше времени и усилий.

### Возражение: транзитный код просто необходим

*«Сама архитектура нашего программного обеспечения требует наличия слоя транзитных вызовов».*

Действительно, некоторым архитектурам необходим промежуточный слой. Как правило, это слой служб, принимающих запросы от одной стороны (например, пользовательского интерфейса) и объединяющий их для передачи другому слою в системе. Наличие такого слоя не всегда вызывает проблемы, при условии что он характеризуется слабыми связями. Он должен четко разделять входящие и исходящие запросы. То есть модуль, принимающий запросы:

- не обрабатывает запросы сам;
- не знает, где и как обрабатываются эти запросы (подробности реализации).

Если выполняются оба условия, тогда модуль в слое служб можно рассматривать как имеющий входящие и исходящие вызовы, а не как транзитный модуль, передающий запросы принимающим компонентам.

Слой служб большого размера, содержащий массу логики, должен вызывать подозрение. В этом случае слой не просто передает запросы, а преобразует их. То есть для их трансформации слой обязан знать детали реализации. Это означает, что слой не инкапсулирует запросы и реализацию должным образом. Если архитектура программного обеспечения требует наличия транзитного кода, имеет смысл озвучить проблему архитектору программного обеспечения.

## 7.4. Дополнительные сведения

Идея независимости компонентов тесно связана с балансом компонентов, как описывается в главе 8. Эта глава расскажет, как обеспечить разумное количество компонентов, близких по размерам.

### Как в компании SIG оценивается независимость компонентов

Силу связей между компонентами в компании SIG определяют и измеряют как «независимость компонентов». Независимость оценивается на уровне модулей, так как каждый модуль в системе должен располагаться в компоненте. Здесь «модуль» является минимальной группой блоков кода, как правило, это файл.

Оценить зависимость между модулями можно, подсчитав вызовы между ними (статический анализ исходного кода). При классификации зависимостей между компонентами они разделяются на скрытый и интерфейсный код.

- Скрытый код состоит из модулей, не имеющих входящих зависимостей от модулей других компонентов, вызовы в них поступают только из собственного компонента (внутренние), но сами они могут выполнять вызовы за пределы своего компонента (исходящие).
- Интерфейсный код состоит из модулей, имеющих входящие зависимости из модулей других компонентов. Он включает код, принимающий входящие и выполняющий транзитные вызовы.

Согласно принципу слабой связанности, низкий уровень зависимости между модулями лучше, чем высокий. Последний указывает на высокую вероятность, что изменения в одном компоненте повлияют на другие.

В компании SIG независимость компонентов оценивается процентом скрытого кода. Для достижения рейтинга в 4 звезды по SIG/TÜViT, свидетельствующего о высоком уровне обслуживаемости программного обеспечения, процент кода, находящегося в модулях с входящими зависимостями (входящий, или транзитный, код), не должен превышать 14,2%.

На рис. 7.3 изображены примеры профилей качества трех систем:

- слева — диаграмма для системы с открытым исходным кодом, в данном случае Jenkins;
- в центре — диаграмма для анонимной системы, используемой SIG в качестве эталона 4-звездной системы;
- справа — диаграмма, отображающая граничные точки характеристик для достижения качества, оцениваемого в 4 звезды.


**Рис. 7.3** ❖ Три профиля качества по независимости компонентов

# Глава 8

# Стремитесь к сбалансированности архитектуры компонентов

Определение границ инкапсуляции является одним из важнейших навыков в проектировании архитектуры программного обеспечения.

*Джордж Х. Фэрбенкс (George H. Fairbanks), «Just Enough Architecture»*

Рекомендация:
- **сбалансируйте число и размеры компонентов верхнего уровня**;
- для этого организуйте исходный код так, чтобы **число компонентов было близко к 9 (то есть от 6 до 12) и все они имели примерно одинаковый размер**;
- это улучшит обслуживаемость, потому что **в сбалансированных компонентах проще искать код и они обеспечивают изолированное обслуживание**.

Хорошо сбалансированная архитектура программного обеспечения характеризуется не слишком большим и не слишком малым количеством компонентов примерно равных размеров. Такая архитектура считается хорошо *сбалансированной на уровне компонентов.*

Примером *дисбаланса на уровне компонентов* является наличие нескольких очень крупных компонентов с непропорционально большим объемом логики и массы маленьких, практически незаметных.

Рисунок 8.1 иллюстрирует понятие сбалансированности компонентов, в том числе идеальной. Наименее желательный случаи показан сверху слева — здесь имеется единственный компонент, не

позволяющий разным разработчикам вносить изменения изолированно. Идеальный случай показан внизу справа — здесь имеются девять компонентов с ограниченными областями влияния, обслуживание которых может производиться изолированно. Во втором случае (вверху справа) система страдает от неравномерного распределения кода между компонентами. Когда один из компонентов имеет очень большой размер, архитектура становится монолитной, что затрудняет навигацию по коду и изолированное его обслуживание. В третьем случае (внизу слева) архитектура разбита на множество компонентов, из-за чего трудно составить мысленное представление о системе и разобраться во взаимодействиях компонентов.


**Рис. 8.1** ❖ Деление системы на компоненты:
с худшим балансом вверху слева и с лучшим — внизу справа

## 8.1. Мотивация

Теперь понятно, что такое баланс компонентов, но мы пока не знаем, почему так важно его добиваться. А причины просты: сбалансированная архитектура программного обеспечения упрощает его обслуживание. В этом разделе рассказывается о выгодах, которые дает хорошая сбалансированность компонентов: упрощение поиска и анализа кода, а также лучшая изолированность и разделение ответственности, облегчающие обслуживание.

### Хороший баланс компонентов упрощает поиск и анализ кода

Четкая организация компонентов упрощает поиск фрагмента кода, который требуется изменить. Конечно, немаловажную роль в этом играют также содержание кода в чистоте и использование понятных имен (подробнее об этом в главе 11). Когда число компонентов невелико (около 9) и их размеры сопоставимы, это упрощает поиск нужного фрагмента кода, его анализ и изменение.

Напротив, компоненты с несбалансированной организацией почти наверняка имеют нечеткие границы ответственности. Например, если какой-то компонент значительно больше других, он явно содержит не связанные между собой функции, и его труднее анализировать.

### Хорошо сбалансированные компоненты улучшают изолированность обслуживания

В системе со сбалансированными компонентами зоны ответственности четко разграничены, а надлежащее *разделение задач* обеспечивает их изолированность. Изолированность компонентов имеет большое значение, поскольку защищает от неожиданных эффектов, таких как регрессия.

В более широком смысле изоляция кода внутри компонентов обеспечивает такое явное преимущество, как модульность. Компоненты, надежно разделенные функционально и технически, легче заменять, удалять и тестировать, чем компоненты со смешанными функциями и техническими переплетениями.

Имейте в виду, что хороший баланс компонентов сам по себе не гарантирует возможности вносить изменения изолированно. В конце концов, из размещения кода в нескольких компонентах не следует, что эти компоненты независимы друг от друга. Степень взаимозависимости компонентов рассматривалась в главе 7.

Принцип изоляции кода применим и на более низком уровне. Например, система, состоящая из небольших простых классов, подразумевает разделение задач, но не гарантирует его. Чтобы убедиться в этом, потребуется исследовать имеющиеся зависимости (как это было сделано в главе 6).

### Хорошо сбалансированные компоненты позволяют распределять ответственность при обслуживании

Наличие четких функциональных границ между компонентами облегчает процесс распределения ответственности за обслуживание между командами разработчиков. Число компонентов системы и их относительный размер должны соответствовать разделению системы по функциональным признакам.

Когда система содержит слишком много или слишком мало компонентов, в ней сложно разобраться, и еще труднее ее обслуживать. Если число компонентов слишком мало, это затрудняет навигацию по функциональным областям системы. С другой стороны, слишком большое количество компонентов затрудняет обзор всей системы в целом.

## 8.2. Как применять рекомендацию

В основе балансировки компонентов лежат два принципа:

- в идеале количество компонентов верхнего уровня должно быть равно 9, допустимым считается диапазон от 6 до 12;
- объем исходного кода в компонентах должен быть примерно одинаков.

Имейте в виду, что баланс компонентов является лишь показателем четкого разделения на компоненты, а не самоцелью. Он должен обеспечиваться процессом проектирования и разработки системы. Разделение системы на компоненты должно быть естественным, нет смысла добиваться наличия ровно девяти компонентов любой ценой, только чтобы их было именно девять.

### Выбор правильного концептуального уровня при распределении функциональных возможностей по компонентам

Для достижения хорошего разделения системы, упрощающего ориентацию разработчиков, важно правильно выбрать концептуальный уровень распределения функциональных возможностей. Обычно программные системы ориентируются на высокоуровневые функции предметной области. С другой стороны, разделение может основываться на технических особенностях.

Например, система, разделенная па компоненты по функциям предметной области, может включать компоненты, отвечающие за получение данных, управление счетами, администрирование и т. д. Каждый компонент содержит код, реализующий одну функцию от начала до конца, от базы данных до веб-интерфейса. Преимущество функционального разделения — в том, что подготавливается на этапе проектирования, еще до начала разработки. В этом случае разработчики получают возможность анализировать исходный код, оперируя высокоуровневыми понятиями. Однако этот вид разделения имеет свой недостаток: разработчики должны быть опытными и хорошо разбираться в различных технических областях, чтобы успешно вносить изменения даже в один-единственный компонент.

Система с техническим разделением может содержать такие компоненты, как клиентская часть, серверная часть, интерфейсы, регистрация и т. д. Этот подход позволяет разделить обязанности среди команд разработчиков на основании их технологической специализации. Здесь разделение на компоненты отражает разделение труда между различными специалистами.

За выбор правильной концепции разделения функций в системе отвечает архитектура программного обеспечения. Этим может заниматься как один человек, так и несколько людей в команде разработчиков. Когда потребуется внести изменения в разделение компонентов, необходимо проконсультироваться с теми, кто занимается архитектурой.

### Последовательное применение разделения системы на основе анализа предметной области

После выбора вида разделения системы на компоненты необходимо последовательно придерживаться его. Несогласованная архитектура не может быть хорошей архитектурой. Поэтому разделение на компоненты следует официально закрепить и контролировать. В то время как при проектировании выбор возлагается на архитектора, дисциплина создания компонентов и их границы должны соблюдаться всеми разработчиками. Этого можно достичь путем согласования изменений в компонентах. Поскольку здесь требуется коллективная ответственность, все действия должны выполняться согласованно.

## 8.3. Типичные возражения против стремления к сбалансированности компонентов

В этом разделе рассматриваются возражения против необходимости достижения баланса компонентов. Обычно возражающие считают, что дисбаланс компонентов на самом деле не является проблемой или что эту проблему невозможно исправить.

### Возражение: системы с дисбалансом компонентов отлично работают

*«Наша система имеет плохой баланс компонентов, но у нас нет никаких проблем в связи с этим».*

Баланс компонентов не является двухвариантным понятием — существуют разные степени сбалансированности, отличающиеся от «идеала» с девятью компонентами одинакового размера. Затрудняет ли дисбаланс обслуживание, во многом зависит от величины отклонения, опыта обслуживающей команды и причин дисбаланса.

Наибольший ущерб обслуживаемости наносит дисбаланс, вызванный отсутствием дисциплины обслуживания, когда разработчики помещают код не в те компоненты. Поскольку несоответствие является врагом предсказуемости, оно может привести к неожиданным последствиям. Код, помещенный в не предназначенный для

него компонент, создает случайные зависимости между компонентами, которые наносят серьезный ущерб тестируемости и гибкости.

### Возражение: запутанность связей между компонентами не позволяет их сбалансировать

*«Мы не можем добиться баланса компонентов из-за запутанности связей между ними».*

Эта ситуация указывает на другую проблему, заключающуюся в технической зависимости между компонентами. Переплетение связей между компонентами указывает на неправильное разделение задач. Эта проблема и соответствующая ей рекомендация рассматривалась в главе 7. В этом случае более важным и срочным является наведение порядка в зависимостях компонентов, например путем сокрытия деталей реализации с помощью интерфейсов и исключения циклических зависимостей. После этого можно заняться структурой компонентов для достижения сбалансированности.

## 8.4. Дополнительные сведения

Связи между компонентами имеют непосредственное отношение к рассматриваемой в этой главе идее баланса компонентов. Связям между компонентами была посвящена глава 7.

---

### Как в компании SIG оценивается баланс компонентов

В компании SIG баланс компонентов определяется и измеряется с помощью комбинирования (то есть перемножения) следующих характеристик:

- число компонентов верхнего уровня в системе;
- примерно одинаковый размер компонентов.

В компании SIG идеальным числом компонентов верхнего уровня в системе считается число девять, которое было выбрано на основе многолетнего опыта. Чем ближе число компонентов к девяти, тем лучше. Числовая оценка количества компонентов верхнего уровня в системе варьируется от 0 до 1. Системам с девятью компонентами присваивается оценка 1 и линейно уменьшается до 0 для систем с одним компонентом. При увеличении числа компонентов применяется коррекция, обеспечивающая более гибкое оценивание систем

с количеством компонентов больше 17. При использовании линейной модели оценка в этом случае была бы равна 0. Коррекция основана на 95-процентной оценке в рамках базового показателя.
Близость размеров компонентов подразумевает распределение исходного кода между компонентами. Предпочтение отдается компонентам верхнего уровня с одинаковыми размерами.
В компании SIG для измерения равенства размеров компонентов используется скорректированный коэффициент Джини (Gini). Коэффициент Джини предназначен для измерения неравенства сущностей и варьируется от 0 (идеальное равенство) до 1 (абсолютное неравенство).
Для достижения рейтинга в 4 звезды по SIG/TÜViT, свидетельствующего о высоком уровне обслуживаемости программного обеспечения, количество компонентов должно быть близко к девяти, а скорректированный коэффициент Джини для размеров компонентов не должен превышать 0,71.

На рис. 8.2 изображены примеры профилей качества трех систем:

- первая диаграмма соответствует анонимной системе, используемой в SIG в качестве эталона 2-звездной системы. Обратите внимание на количество компонентов (шесть) и объемы размещенного в них кода;
- вторая диаграмма соответствует анонимной системе, используемой в SIG в качестве эталона 4-звездной системы. Обратите внимание, что, несмотря на неравный объем кода в компонентах, число компонентов ровно именно девяти, что перекрывает влияние неодинаковости размеров компонентов.

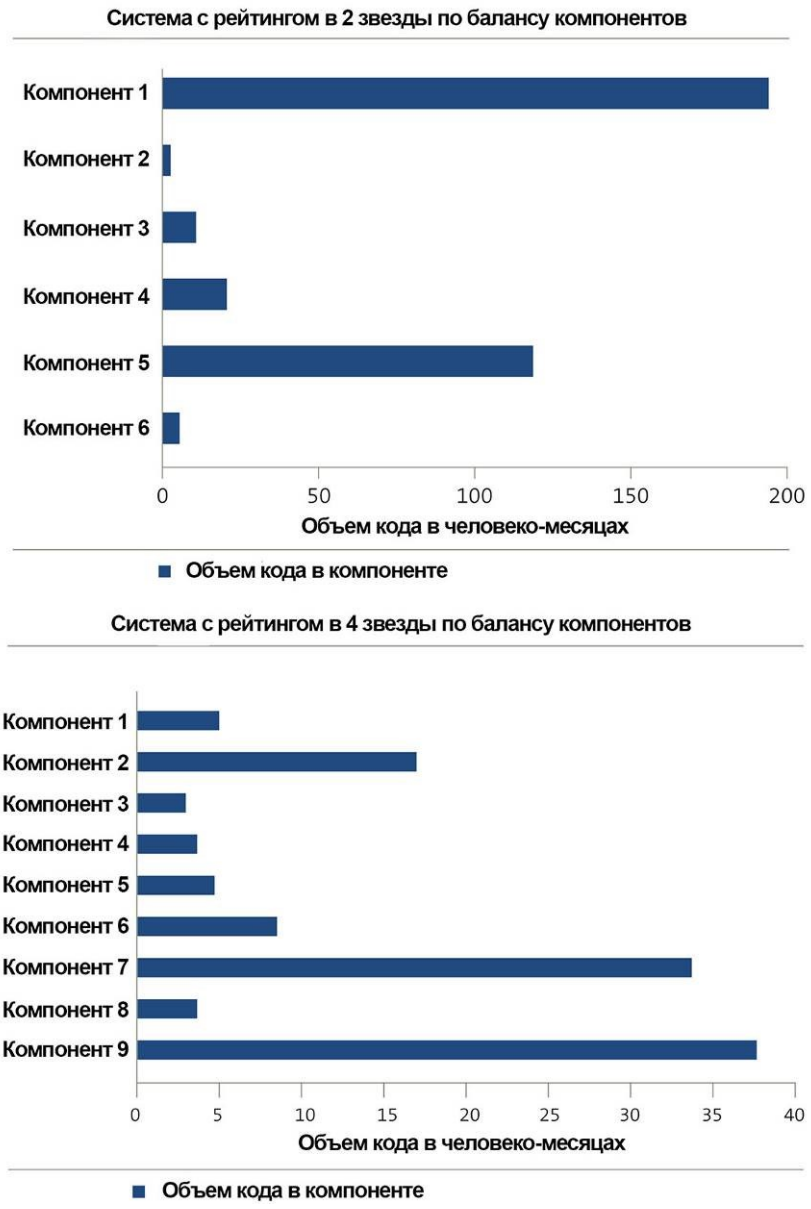


**Рис. 8.2** ❖ Два профиля качества
по сбалансированности компонентов

# Глава 9

# Следите за размером базы кода

Сложность программы растет до тех пор,
пока она не превысит возможности программиста,
который должен ее поддерживать.

*7-й закон компьютерного программирования*

Рекомендация:
• **сократите объем кода настолько, насколько это возможно**;
• для этого **предотвращайте разрастание кода и прилагайте активные усилия по снижению размера системы**;
• это улучшит обслуживаемость, поскольку **небольшой размер продукта, проекта и команды является важным фактором успеха**.

База кода — это коллекция исходного кода, который размещен в одном хранилище и может компилироваться, развертываться и обслуживаться одной командой. Любая система содержит, по крайней мере, одну базу кода. Некоторые крупные системы имеют несколько баз кода. Типичным примером является пакетное программное обеспечение. В нем может иметься база кода, реализующая стандартную функциональность, и несколько дополнительных баз кода с пользовательскими или сторонними плагинами, которые обслуживаются независимо.

При выборе из двух систем с одинаковыми возможностями, одна из которых имеет небольшую, а вторая — большую базу кода, предпочтение обычно отдается системе с меньшей базой кода. В системе малого размера проще искать и анализировать код. При внесении изменений в ней проще определить, как это скажется на других частях системы. Такая простота обслужива-

ния приводит к меньшему количеству ошибок и снижает затраты. Это же очевидно.

## 9.1. Мотивация

Разработка и обслуживание программного обеспечения значительно усложняются с ростом его размеров. Для создания больших систем требуются многочисленная команда разработчиков и длительное время, что связано с дополнительными издержками и риском провала (всего проекта). В остальной части этого раздела мы рассмотрим отрицательные качества больших программных систем.

### Проект с большой базой кода, скорее всего, обречен на неудачу

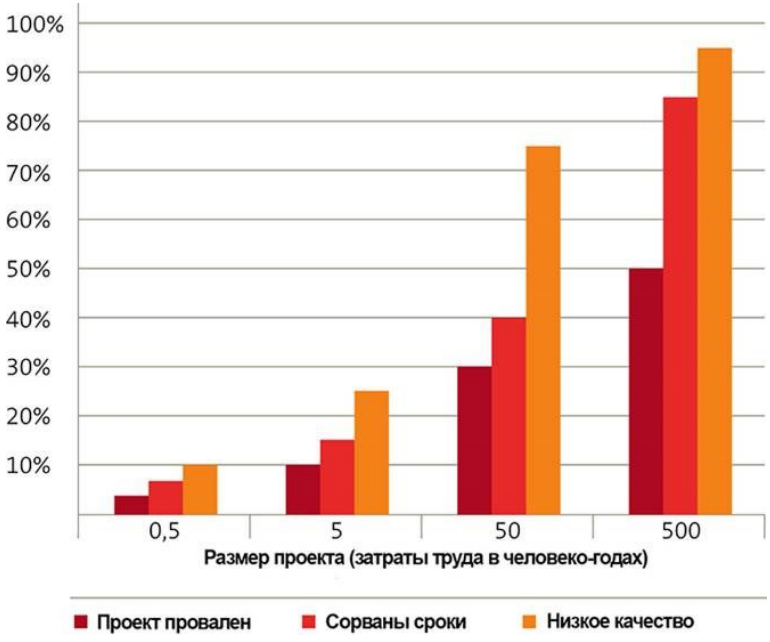


**Рис. 9.1** ❖ Связь неудач проектов с их размерами[1]

[1] Источник: *Jones C., Bonsignour O.* The Economics of Software Quality. Addison-Wesley Professional, 2012. Оригинальные данные переведены в человеко-года (200 функциональных единиц в год для языка Java).

Существует тесная связь между размером проекта и его рискованностью. Крупный проект требует привлечения большей команды разработчиков, комплексного проектирования и длительной разработки. В результате затрудняются взаимодействие и координация между заказчиками и членами команды, усложняется проект программного обеспечения и в течение разработки в требования вносится большее количество изменений. Все это увеличивает риск снижения качества проекта, срыва его сроков и даже провала проекта. Диаграмма на рис. 9.1 представляет накопительные вероятности. Например, из всех проектов с затратами труда на разработку, превышающими 500 человеко-лет, более 90% характеризуются как «низкокачественные». В их число входят проекты с превышением сроков разработки (80-90% от общего числа) и закрытые проекты (50% от общего числа).

Рисунок 9.1 демонстрирует взаимосвязь между размерами проектов и риском их провала, из него ясно видно, что по мере увеличения размера проекта растет вероятность провала (например, проект закрывается или заканчивается ничем), срыва сроков проекта и низкого качества проекта.

### Большие базы кода труднее обслуживать

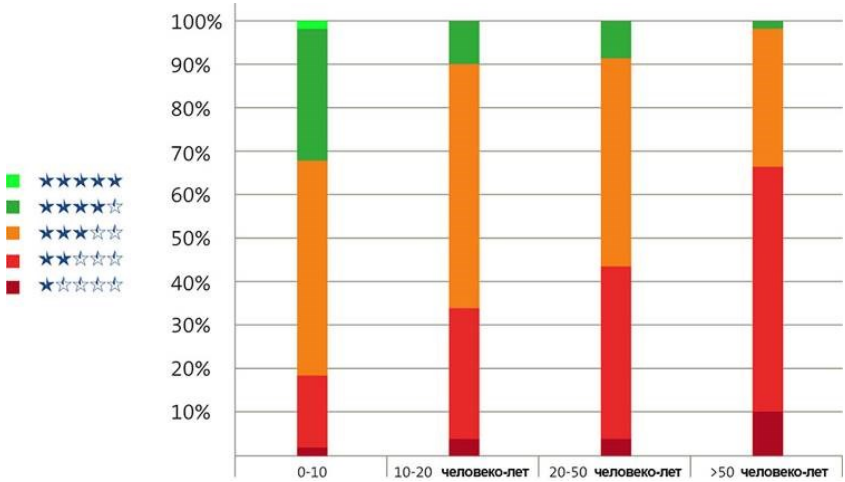


**Рис. 9.2** ❖ Разделение систем разных размеров в соответствии с эталонами обслуживаемости, принятыми в компании SIG

Рисунок 9.2 демонстрирует влияние размеров базы кода на ее обслуживаемость.

Диаграмма основана на сведениях о базах кода более чем 1500 систем, проанализированных компанией SIG. Объем трудозатрат измеряется в человека-годах, необходимых на воспроизведение системы (более подробно эта оценка рассматривается в разделе «Как в компании SIG оценивается объем базы кода» ниже). Каждый столбик отражает распределение систем по уровню обслуживаемости (на основе эталонов с рейтингом в виде звезд). Как демонстрирует диаграмма, более 30% систем из категории с наименьшим объемом кода удается достичь рейтинга в 4 и 5 звезд по удобству обслуживания, в то же время среди систем из категории с наибольшим объемом кода лишь малый процент достигает этого уровня.

### Большие системы отличаются высокой плотностью дефектов

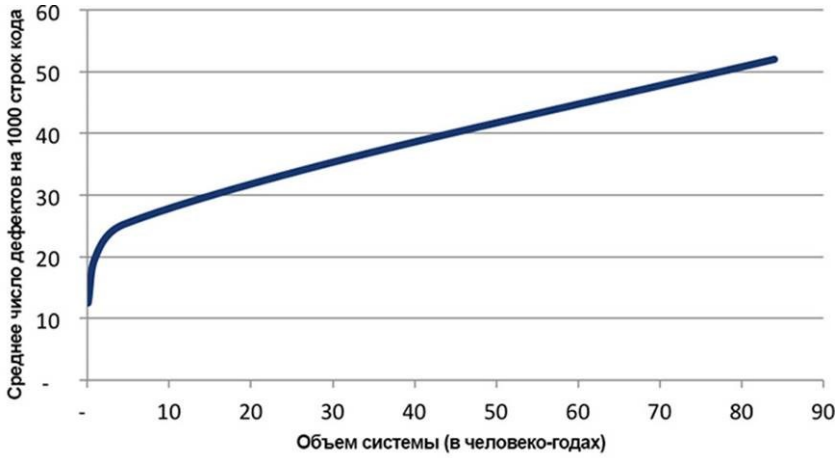


**Рис. 9.3** ❖ Влияние объема кода на количество дефектов[1]

Естественно, что большая система содержит большое количество дефектов в абсолютном выражении. Но плотность дефектов (определяемая как число дефектов на 1000 строк кода) также существенно возрастает с размером систем. Рисунок 9.3 демонстри-

[1] Источник: *McConnell S.* Code Complete. 2nd ed. Microsoft Press, 2004. С. 652.

рует связь между объемом кода и числом дефектов на 1000 строк кода. Поскольку число дефектов увеличивается с объемом кода, диаграмма показывает, что крупные системы имеют большее число дефектов как в абсолютном, так и в относительном выражении.

## 9.2. Как применять рекомендацию

При прочих равных условиях система с меньшими функциональными возможностями меньше системы с большими функциональными возможностями. Далее, реализация этих функциональных возможностей может быть лаконичной или пространной. Поэтому для уменьшения размеров базы кода сначала требуется ограничить функциональные возможности системы, а затем сосредоточиться на сокращении объема кода.

### Функциональные меры

Меры, связанные с функциональностью, не всегда находятся в вашей компетенции, но когда с разработчиками обсуждается добавление новых или адаптация существующих функциональных возможностей, необходимо учитывать следующее:

*Не допускайте расползания проекта*

Расползание проекта — обычное явление, когда масштабность требований к нему увеличивается в процессе разработки. Оно прикрывается лозунгом «а хорошо бы еще добавить такую функцию» и приводит к увеличению размера системы без добавления чего-то значимого для бизнеса или пользователей. Расползанию нужно противостоять, оценивая внесение дополнительных функциональных возможностей с позиции переноса сроков готовности проекта и увеличения будущих эксплуатационных расходов.

*Стандартизируйте функциональные возможности*

Под стандартизацией функциональных возможностей здесь понимается согласованность поведения и взаимодействия в программе. Стандартизация функциональных возможностей в первую очередь нацелена на исключение реализаций похожих функциональных возможностей, мало чем отличающихся друг от

друга. Во-вторых, стандартизация направлена на повторное использование кода, предполагая, что сам код пишется с учетом его многократного использования.

### Технические меры

Техническая реализация должна быть нацелена на использование как можно меньшего объема кода для реализации функциональных возможностей. Это достигается в основном за счет повторного использования кода посредством ссылок на него (вместо повторения в нескольких местах) или применения уже существующих библиотек и фреймворков.

*Не занимайтесь копированием и вставкой кода*

Ссылки на существующий код всегда предпочтительнее копирования и вставки кода в нескольких местах. При наличии нескольких копий одного и того же кода в процессе его обслуживания потребуется вносить изменения во все копии, размещенные в нескольких местах. При этом легко допустить ошибки, решая, нужна ли индивидуальная настройка каждому из повторяющихся фрагментов, и тестируя разбросанные копни. Обратите внимание, что представленная в главе 4 рекомендация посвящена именно отказу от копирования и вставки кода.

*Позаботьтесь об улучшении существующего кода*

Рефакторинг, улучшающий обслуживаемость, напрямую способствует уменьшению размеров базы кода. Обычно он включает ревизию кода, упрощение его структуры, удаление избыточных участков и увеличение количества многократно используемых фрагментов. Он может заключаться в простом удалении устаревших и больше не используемых функций. Более подробно шаблоны рефакторинга рассматривались в главе 4.

*Пользуйтесь сторонними библиотеками и фреймворками*

Многие приложения используют модели поведения, уже реализованные в огромном количестве фреймворков и библиотек, таких как jQuery (модель поведения пользовательского интерфейса), Hibernate (доступ к базам данных), Spring Security (механизм безопасности), SLF4J (журналирование), или общих служб

(например, Google Guava). Использование сторонних библиотек особенно полезно при осуществлении универсальных функций. Если функциональные возможности реализованы и поддерживаются сторонними разработчиками, для чего создавать свои собственные? Использование стороннего кода позволяет избежать ненужных трудозатрат. Стоит попытаться найти нужный код у сторонних производителей, а не создавать пользовательское решение.

Никогда не вносите изменений в исходный код сторонних библиотек, так как в этом случае часть кода библиотеки становится вашим собственным кодом. Это затрудняет обновление измененных библиотек и легко может привести к ошибкам. Как правило, это вызывает трудности при попытке обновить библиотеку до новой версии, поскольку необходимо проанализировать, что было изменено в коде библиотеки и как это повлияет на код, измененный локально.

*Разделяйте большие системы на части*

Разделение большой системы на несколько систем позволяет свести к минимуму проблемы, свойственные крупным системам. Необходимым условием разделения системы на части является их независимость с функциональной и технической точек зрения, а также точки зрения жизненного цикла. При этом пользователи должны четко представлять деление систем (или дополнительных модулей). Технически код в разных системах должен быть слабо связанным, то есть взаимодействия должны осуществляться через интерфейсы, а не посредством прямых вызовов. Системы действительно независимы, только когда имеют несвязанные жизненные циклы (то есть разрабатываются и выпускаются как самостоятельные продукты). Имейте в виду, что разделенные системы могут иметь взаимные или общие зависимости. В этом есть дополнительные преимущества. Это делает возможным замену некоторых подсистем пакетами сторонних разработчиков, что полностью исключает необходимость поддерживать соответствующие базы кода. Примером может служить распространение дистрибутивов Linux, подобных Ubuntu. Кодовая база ядра Linux распространяется через kernel.org (http://kernel.org/) и обслуживается большой командой добровольцев, возглавляемых Линусом Торвальдсом (Linus Torvalds).

Вместе с актуальным ядром типичный дистрибутив Linux содержит тысячи других программных приложений, каждое из которых имеет собственную базу кода. Все они по отношению к ядру являются дополнительными модулями.

Разделение (на уровне кода) более подробно рассматривалось в предыдущих главах, имеющих отношение к слабой связанности, главным образом в главе 7.

## 9.3. Типичные возражения против уменьшения размеров базы кода

Меры, описанные в этой главе, применимы ко всем этапам разработки программного обеспечения. Главной целью уменьшения базы кода является обслуживаемость.

Существуют два способа эффективного уменьшения размера базы кода: предупреждение (предотвращение дальнейшего роста базы кода) и исправление проблемы (уменьшение размеров базы кода).

Наиболее выгодной является ситуация, когда система уже имеет достаточно малый размер или находится на ранней стадии разработки. Чисто техническая корректировка, такая как рефакторинг кода и повторное использование функциональных возможностей, легче выполняется в небольшой системе и несет пользу для дальнейшей разработки.

Улучшения в большой системе становятся более заметными при удалении значительной ее части, например когда некоторые функциональные возможности заменяются сторонним кодом или после разделения системы на несколько частей.

### Возражение: сокращение размера базы кода снижает продуктивность разработки

*«Я не могу уменьшить размер своей системы, поскольку производительность процесса разработки оценивается объемом добавленного в нее кода».*

Подобные возражения только усугубляют проблему. Оценка продуктивности разработки объемом добавленного кода — плохая методика. Она служит отрицательным стимулом, поощряя дурную

привычку копировать и вставлять код. Предпочтение следует отдавать ссылкам на код, поскольку это упрощает анализ, тестирование и внесение изменений.

Понятно, что объем добавленного кода помогает руководителям следить за ходом разработки и прогнозировать сроки ее завершения. Но вообще продуктивность следует измерять добавленными функциями, а не строками кода. Опытные разработчики способны добавлять новые функции с минимальным количеством строк кода и всякий раз, когда появляется такая возможность, проводят рефакторинг, что часто приводит к уменьшению объема кода.

### Возражение: выбранный язык программирования препятствует уменьшению объема кода

*«Я работаю с языком программирования, более пространным, чем прочие, поэтому не имею возможности уменьшить базу кода».*

В большинстве проектов язык программирования четко определен. Действительно, некоторые языки программирования (например, основанные на SQL) не позволяют добиться малого размера базы кода. Тем не менее всегда следует стремиться уменьшить текущий объем кода при использовании одного и того же языка программирования. Уменьшение размеров любой базы кода выгодно, причем это касается даже кода, написанного на низкоуровневых языках программирования, слабо поддерживающих абстракцию.

### Возражение: сложность системы заставляет дублировать код

*«Наша система настолько сложна, что добавлять новые функции можно только копированием больших частей существующего кода. Поэтому уменьшить размер базы кода невозможно*».

Плохое понимание существующего кода приводит к появлению неуверенности при внесении изменений, что является распространенной причиной копирования существующего кода. Это особенно характерно для случаев, когда код плохо охвачен автоматизированным тестированием.

В подобных случаях постарайтесь найти функциональность, близкую к той, что требуется добавить. Проанализируйте код и выделите некие общие черты. Если таких черт нет, значит, нет смысла копировать этот код. Если оригинальную функциональность можно разбить на несколько частей, в идеале вы получите фрагмент, на который можно сослаться при реализации новой функциональности, избежав дублирования и роста размера базы кода. Напишите модульные тесты для новых блоков кода, чтобы убедиться в правильности понимания их внутренней структуры. Более подробные сведения об этом можно найти в главе 10.

### Возражение: разделение базы кода невозможно из-за архитектуры платформы

*«Мы не можем разделить систему на меньшие части, поскольку сборка предназначена для платформы, чья функциональность используется повсюду в базе кода».*

Действительно, программное обеспечение для платформ имеет тенденцию разрастаться со временем, поскольку осваивает новые функции и редко от них отказывается. Тем не менее есть возможность резко уменьшить размер базы кода, реализовав модульную архитектуру и разделив систему на подключаемые модули. В результате появится несколько баз кода, каждая из которых будет меньше первоначальной. Одна база кода будет содержать общее ядро, и одна или несколько — дополнительные модули. Если модули не будут технически связаны, их можно выпускать отдельно. Это означает, что небольшие изменения не будут требовать обновления всей системы. Имейте в виду, что даже небольшие обновления должны охватываться интеграционным и регрессивным тестированием, чтобы иметь уверенность, что система в целом по-прежнему функционирует правильно.

### Возражение: разделение кода приводит к дублированию

*«Разделение кода заставляет меня дублировать код».*

Иногда разделение системы на отдельные части (такие как плагины и расширения) требует дублирования в них общих функций (интерфейсов) или структур данных.

В таком случае дублирование является большей проблемой, чем размер базы кода, и рекомендация, представленная в главе 4, превалирует над рекомендацией стремиться к уменьшению объема общего объема кода. Попробуйте выделить код с реализацией общих функций в отдельное расширение или включите его в базу общего кода.

### Возражение: разделение базы кода невозможно из-за тесной связанности

*«Я не могу разделить систему, поскольку все ее части тесно связаны между собой».*

Для начала устраните тесные связи. Для этого попробуйте определить специальные интерфейсы, выступающие в роли единой точки входа. Это можно сделать с помощью веб-служб или других инструментов, обеспечивающих такую возможность (например, промежуточного программного обеспечения или ESB).

Имейте в виду, что цель состоит в создании подсистем, обслуживаемых независимо друг от друга, которые совсем не обязательно должны работать независимо друг от друга.

---

### Как в компании SIG оценивается объем базы кода

При оценке объема базы кода не предусмотрено разделение на различные категории риска, поскольку она основана на измерении только одной величины. Объем базы кода системы с рейтингом в 4 звезды не должен превышать 20 человеко-лет, необходимых для воспроизведения. Для систем на языке C# это соответствует примерно 160 000 строкам кода.

#### Человеко-месяцы и человеко-годы

Общий объем базы кода, выраженный в строках кода, преобразуется в человеко-месяцы. Человеко-месяц — это стандартная мера объема исходного кода, который может написать один разработчик со средней производительностью за один месяц. Измерение в «человеко-месяцах» позволяет производить сравнение объемов исходного кода, написанного с применением различных технологий. Это важно, поскольку языки программирования имеют различные степени производительности, или «уровни детализации». Таким

образом, в системе, где используется несколько языков программирования, строки кода можно перевести в агрегированный показатель, основанный на затратах труда, необходимых для ее «воспроизведения».
Опыт SIG показывает, что человеко-месяцы — эффективный показатель для оценки размера системы и сравнения систем друг с другом. Человеко-год — это просто 12 человеко-месяцев. Конечно, фактическая производительность зависит также от мастерства и стиля программирования. Величина объема ничего не говорит о том, сколько месяцев или лет действительно было затрачено на разработку системы.

# Глава 10

# Автоматизируйте тестирование

Добивайтесь появления зеленой полосы,
чтобы сохранить код в чистоте.

*Девиз jUnit*

Рекомендация:
• **автоматизируйте тестирование базы кода**;
• для этого **пишите автоматизированные тесты с использованием специализированных фреймворков**;
• это улучшит обслуживаемость, поскольку **автоматизированное тестирование делает разработку предсказуемой и менее рискованной**.

В главе 4 рассматривался метод `IsValid`, проверяющий допустимость номера банковского счета вычислением контрольной суммы. В подобных методах легко допустить ошибку. Вот почему практически каждому программисту в какой-то момент приходилось писать маленькие одноразовые программы для тестирования подобных методов, которые выглядят примерно так:

```
using System;
using eu.sig.training.ch04.v1;

namespace eu.sig.training.ch10
{
    public class Program
    {
        [STAThread]
        public static void Main(string[] args)
        {
            string acct;
            do
            {
                Console.WriteLine("Type a bank account number on the next line.");
```

```
            acct = Console.ReadLine();
            Console.WriteLine($"Bank account number '{acct}' is" +
                (Accounts.IsValid(acct) ? "" : " not") + " valid.");
        } while (!String.IsNullOrEmpty(acct));
    }
  }
}
```

Это класс C# с методом `Main`, поэтому его можно запустить из командной строки:

```
C:\> Program.exe
Type a bank account number on the next line.
123456789
Bank account number '123456789' is valid.
Type a bank account number on the next line.
123456788
Bank account number '123456788' is not valid.
C:\>
```

Эту программу можно назвать *ручным модульным тестом*. Модульным этот тест называется потому, что предназначен для тестирования единственного модуля (блока кода) `IsValid`, а ручным — потому, что пользователь должен вручную вводить тестовые значения и сам определять правильность результата.

Хотя такой подход предпочтительнее, чем полное отсутствие модульного тестирования, он обладает рядом недостатков:

- тестовые данные вводятся вручную, поэтому тест не может быть выполнен автоматически, без приложения дополнительных усилий;
- разработчик, написавший тест, основное внимание уделяет логике выполнения теста (цикл `do ... while`, обработка ввода и вывода), а не собственно тестированию;
- программа не сообщает, как метод `IsValid` должен вести себя;
- программа не распознается как тест (хотя довольно общеупотребительное название программы `Program` говорит, что это одноразовый эксперимент).

Именно поэтому следует писать автоматизированные модульные тесты, а не ручные. Они описывают тестирование блоков кода, выполняющихся автономно. Это справедливо и в отношении других видов тестирования, таких как регрессионных и приемочных, автоматизируемых с помощью стандартных *фреймворков тести-*

*рования.* Для создания модульных тестов широко используется популярный фреймворк NUnit (http://nunit.org/).

## 10.1. Мотивация

В этом разделе описываются преимущества полной *автоматизации* тестирования.

### Автоматизация делает тестирование повторяемым

Подобно другим программам и сценариям, автоматизированные тесты всегда выполняются совершенно одинаково. Это делает тестирование повторяемым: если некоторый тест в разные моменты времени возвращает разные результаты, эго не значит, что сам тест содержит ошибку, скорее, это свидетельствует о том, что в системе произошли изменения, вызвавшие появление другого результата. При ручном тестировании всегда есть возможность непоследовательного выполнения тестов из-за ошибок, сделанных человеком.

### Автоматизированное тестирование увеличивает эффективность разработки

Для выполнения автоматических тестов требуется гораздо меньше усилий, чем при ручном тестировании, благодаря чему тестирование можно проводить так часто, как это необходимо. Кроме того, автоматизированное тестирование занимает меньше времени, чем проверка кода вручную. Тестирование следует начинать с самых ранних этапов разработки, чтобы уменьшить затраты на исправление ошибок.

Откладывание тестирования до заключительного этапа разработки несет риск позднего выявления проблем и увеличения затрат на внесение исправлений, поскольку требует возврата к началу конвейера разработки и повторного проведения тестирования.

### Автоматизированное тестирование делает код предсказуемым

С технической точки зрения, процесс тестирования можно сделать высокоавтоматизированным. Возьмем модульные и интеграционные тесты: вместе они проверяют не только внутреннюю работу

кода, но и его согласованность. Не будучи уверенным в правильности внутренней работы системы, нужных результатов можно достичь только случайно. Это подобно передвижению на автомобиле, вы можете случайно прибыть в пункт назначения, выбрав неправильное направление, но у вас нет уверенности, что вы попадете в нужный пункт, если наверняка не известно, что выбрано правильное направление.

Преимуществом автоматизированного тестирования является определение момента наступления регрессии. Без пакета автоматизированных модульных тестов разработка быстро становится похожа на игру в прятки, когда после изменения одного фрагмента кода, при следующем внесении изменений в другой фрагмент выясняется, что при предыдущем изменении была допущена ошибка. Автоматическое тестирование позволяет без особых усилий проверить всю базу кода, прежде чем переходить к следующему изменению. И поскольку результат автоматизированных модульных тестов предопределен, можно быть уверенным, что после исправления ошибка не появится вновь в будущем.

Таким образом, автоматизированное тестирование дает гарантии правильной работы кода. Следовательно, предсказуемость автоматизированных тестов обеспечивает предсказуемое качество разработанного кода.

### Тесты документируют тестируемый код

Код теста сценария или программы содержит сведения об ожидаемом поведении системы. Например, как будет показано ниже в этой главе, тест метода `IsValid` содержит следующую строку: `AssertIsFalse(IsValid(""))`. Это документирующее утверждение на языке C#, что, получив пустую строку, метод `IsValid` должен вернуть значение `false`. То есть инструкция `AssertIsFalse` играет двойную роль: тестирует поведение метода и описывает его. Другими словами, тесты служат примерами, как должна действовать система.

### Разработка тестов улучшает качество кода

Разработка тестов помогает писать тестируемый код. Как побочный эффект разработки тестов блоки основного кода получаются более короткими, простыми, с меньшим числом параметров и слабо связанными (в соответствии с рекомендациями в предыдущих главах). Например, метод сложно тестировать, когда он решает не одну, а сразу несколько задач. Чтобы упростить тестирование, разработчик часто стремится разделить обязанности между несколькими методами, что приводит к улучшению обслуживаемости в целом. Поэтому некоторые разработчики практикуют создание модульных тестов до тестируемого ими кода. Такой подход лежит в основе методики *разработки через тестирование* (Test-Driven Development, TDD). Вы сами убедитесь, что проектировать методы становится проще, когда начнете думать о том, как будете их тестировать: как проверить допустимость аргументов и что должен вернуть метод в результате?

## 10.2. Как применять рекомендацию

Порядок автоматизации тестирования зависит от вида тестов, которые необходимо автоматизировать. Виды тестов перечислены в табл. 10.1 и различаются по признакам: *что* тестируется, *кем* и *почему*. Они отсортированы по областям тестирования. Например, областью модульных тестов являются блоки кода, а сквозные, регрессионные и приемочные тесты выполняют тестирование на уровне системы.

***Таблица 10.1 ❖ Виды тестирования***

| Вид | Что тестируется | Почему | Кто |
|---|---|---|---|
| Модульные тесты | Один блок кода в изоляции | Чтобы убедиться, что блок кода ведет себя ожидаемо | Разработчик (предпочтительно этого блока) |
| Интеграционные тесты | Функциональные возможности, производительность и другие качественные характеристики, по крайней мере двух классов | Чтобы убедиться в правильности взаимодействий частей системы | Разработчик |
| Сквозные тесты | Взаимодействие с системой (пользователя или другой системы) | Чтобы убедиться, что система ведет себя ожидаемо | Разработчик |
| Регрессионные тесты | Выявленное ранее ошибочное поведение блока кода, класса или интерактивности системы | Чтобы убедиться, что ошибки не появились вновь | Разработчик |
| Приемочные тесты | Взаимодействие с системой (пользователя или другой системы) | Для подтверждения требуемого поведения системы | Представитель конечного пользователя (ни в коем случае не разработчик) |

Таблица 10.1 показывает, что регрессионные тесты могут быть модульными, интеграционными и сквозными и создаются после исправления ошибок. Приемочные тесты — это те же сквозные тесты, но выполняются представителями конечного пользователя.

Для автоматизации различных видов тестирования применяются различные фреймворки. Для автоматизации модульного тестирования существует несколько популярных C#-фреймворков, например NUnit (http://nunit.org/). Для автоматизации сквозного тестирования необходим фреймворк, способный имитировать ввод пользователя и перехватывать выходные данные. Одним из таких фреймворков является Selenium (http://www.seleniumhq.org/). Выбор средств интеграционного тестирования зависит от рабочего окружения и проверяемых качественных характеристик. Например, фреймворк SoapUI (http://www.soapui.org/) используется для интеграционного тестирования веб-служб и инструментов обмена сообщениями. Фреймворк Apache jMeter (http://jmeter.apache.org/)

позволяет тестировать производительность C#-приложений в условиях тяжелых рабочих нагрузок.

Выбор фреймворка производится на уровне команды. Создание интеграционных тестов требует весьма специфических навыков, но модульное тестирование доступно каждому разработчику. Поэтому далее в этой главе основное внимание уделяется разработке модульных тестов с помощью широко известного фреймворка NUnit.

В отличие от специализированных интеграционных и сквозных тестов, для разработки модульных тестов требуются навыки, которыми должен владеть каждый разработчик.

Для создания модульных тестов не требуется ничего особенного[1], просто загрузите фреймворк NUnit на странице http://nunit.org/.

### Начало работы с NUnit

Как упоминалось во введении к этой главе, нужно проверить метод `IsValid` класса `Accounts`. Класс `Accounts` называется *тестируемым классом*. В NUnit тесты помещаются в отдельный класс, называемый *тестовым классом,* или *оснасткой теста.* Этот класс отмечается атрибутом `[TestFixture]`. По соглашению имя тестового класса должно повторять имя тестируемого класса с добавлением суффикса `Test`. В данном случае тестовый класс получит имя `AccountsTest`. Это должен быть общедоступный класс, но к нему не предъявляется никаких других требований. В частности, не требуется, чтобы он наследовал какой-либо другой класс. Это позволяет, но не обязывает, поместить тестовый класс в то же пространство имен, в котором находится тестируемый класс. Благодаря этому тестовый класс сможет получить доступ ко всем свойствам и методам (не общедоступным) пакета, к которым имеется доступ из пространства имен тестируемого класса.

Сам тест в фреймворке NUnit определяется как самый обычный метод, но отмеченный атрибутом `[Test]`. Для тестирования метода `IsValid` можно использовать следующий тестовый класс NUnit:

```
using NUnit.Framework;
```

[1] В настоящее время Visual Studio включает фреймворк Unit Testing Framework (https://msdn.microsoft.com/ru-ru/library/ms243147%28v=vs.90%29.aspx) от Microsoft, который мало чем отличается от NUnit.

```
namespace eu.sig.training.ch04.v1
{
    [TestFixture]
    public class AccountsTest
    {
        [Test]
        public void TestIsValidNormalCases()
        {
            Assert.IsTrue(Accounts.IsValid("123456789"));
            Assert.IsFalse(Accounts.IsValid("123456788"));
        }

}
```

Этот тест проверяет два случая:

- строка 123456789, как упоминалось во врезке «Проверка контрольной суммы номеров банковских счетов на кратность 11» (в главе 2), является допустимым номером банковского счета, поэтому для нее метод `IsValid` должен вернуть значение `true`. Вызов метода `Assert.IsTrue` проверяет это;
- строка 123456788 является недопустимым номером банковского счета (поскольку отличается от допустимого номера банковского счета только одной цифрой), поэтому метод `IsValid` должен вернуть значение `false`. Вызов метода `Assert.IsFalse` проверяет это.

Модульные тесты можно запускать непосредственно из Visual Studio. Кроме того, в состав NUnit входит средство для запуска тестов из командной строки. Тесты могут также выполняться из Maven или Ant. На рис. 10.1 показан результат выполнения предыдущего теста в Visual Studio. Красная полоса указывает на неудачи при выполнении тестов.

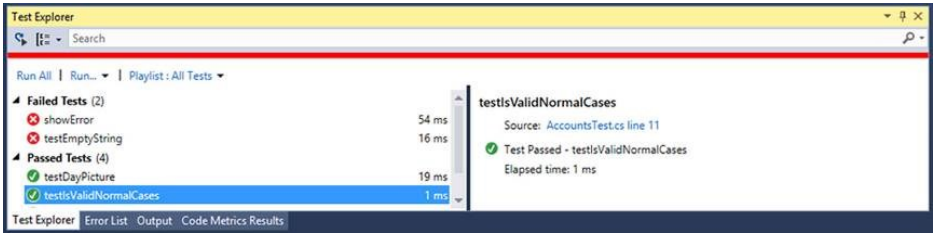


**Рис. 10.1** ❖ Все тесты успешно пройдены!

В предыдущем тестовом классе тестировались только нормальные случаи: два номера банковских счетов в требуемом формате (ровно девять символов и только цифры). А как же особые случаи? Одним из очевидных особых случаев является пустая строка. Пустая строка, несомненно, не является допустимым номером банковского счета, поэтому проверим ее с помощью метода `Assert.IsFalse`:

```
[Test]
public void TestEmptyString()
{
    Assert.IsFalse(Accounts.IsValid(""));
}
```

Как можно видеть на рис. 10.2, этот тест не был пройден! Метод `IsValid` должен был вернуть значение `false`, а вернул что-то другое (конечно же, это значение `true`, поскольку других вариантов просто нет).

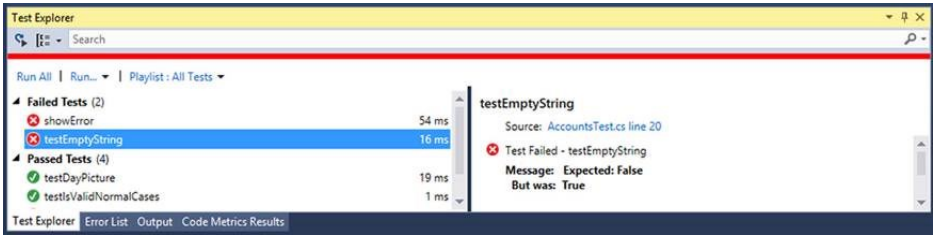


**Рис. 10.2** ❖ Один из тестов не пройден

Неудачный тест указывает на наличие дефекта в методе `IsValid`. В случае если аргумент `IsValid` является пустой строкой, цикл `for` не выполняется. То есть выполняются только строки:

```
int sum = 0;
return sum % 11 == 0;
```

В результате возвращается значение true, когда должно возвращаться значение `false`. Тест указывает на необходимость добавить в метод `IsValid` код, контролирующий длину номера банковского счета[1].

[1] Этого тоже недостаточно. Поскольку метод `System.GetNumericValue` возвращает значение —1,0 для всех нецифровых символов, вызов `IsValid("72345678z")` вернет значение `true`.

Инструмент запуска тестов в NUnit в данном случае сообщает о *неудачном тестировании,* а не об *ошибке* теста. Под *неудачным* тестированием подразумевается, что сам тест (метод `TestEmptyString`) работает отлично, но проверяемое утверждение оказалось ошибочным. *Ошибка* теста означает, что ошибку вызвал сам метод тестирования. Это демонстрирует следующий фрагмент кода, содержащий метод `ShowError`, вызывающий исключение деления на нуль еще до того, как будет выполнен метод `Assert.IsTrue`:

```
[Test]
public void ShowError()
{
    int tmp = 0, dummy = 1 / tmp;
    // Следующая строка никогда не будет выполнена, потому что
    // предыдущая строка вызовет исключение.
    // Если бы она была выполнена, сообщение не было бы выведено,
    // поскольку тест всегда завершается успехом.
    Assert.IsTrue(true);
}
```

Далее перечисляются основные принципы, помогающие писать хорошие модульные тесты. Начнем с самых простых, а затем перейдем к более продвинутым способам тестирования.

### Общие принципы разработки хороших модульных тестов

При разработке тестов важно помнить следующие общие принципы:

*Тестируйте как обычные, так и особые случаи*

Подобно примерам, приведенным в этой главе, тесты должны охватывать два вида случаев. Тесты должны проверять правильное поведение блока кода при передаче обычных входных данных (так называемое *оптимистичное,* или *светлое, тестирование*). Кроме этого, тесты должны подтверждать ожидаемое поведение блока кода при передаче необычных входных данных и в особых обстоятельствах (так называемое *пессимистическое,* или *темное, тестирование).* Например, в NUnit можно писать тесты, подтверждающие, что тестируемый метод действительно вызывает определенные исключения.

*Обслуживайте тесты так же, как основной (рабочий) код*

За изменением кода системы должны следовать соответствующие корректировки в модульных тестах. Это наиболее актуально для модульных тестов, хотя касается тестов всех других видов. В частности, при добавлении новых или расширении существующих методов не забывайте добавлять новые тесты, охватывающие новый код.

*Пишите изолированные тесты, их результаты должны отражать только поведение тестируемого субъекта*

То есть каждый тест должен выполняться независимо от других. В модульном тестировании это означает, что каждый тест должен контролировать только одну функцию. Модульные тесты не должны зависеть от текущего состояния системы, например от файлов, хранимых другими тестами. Именно поэтому модульный тест, который, скажем, вызывает тестируемый класс для доступа к файловой системе или к серверу базы данных, не является хорошим модульным тестом.

Как следствие в модульных тестах требуется, при необходимости, моделировать состояние или входные данные, получаемые из других классов (например, в виде аргументов). В противном случае тест получится неизолированным и будет тестировать сразу несколько блоков кода. В случае с методом `IsValid` достичь этого было легко, поскольку метод `IsValid` принимает один строковый аргумент и не вызывает других методов системы. В других ситуациях может потребоваться специальный подход, с применением *заглушек* или *фиктивных объектов.*

В главе 6 был представлен C#-интерфейс для простой цифровой камеры, который повторяется ниже для удобства:

```
public interface ISimpleDigitalCamera
{
    Image TakeSnapshot();

    void FlashLightOn();

    void FlashLightOff();
}
```

Предположим, что этот интерфейс используется в приложении, принудительно включающем вспышку в темное время суток:

```
public const int DAYLIGHT START = 6;

public Image TakePerfectPicture(int currentHour)
{
    Image image;
    if (currentHour < PerfectPicture.DAYLIGHT START)
    {
        camera.FlashLightOn();
        image = camera.TakeSnapshot();
        camera.FlashLightOff();
    }
    else
    {
        image = camera.TakeSnapshot();
    }
    return image;
}
```

Несмотря на всю простоту логики (метод `TakePerfectPicture` предполагает, что если текущее время суток, выраженное в 24-часовом формате, меньше 6 часов, — это ночное время), метод нуждается в тестировании. Для правильного модульного тестирования метода `TakePerfectPicture` необходимо обеспечить автоматическое и независимое получение снимка. Это означает, что нельзя использовать нормальную реализацию интерфейса цифровой камеры. В нормальной реализации для обычного устройства требуется вмешательство пользователя (человека), который наведет камеру на что-то интересное и нажмет кнопку. На фотографии может получиться любое изображение, поэтому трудно проверить, является ли полученная (предположительно качественная) фотография той, что ожидалось.

Решение заключается в использовании реализации интерфейса камеры, специально созданной для тестирования. Эта реализация является *фиктивным объектом* и называется *тестовой заглушкой,* или просто *заглушкой*[1]. Здесь требуется, чтобы фиктивный объект

[1] Книги н другие ресурсы, посвященные тестированию, практически не содержат соглашений о терминологии. Здесь применены термины из книги Роя Ошерове (Roy Osherove) «The Art of Unit Testing» (издательство Manning, 2009).

вел себя жестко запрограммированным (и поэтому предсказуемым) образом. Напишем следующую тестовую заглушку:

```
class DigitalCameraStub : ISimpleDigitalCamera
{
    public Image TestImage;

    public Image TakeSnapshot()
    {
        return this.TestImage;
    }

    public void FlashLightOn()
    {
    }

    public void FlashLightOff()
    {
    }
}
```

В этой заглушке метод `TakeSnapshot` всегда возвращает одно и то же изображение, просто присваивая его переменной `testImage` (для упрощения свойство `testImage` сделано общедоступным, и для него не предусмотрены методы чтения и записи). Теперь заглушку можно использовать в тесте:

```
[Test]
public void TestDayPicture()
{
    Image image =
        Image.FromFile("../../../../test/resources/VanGoghSunflowers.jpg");
    DigitalCameraStub cameraStub = new DigitalCameraStub();
    cameraStub.TestImage = image;
    PerfectPicture.camera = cameraStub;
    Assert.AreSame(image, new PerfectPicture().TakePerfectPicture(12));
}
```

В этом тесте создается заглушка камеры, и ей передается возвращаемое изображение. Затем вызывается метод `TakePerfectPicture(12)` и проверяется, вернет ли он правильное изображение. Значение 12, передаваемое в вызов, означает, что метод `TakePerfectPicture` должен воспринимать текущее время как находящееся в пределах между полуднем и 13:00.

Теперь предположим, что нужно проверить работу метода TakePerfectPicture для темного времени суток, то есть проверить, действительно ли вызов метода TakePerfectPicture с аргументом меньше PerfectPicture.DAYLIGHT_START включает вспышку. Другими словами, требуется проверить вызов метода FlashLightOn. Однако FlashLightOn ничего не возвращает, и интерфейс ISimpleDigitalCamera не предусматривает способа, позволяющего узнать, была ли включена вспышка. Так как это проверить?

Решение заключается в создании в заглушке цифровой камеры механизма регистрации вызовов определенных методов. Объект заглушки, регистрирующий вызовы методов, называется *фиктивным объектом.* То есть фиктивный объект — эго объект заглушки с дополнительными функциями, необходимыми для тестирования. Ниже показано, как выглядит фиктивный объект цифровой камеры для нашего случая:

```
class DigitalCameraMock : ISimpleDigitalCamera
{
    public Image TestImage;
    public int FlashOnCounter = 0;

    public Image TakeSnapshot()
    {
        return this.TestImage;
    }

    public void FlashLightOn()
    {
        this.FlashOnCounter++;
    }

    public void FlashLightOff()
    {
    }
}
```

От объекта DigitalCameraStub он отличается дополнительным счетчиком вызовов метода FlashLightOn. Фиктивный объект DigitalCameraMock поддерживает то же запрограммированное поведение, что и заглушка. Проверку вызова метода FlashLightOn можно выполнить, как показано ниже:

```
[Test]
```

```
public void TestNightPicture()
{
    Image image =
        Image.FromFile("../../../../test/resources/VanGoghStarryNight.jpg");
    DigitalCameraMock cameraMock = new DigitalCameraMock();
    cameraMock.TestImage = image;
    PerfectPicture.camera = cameraMock;
    Assert.AreSame(image, new PerfectPicture().TakePerfectPicture(0));
    Assert.AreEqual(1, cameraMock.FlashOnCounter);
}
```

В этих примерах мы написали свои заглушки и фиктивные объекты. Это потребовало достаточного большого объема кода. Как правило, для этого имеет смысл воспользоваться специализированным фреймворком, таким как Moq (https://github.com/Moq/moq4). Фреймворки используют возможности среды выполнения .Net для автоматического создания фиктивных объектов из обычных интерфейсов и классов. Они также имеют методы для регистрации вызовов методов фиктивного объекта и аргументов. Некоторые фреймворки поддерживают возможность определения запрограммированного поведения фиктивных объектов, придавая им характеристики заглушек и фиктивных объектов одновременно.

В самом деле, с помощью фреймворка Moq можно написать метод `TestNightPicture`, не создавая класса, подобного `DigitalCameraMock`:

```
[Test]
public void TestNightPictureMoq()
{
    Image image =
        Image.FromFile("../../../../test/resources/VanGoghStarryNight.jpg");
    var cameraMock = new Mock<ISimpleDigitalCamera>();
    cameraMock.Setup(foo => foo.TakeSnapshot()).Returns(image);
    PerfectPicture.camera = cameraMock.Object;
    Assert.AreSame(image, new PerfectPicture().TakePerfectPicture(0));
    cameraMock.Verify(foo => foo.FlashLightOn(), Times.AtMostOnce());
}
```

Этот тест создает фиктивный объект `cameraMock` с помощью конструктора `Mock`. Затем вызовом методов `Setup` и `Returns` фреймворка Moq определяется требуемое поведение, а метод `Verify` используется для проверки вызова метода `FlashLightOn`.

### Оценка охвата для определений достаточности количества тестов

Сколько необходимо модульных тестов? Для определения достаточности написанных модульных тестов часто используется оценка *охвата* кода модульными тестами. Охват — это процент строк кода, фактически выполняемых во время тестирования. Как правило, нужно стремиться к *охвату тестами не менее 80% строк рабочего кода.*

Почему нужно охватить тестами именно 80% кода (а не 100%)? Любая база кода содержит фрагменты тривиального кода, которые технически можно протестировать, но они столь тривиальны, что их тестирование бессмысленно. В качестве примера рассмотрим метод чтения значения свойства:

```
public string Name { get; }
```

Мы можем протестировать этот метод, читая значение (с помощью чего-то вроде `Assert.AreEqual(myObj.Name, "John Smith")`), но этот тест в основном предназначен для тестирования C#-компилятора и среды выполнения .Net. Это не означает, что никогда не нужно тестировать методы чтения свойств. Возьмем типичный класс, представляющий почтовые адреса. Он, как правило, содержит два или три строковых поля, содержащих (дополнительные) строки адреса. При этом легко можно допустить ошибку, например:

```
public string getAddressLine3() {
    return this.addressLine2;
}
```

Одного только соблюдения порога охвата 80% недостаточно для высококачественного модульного тестирования. Можно добиться высокого уровня охвата, реализовав тестирование лишь нескольких высокоуровневых методов (таких как метод `Main`, первый метод, вызываемый средой выполнения .NET), *не* проверяя низкоуровневых методов. Именно поэтому рекомендуется стремиться к соотношению рабочего кода к коду тестов, как 1 к 1.

Для оценки охвата часто используются специализированные инструменты. Некоторые редакции Visual Studio имеют встроенные инструменты оценки охвата. На рис. 10.3 показан отчет об охвате

тестами базы кода для примеров в этой книге, полученный в Visual Studio 2015 Enterprise Edition.

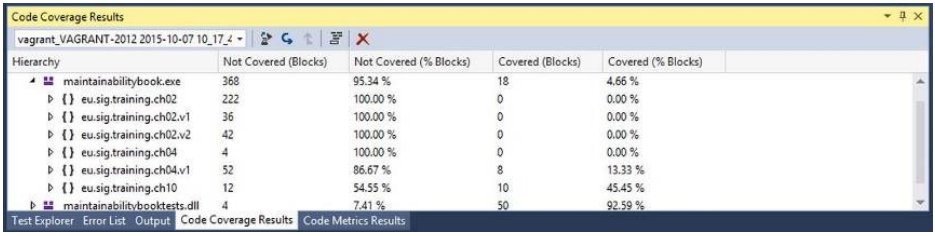


**Рис. 10.3** ❖ Отчет об охвате тестами примеров для книги, полученный в Visual Studio 2015 Enterprise Edition

## 10.3. Типичные возражения против автоматизации тестов

В этом разделе рассматриваются типичные возражения и ограничения, касающиеся автоматизации тестирования. Они основаны на причинах и соображениях, связанных с затратами на автоматизацию тестирования.

### Возражение: нам нужно и ручное тестирование

*«Почему мы должны нести затраты на автоматизацию, если нам все равно не обойтись без ручного тестирования*?»

Ответ на этот вопрос прост: потому что автоматизация тестирования освобождает время на то, чтобы вручную проверить те моменты, которые не могут быть автоматизированы.

Рассмотрим минусы альтернативы автоматическому тестированию. Ручное тестирование имеет четкие ограничения. Это медленный, дорогой и трудно повторяемый, в смысле последовательности его выполнения, процесс. В самом деле, техническая проверка системы *должна* иметь место в любом случае, поскольку невозможно вручную протестировать код, который не работает. Ручное тестирование не слишком легко повторить, а после внесения даже небольших изменений требуется полное повторное тестирование кода, чтобы удостовериться, что система работает правильно.

Большая часть ручного тестирования может быть автоматизирована с помощью автоматизированных регрессионных тестов. Благодаря этому уменьшится объем ручного тестирования. Вам

все еще может потребоваться произвести ревизию кода или приемочные испытания, чтобы убедиться, что *бизнес-логика* работает правильно. Это обычно касается процессов с изобилием функциональных возможностей.

### Возражение: мне не разрешается писать модульные тесты

*«Мне не разрешается писать модульные тесты, потому что руководство считает, что это снижает мою производительность».*

На самом деле написание модульных тестов во время разработки повышает производительность труда и качество кода системы, потому что переносит внимание с точки зрения «что код должен делать» на «чего он делать не должен». Если не учитывать, что код может завершиться ошибкой, нельзя быть уверенным в его устойчивости в неожиданных ситуациях.

Отсутствие модульных тестов чревато непредсказуемостью и риском необходимости переделок. При каждом изменении фрагмента кода требуется вновь и вновь внимательно проверять код, чтобы убедиться, что он выполняет то, что от него требуется.

### Возражение: зачем тратить время на модульные тесты при низком текущем охвате ими?

*«Текущий охват системы модульным тестированием очень низок. Почему я должен тратить время на написание модульных тестов?»*

Мы рассказали, почему модульные тесты полезны и как они помогают писать код, выполняющийся предсказуемо. Однако когда большая система практически не охвачена модульными тестами, их создание может превратиться в тяжелое бремя. Написание модульных тестов с нуля для существующей системы займет массу времени, поскольку потребуется еще раз проанализировать все блоки ее кода. Поэтому тратить время на написание модульных тестов следует только в случае полной уверенности, что это стоит затраченных усилий. Это особенно касается критических, основных функциональных возможностей и тогда, когда есть основания полагать, что блоки кода ведут себя непредсказуемо. В противном

случае добавляйте модульные тесты постепенно, при каждом изменении существующего или добавлении нового кода.

Вообще, когда охват системы модульными тестами значительно ниже стандартных 80%, лучшей стратегией является применение «правила бойскаутов». Это правило советует оставить после себя код в лучшем состоянии, чем в котором вы его получили (более подробно применение этого принципа рассматривается в главе 12). То есть, изменяя код, вы можете добавлять (изменять) модульные тесты, чтобы убедиться, что он все еще работает, как ожидалось.

## 10.4. Дополнительные сведения

Стандартизация и последовательность в применении играют важную роль в создании автоматизированной среды разработки. Дополнительные сведения см. в главе 11.

---

**Как в компании SIG оценивается тестируемость**

Тестируемость, согласно стандарту ISO 25010, является одной из характеристик обслуживаемости. В SIG рейтинг тестируемости получают путем суммирования рейтингов систем по сложности блоков кода (глава 3), независимости компонентов (глава 7) и размеру систем (глава 9) с помощью механизма агрегации, рассматриваемого в приложении A.

Объясняется это тем, что сложные блоки кода тяжело тестировать, зависимость компонентов увеличивает потребность в фиктивных объектах и заглушках, а больший объем рабочего кода требует большего объема тестового кода.

# Глава 11

# Пишите чистый код

Чтобы называть себя профессионалом, вы должны писать чистый код.

*Роберт С. Мартин (Robert C. Martin)*

Рекомендация:

- **пишите чистый код;**
- для этого **не оставляйте грязи в коде после завершения работы над ним;**
- это улучшит обслуживаемость, потому что **обслуживать чистый код значительно проще.**

Грязь в коде свидетельствует о наличии в нем проблем. Добавление или неудаление грязи является плохой привычкой и снижает обслуживаемость кода. В этой главе рассматриваются рекомендации по сохранению кода чистым и соответствующим «гигиеническим условиям».

## 11.1. Не оставляйте следов

У бойскаутов есть правило, которое гласит: «оставляйте лагерь в более чистом состоянии, чем он был до вас». Применительно к разработке программного обеспечения правило бойскаутов означает, что при написании или изменении фрагмента кода всегда есть возможность хотя бы немного улучшить его, сделать чище и удобнее. Если сейчас вы корректируете фрагмент кода, значит, потом потребуется его обслуживать. Это увеличивает вероятность, что к нему придется вернуться, и когда это произойдет, вы получите выгоды от рефакторинга, выполняемого сейчас.

## 11.2. Как применять рекомендацию

Писать только чистый код сложно, но мы дадим вам несколько советов, которые помогут достичь этой цели. Опираясь на наш опыт, мы выделили семь «правил бойскаутов», следование которым поможет убрать грязь из кода и улучшить его обслуживаемость:

1) не оставляйте после себя грязи на уровне блоков кода;
2) не оставляйте после себя неудачных комментариев;
3) не оставляйте после себя закомментированного кода;
4) не оставляйте после себя неиспользуемого кода;
5) не оставляйте после себя длинных идентификаторов;
6) не оставляйте после себя таинственных констант;
7) не оставляйте после себя плохую обработку исключений.

А в следующих разделах подробно объясним их.

### Правило 1: не оставляйте после себя грязи на уровне блоков кода

На данный момент вы познакомились с девятью рекомендациями создания обслуживаемого программного обеспечения, которым были посвящены девять предыдущих глав. Из этих девяти рекомендаций три касаются грязи на уровне блоков кода: длинные блоки кода (глава 2), сложные блоки кода (глава 3) и блоки кода с длинными интерфейсами (глава 5). Современные языки программирования не оставляют причин нарушать эти рекомендации при написании нового кода.

Для следования этому правилу достаточно вовремя заниматься реорганизацией «грязного» кода. Под «вовремя» подразумевается — как можно быстрее, но определенно до передачи кода в систему управления версиями. Конечно, в работе допустимы небольшие отступления от правил, например наличие метода в 20 строк кода или метода с 5 параметрами. Но эти нарушения должны быть устранены до фиксации изменений.

Разумеется, другие рекомендации, такие как отказ от дублирования кода и предотвращение тесной связи, имеют столь же большое значение для обслуживания системы. Но как ответственный разработчик вы легко справитесь с применением первых трех ре-

комендаций в повседневной работе. Нарушения, касающиеся размеров, сложности и количества параметров блоков кода, легко обнаружить. Соответствующие средства проверки доступны во многих современных интегрированных средах разработки. Мы рекомендуем вам включить эту функцию и перед каждой отправкой кода в репозиторий проверять наличие грязи на уровне блоков.

### Правило 2: не оставляйте после себя неудачных комментариев

Некоторые комментарии считаются антишаблоном. Наш опыт позволяет утверждать, что встроенные комментарии обычно указывают на отсутствие в коде элегантных инженерных решений. Рассмотрим следующий метод, взятый из системы Jenkins:

```
public HttpResponse doUploadPlugin(StaplerRequest req)
    throws IOException, ServletException {
    try {
        Jenkins.getInstance().checkPermission(UPLOAD PLUGINS);

        ServletFileUpload upload = new ServletFileUpload(
            new DiskFileItemFactory());

        // Анализ запроса
        FileItem fileItem = (FileItem)upload.parseRequest(req).get(0);
        String fileName = Util.getFileName(fileItem.getName());
        if ("".equals(fileName)) {
            return new HttpRedirect("advanced");
        }
        // разрешено загружать файлы с новым расширением jpi
        // и устаревшим hpi
        if (!fileName.endsWith(".jpi") && !fileName.endsWith(".hpi")) {
            throw new Failure("Not a plugin: " + fileName);
        }

        // сначала скопировать во временный файл
        File t = File.createTempFile("uploaded", ".jpi");
        t.deleteOnExit();
        fileItem.write(t);
        fileItem.delete();

        final String baseName = identifyPluginShortName(t);

        pluginUploaded = true;
```

```
        // Теперь создать заготовку для последующей динамической загрузки
        // плагина (при необходимости будет выполнен принудительный
        // перезапуск InstallationJob):
        JSONObject cfg = new JSONObject().element("name", baseName)
            .element("version", "0"). // не используется, но требуется
            element("url", t.toURI().toString())
            .element("dependencies", new JSONArray());
        new UpdateSite(UpdateCenter.ID UPLOAD, null).new Plugin(
            UpdateCenter.ID UPLOAD, cfg).deploy(true);
        return new HttpRedirect("../updateCenter");
    } catch (IOException e) {
        throw e;
    } catch (Exception e) {// возбуждается методом fileItem.write
        throw new ServletException(e);
    }
}
```

Хотя метод `doUploadPlugin` не слишком сложно обслуживать (он имеет всего один параметр, содержит 31 строку кода и оценивается по Маккейбу индексом 6), но встроенные комментарии указывают на отдельные проблемы, которые легко решить выделением их из метода. Например, копирование файла `fileItem` во временный файл и создание заготовки плагина можно реализовать в виде отдельных методов (что упростит их тестирование и повторное использование).

Комментарии в коде позволяют выявить множество различных проблем:

- отсутствие понимания кода:

```
// Я не знаю, что здесь происходит, но когда я удаляю
// эту строку, появляется бесконечный цикл
```

- неправильное отслеживание ошибок:

```
// JIRA-1234: Исправлена ошибка при суммировании отрицательных
// чисел
```

- игнорирование соглашений или оснасток:

```
// CHECKSTYLE:OFF
// NOPMD
```

- добрые намерения:

```
// ДОДЕЛАТЬ: Надо ускорить этот метод
```

Действительно ценные комментарии встречаются довольно редко. Документирование прикладного программного интерфейса является именно таким случаем, но постарайтесь избегать надоедливых догматических комментариев. В целом лучший совет, который мы можем дать, — исключите комментарии из кода.

### Правило 3: не оставляйте после себя закомментированного кода

В редких случаях действительно бывает необходимо закомментировать тот или иной фрагмент кода, но отправке закомментированного кода в репозиторий нет оправданий. Система управления версиями сохраняет старый код, поэтому его можно смело удалить. Взгляните на следующий пример, взятый из базы кода Apache Tomcat (оригинальный код написан на Java, но мы переложили его на язык C# для удобства):

```
private void ValidateFilterMap(FilterMap filterMap) {
    // Проверить предложенный массив фильтров
    string filterName = filterMap.GetFilterName();
    string[] servletNames = filterMap.GetServletNames();
    string[] urlPatterns = filterMap.GetURLPatterns();
    if (FindFilterDef(filterName) == null)
        throw new Exception(
            sm.GetString("standardContext.filterMap.name", filterName));

    if (!filterMap.GetMatchAllServletNames() &&
        !filterMap.GetMatchAllUrlPatterns() &&
        (servletNames.Length == 0) && (urlPatterns.Length == 0))
        throw new Exception(
            sm.GetString("standardContext.filterMap.either"));
    // ИСПРАВИТЬ: Здесь используются старые спецификации
    /*
    if ((servletNames.length != 0) && (urlPatterns.length != 0))
        throw new IllegalArgumentException
            (sm.getString("standardContext.filterMap.either"));
    */
    for (int i = 0; i < urlPatterns.Length; i++) {
        if (!ValidateURLPattern(urlPatterns[i])) {
            throw new Exception(
                sm.GetString("standardContext.filterMap.pattern",
                    urlPatterns[i]));
        }
    }
```

```
}
```

Пометка ИСПРАВИТЬ и сопровождающий ее код понятны только оригинальному разработчику, для нового разработчика этот фрагмент станет лишь отвлекающим фактором. Оригинальный разработчик должен был принять решение, прежде чем оставить этот закомментированный код: исправить его на месте, создать новый запрос, чтобы исправить позднее, или вообще отказаться от этого варианта.

### Правило 4: не оставляйте после себя неиспользуемого кода

Неиспользуемый код имеет различные формы. Это код, который вообще никогда не выполнится или выполняется, но результат его работы нигде не используется. Закомментированный код в предыдущем разделе является примером неиспользуемого кода, но есть много других форм. Далее приводятся еще три примера неиспользуемого кода.

#### *Недостижимый код в методах*

```
public Transaction GetTransaction(long uid)
{
    Transaction result = new Transaction(uid);
    if (result != null)
    {
        return result;
    }
    else
    {
        return LookupTransaction(uid); ❶
    }
}
```

❶ Недостижимый код

#### *Неиспользуемые закрытые методы*

Закрытые методы могут вызываться только внутри своего класса. Если таких вызовов нет, они оказываются неиспользуемыми. Общедоступные методы, которые не вызываются в своем классе, также могут быть неиспользуемыми, но этого нельзя определить, просматривая только код класса.

***Код в комментариях***

Не путайте с закомментированным кодом. Иногда полезно использовать краткие фрагменты кода при документировании прикладных программных интерфейсов (например, теги XMLDOC C#), но помните, что сохранение этих фрагментов синхронными с актуальным кодом является задачей, о которой часто забывают. Избегайте кода в комментариях по мере возможности.

### Правило 5: не оставляйте после себя длинных идентификаторов

Хорошие идентификаторы отличают хороший код, который приятно читать, от плохого, в котором сложно что-либо понять. Знаменитая поговорка Фила Карлтона (Phil Karlton) гласит: «есть только две сложные проблемы: неактуальный кэш и именование сущностей». Первая проблема не рассматривается в этой книге, а о длинных идентификаторах хотелось бы сделать несколько замечаний.

Идентификаторы служат для именования элементов кода, от блоков до модулей, от компонентов до самой системы. Важно стараться подбирать хорошие имена, чтобы разработчики не затрачивали особых усилий на поиск в базе кода. Имена большинства идентификаторов в коде определяются предметной областью системы. Обычно в коллективах используются официальные или неофициальные (но согласованные) соглашения об именах, основанные на терминологии предметной области.

Подбор правильных идентификаторов — не самая простая задача, и, к сожалению, не существует четких правил, определяющих, что является хорошим идентификатором. Иногда приходится даже сделать несколько попыток, чтобы подобрать правильное имя для метода или класса.

Но, как бы то ни было, избегайте излишне длинных идентификаторов. Максимально допустимую длину идентификатора трудно определить (поскольку в некоторых предметных областях используются более длинные термины, чем в других), но в большинстве случаев в командах разработчиков не возникает споров о том, что считать слишком длинным идентификатором. Идентификаторы, которые явно описывают несколько функций (такие как

GenerateConsoleAnnotationScriptAndStylesheet) или содержат слишком много технических терминов (например, GlobalProjectNamingStrategyConfiguration), однозначно нарушают это правило.

### Правило 6: не оставляйте после себя таинственных констант

Таинственные константы — это числовые или другие литералы, используемые в коде без четкого определения их назначения (отсюда и название *«таинственные* константы»). Взгляните наследующий пример:

```
float CalculateFare(Customer c, long distance)
{
    float travelledDistanceFare = distance * 0.10f;
    if (c.Age < 12)
    {
        travelledDistanceFare *= 0.25f;
    }
    else
        if (c.Age >= 65)
    {
        travelledDistanceFare *= 0.5f;
    }
    return 3.00f + travelledDistanceFare;
}
```

Все числа в этом примере можно считать таинственными. Возрастные пороги для детей и пожилых людей еще кажутся понятными, но не забывайте, что они могут использоваться во многих других местах в базе кода. Ставки тарифов — это константы, которые могут меняться с течением времени согласно требованиям бизнеса.

Следующий фрагмент показывает, как мог бы выглядеть код, если все таинственные константы объявить явно. Объем кода увеличивается на шесть строк, что сравнимо с объемом оригинального исходного кода, но не забывайте, что эти константы могут повторно использоваться во многих других местах:

```
private static readonly float BASE RATE = 3.00f;
private static readonly float FARE PER KM = 0.10f;
private static readonly float DISCOUNT RATE CHILDREN = 0.25f;
private static readonly float DISCOUNT RATE ELDERLY = 0.5f;
private static readonly int MAXIMUM AGE CHILDREN = 12;
private static readonly int MINIMUM_AGE_ELDERLY = 65;
```

```
float CalculateFare(Customer c, long distance)
{
    float travelledDistanceFare = distance * FARE PER KM;
    if (c.Age < MAXIMUM AGE CHILDREN)
    {
        travelledDistanceFare *= DISCOUNT RATE CHILDREN;
    }
    else
        if (c.Age >= MINIMUM AGE ELDERLY)
    {
        travelledDistanceFare *= DISCOUNT RATE ELDERLY;
    }
    return BASE RATE + travelledDistanceFare;
}
```

### Правило 7: не оставляйте после себя плохую обработку исключений

Здесь приводятся три рекомендации по обработке исключений, потому что в нашей практике мы часто наблюдаем массу недостатков, связанных с этим.

- Всегда перехватывайте исключения. Регистрация сбоев системы помогает разобраться в них, а затем улучшить реакцию системы на них. Это означает, что исключения всегда должны перехватываться. В некоторых случаях разработчики используют пустые блоки `catch`, но это плохая практика, поскольку не позволяет получить сведения о контексте исключения.
- Перехватывайте конкретные исключения. Для анализа исключений, вызванных конкретными событиями, следует перехватывать конкретные исключения. Общие исключения, не имеющие сведений о состоянии системы или событиях, вызвавших их, не дают пищи для анализа. Поэтому не следует перехватывать исключения, такие как `Throwable`, `Exception` или `RuntimeException`.
- Преобразуйте конкретные исключения в информативные сообщения для вывода их перед конечными пользователями. Не «грузите» конечных пользователей подробными сведениями о возникшем исключении, поскольку их эта информация может только запутать, а ее вывод может нанести урон безопасности

системы (то есть не выводите сведений о внутренней работе системы больше, чем это необходимо).

## 11.3. Типичные возражения против написания чистого кода

Ниже обсуждаются типичные возражения против написания чистого кода. Самые распространенные из них основываются на том, что комментарии являются хорошим способом документирования кода, а редко возникающие ситуации можно не учитывать при обработке исключений.

### Возражение: комментарии являются документацией

*«Мы используем комментарии для документирования особенностей работы кода».*

Актуальные комментарии могут стать ценным подспорьем для неопытных разработчиков, пытающихся понять, как работает код. Но обычно в комментарии, конечно непреднамеренно, забывают внести изменения, и они перестают быть актуальными. С развитием системы устаревших комментариев становится все больше и больше. Синхронизация комментариев с кодом является отдельной задачей, требующей повышенной аккуратности, и об этом часто забывают при обслуживании.

Код, способный сам «рассказать» о себе, не требует пространных комментариев для пояснения его работы. Небольшие и простые блоки кода, использование понятных идентификаторов делают ненужным документирование кода с помощью комментариев.

### Возражение: обработка исключений увеличивает объем кода

*«Реализация классов исключений заставляет меня добавлять массу дополнительного кода без видимой на то необходимости».*

Обработка исключений — важная часть *защитного программирования*: методики предотвращения неустойчивых ситуаций и непредсказуемого поведения. Предвидение нестабильных ситуаций означает попытку предугадать, что может пойти не так. Но реализация такой обработки действительно требует определенных уси-

лий и дополнительного кода. Однако это оправданные затраты. Возможно, преимущества обработки исключений проявятся не сразу, но она, безусловно, будет полезна для предотвращения и смягчения последствий нестабильных ситуаций, могущих возникнуть в будущем.

Определяя исключения, вы документируете и фиксируете сделанные предположения. Позже, при изменении обстоятельств, их можно будет скорректировать.

### Возражение: почему выбраны именно эти правила?

*«В нашей команде используется гораздо более длинный список соглашений, касающихся программирования и контроля качества кода. Этот список из семи правил выглядит случайным их подмножеством с массой важных упущений».*

Расширить перечень из семи правил, представленных в этой главе, конечно же, не проблема. Но это именно те правила, которые мы сочли наиболее важными для написания простого в обслуживании кода и которые должны соблюдаться каждым членом команды разработчиков. При слишком большом количестве рекомендаций и проверок высок риск перегрузки разработчиков, что заставит их уделять наибольшее внимание второстепенным проблемам. Естественно, команды могут добавить в этот перечень любые правила, которые посчитают важными для улучшения обслуживаемости системы.

# Глава 12

# Дальнейшие действия

Теперь вы многое знаете об обслуживании кода, о его важности и приемах применения 10 рекомендаций, представленных в этой книге. Но написанию простого в обслуживании кода нельзя научиться, читая книги. Вы научитесь этому, когда начнете писать такой код! Поэтому сейчас рассмотрим несколько простых советов по применению на практике 10 рекомендаций, направленных на улучшение обслуживаемости программного обеспечения.

## 12.1. Применение рекомендаций на практике

Простота обслуживания кода зависит от двух обыденных вещей: *дисциплины* и *правильного выбора приоритетов.* Дисциплина помогает последовательно улучшать навыки программирования, пока вновь написанный код не будет соответствовать высокому уровню обслуживаемости. Что касается приоритетов, то некоторые из представленных рекомендаций могут вступать в конфликт друг с другом. Тогда вам придется самостоятельно решить, что оказывает наибольшее влияние на удобство обслуживания. Потратьте время на обсуждение этого вопроса и выслушайте мнение членов своей команды.

## 12.2. Низкоуровневые (блоки кода) рекомендации имеют более высокий приоритет, если они противоречат высокоуровневым (компоненты) рекомендациям

Имейте в виду, что на агрегатные высокоуровневые рекомендации оказывает влияние применение низкоуровневых рекомендаций. Например, большой размер блоков кода (глава 2) и дублирование (глава 4) наверняка приведут к увеличению объема базы кода (глава 9). Дублирование длинных блоков кода является одной из причин разрастания базы кода.

Поэтому, когда две рекомендации противоречат друг другу, для достижения лучшей обслуживаемости системы следует придерживаться низкоуровневой рекомендации. Например, деление блоков кода на несколько меньших по размеру увеличивает общий объем базы кода. Но преимущество возможности повторного использования небольших блоков кода сулит огромные выгоды при добавлении в систему новых функциональных возможностей.

То же относится к рекомендациям уровня архитектуры (главы 7 и 8): нет смысла заниматься реорганизацией структуры кода, когда компоненты тесно связаны друг с другом. Проще говоря: исключите тесные зависимости и только потом пытайтесь сбалансировать компоненты.

## 12.3. Помните, что учитывается каждое действие

Трудность применения рекомендаций из этой книги заключается в необходимости постоянного соблюдения дисциплины. Все время возникает соблазн нарушить рекомендации, когда «исправление побыстрому» кажется наиболее эффективным решением. Для поддержания дисциплины следуйте правилам бойскаутов, перечисленным в главе 11.

Следование правилам бойскаутов особенно эффективно при работе с базами кода большого размера. Если у вас не хватает времени, чтобы заняться улучшением обслуживаемости системы в

целом, попробуйте делать это шаг за шагом, при выполнении обыденной работы. При этом обслуживаемость будет постепенно улучшаться, а вы отточите навыки улучшения кода. Таким способом в долгосрочной перспективе вы освоите умение писать очень простые в обслуживании программы.

## 12.4. Передовой опыт разработки будет рассмотрен в следующей книге

Как упоминалось в предисловии, следующая книга в этой серии будет посвящена подробному описанию процесса разработки высококачественного программного обеспечения. В ней мы представим 10 рекомендаций, касающихся управления и оценки процесса разработки программного обеспечения. Основное внимание в ней будет уделено оценке и управлению с применением передового опыта разработки программного обеспечения (то есть инструментам разработки, автоматизации и стандартизации).

# Приложение А

# Как в компании SIG оценивается обслуживаемость

Обслуживаемость систем в компании SIG оценивается на основании восьми показателей. Эти восемь показателей были рассмотрены в главах со 2 по 9, где имеются врезки, поясняющие порядок определения рейтингов по свойствам исходного кода. Эти рейтинги выведены из критериев оценки уверенной обслуживаемости продуктов SIG/TÜViT[1]. В этом приложении будет предоставлена дополнительная информация по данным вопросам.

Совместно с TÜViT компания SIG определяет восемь свойств исходного кода, которые поддаются автоматической оценке. Во врезке «Почему именно эти десять конкретных рекомендаций?» (во вступлении) поясняются причины выбора этих свойств.

Для оценки удобства обслуживания выполняется измерение этих восьми свойств исходного кода системы, полученные значения суммируются и преобразуются в одно (например, процент повторяющегося кода) или в несколько чисел (например, процент кода, относящегося к четырем категориям сложности, называемых профилями качества, более подробно об этом рассказывается в разделе «Рейтинг обслуживаемости» в главе 1).

Затем полученные числа сравниваются с эталоном, построенным на основании анализа данных нескольких сотен систем, и для определения уровня качества каждого свойства используется табл. А.1. То есть если оценка попадает в верхние 5% всех систем, со-

[1] TÜViT является частью TÜV, всемирной организации контроля качества техники немецкого происхождения. Она специализируется на сертификации н консалтинге в области программного обеспечения, в частности в вопросах его безопасности.

бранных в эталоне, оцениваемая система получает рейтинг в 5 звезд по этому свойству. Если она оказывается в числе 30%, следующих за лучшими, ей присваивается рейтинг в 4 звезды, и так далее. В результате такого сравнения профилей качества системы с эталонными значениями получаются восемь оценок для каждого из свойств.

***Таблица А. 1 ❖ Рейтинги обслуживаемости SIG***

| Рейтинг | Обслуживаемость |
|---|---|
| 5 звезд | Лучшие 5% из эталонных систем |
| 4 звезды | Следующие 30% из эталонных систем (уровень выше среднего) |
| 3 звезды | Следующие 30% из эталонных систем (средний уровень) |
| 2 звезды | Следующие 30% из эталонных систем (уровень ниже среднего) |
| 1 звезда | 5% эталонных систем с худшей обслуживаемостью |

Затем путем агрегирования рейтингов рассчитывается общий рейтинг. Это делается в два этапа. Сначала определяются рейтинги субхарактеристик обслуживаемости, определяемых стандартом ISO 25010 (например, анализируемость, простота изменения и т. д.), с учетом средневзвешенных показателей в табл. А.2. Крестик в строке указывает, что соответствующее системное свойство (столбец) влияет на соответствующую субхарактеристику. Затем по средневзвешенным значениям пяти субхарактеристик определяется общий рейтинг обслуживаемости.

***Таблица А.2 ❖ Связь характеристик и свойств системы***

| | Объем | Дублирование | Размеры блоков кода | Сложность блоков кода | Интерфейсы блоков кода | Связанность модулей | Баланс компонентов | Независимость компонентов |
|---|---|---|---|---|---|---|---|---|
| Анализируемость | X | X | X | | | | X | |
| Обслуживаемость | | X | | X | | X | | |
| Тестируемость | X | | | X | | | | X |
| Модульность | | | | | | X | X | X |
| Повторное использование | | | X | | X | | | |

Здесь модель обслуживаемости, используемая в SIG, описана лишь в общих чертах, потому что детально рассмотреть ее в коротком приложении невозможно. Более подробные сведения можно найти в следующей публикации:

- *Visser J.* SIG/TÜViT Evaluation Criteria Trusted Product Maintainability // http://bit.ly/eval_criteria.

Справочная информация по разработке модели и ее применению приводится в следующих публикациях:

- *Heitlager I.*, *Kuipers T., Visser J.* A Practical Model for Measuring Maintainability. Proceedings of the 6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007). 30—39. IEEE Computer Society Press, 2007.
- *Baggen R., Correia J. P., Schill K., VisserJ.* Standardized code quality benchmarking for improving software maintainability // Software Quality Journal 20. № 2 (2012): 287—307.
- *Bijlsma D., Ferreira M. A., Luijten B., Visser J.* Faster issue resolution with higher technical quality of software // Software Quality Journal 20. № 2 (2012): 265—285.

---

**Улучшается ли обслуживаемость с течением времени?**

Нам в SIG часто задают вопрос: улучшается ли обслуживаемость анализируемых нами систем с течением времени. Ответ: да, но очень медленно. Проводимая нами ежегодно калибровка показывает, что пороговые значения становятся строже. Это означает, что для получения высокого рейтинга обслуживаемости система должна содержать меньшее количество слишком длинных и слишком сложных блоков кода, меньший процент повторяющегося кода, меньше тесных связей и т. д. Причина этого (с учетом структуры модели) заключается в том, что эталонные системы с течением времени содержат меньший процент повторяющегося кода, меньше тесных связей и т. д. Это позволяет утверждать, что обслуживаемость систем, выбранных в качестве эталонов, улучшается. Это не очень большие улучшения. Приближенно можно сказать, что за год добавляется по одной звезде в каждую десятую систему.

Системы, выбранные компанией SIG в качестве эталонных, являются представительными для всех сфер индустрии программного обеспечения, включая закрытые системы и системы с открытым исходным кодом, написанные на различных языках, для разных предметных областей, платформ и т. д. Таким образом, добавление одной звезды в год для каждой десятой системы означает, что отрасль в целом медленно, но постоянно улучшается.

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс»
наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: www.alians-kniga.ru.
Оптовые закупки: тел. (499) 782-38-89.
Электронный адрес: books@alians-kniga.ru.

Джуст Виссер

**Разработка обслуживаемых программ на языке C#**

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Редактор *Киселев А. Н.*
Перевод *Рагимов Р. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60x90 1/16.
Гарнитура «Петербург». Печать офсетная.
Уел. печ. л. 18. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

*«Эти рекомендации безукоризненны. Они представляют собой понятные и практичные правила, помогающие программисту писать качественный код.»*
***— Джордж Маринос,***
*Архитектор приложений из национального банка Греции*

Вы когда-нибудь ощущали разочарование, работая с чужим кодом? Сегодня трудности сопровождения исходного кода представляют важную проблему разработки программного обеспечения, приводящую к дорогостоящим срывам сроков и ошибкам. Подключайтесь к ее решению. Данное практическое руководство познакомит вас с 10 простыми рекомендациями, помогающими писать программное обеспечение, которое легко поддерживать и адаптировать. Эти тезисы сформулированы на основании анализа сотен реальных систем.

Написанная консультантами компании Software Improvement Group (SIG), книга содержит ясные и краткие советы по применению рекомендаций на практике. Примеры для этого издания написаны на языке C#, но существует аналогичная книга с примерами на языке Java.

- Пишите короткие блоки кода: ограничьте длину методов и конструкторов
- Пишите простые блоки кода: ограничьте число точек ветвления в методах
- Не повторяйте один и тот же код, избегайте риска внесения ошибок в повторяющийся код
- Стремитесь к уменьшению размеров интерфейсов, группируя параметры в объекты
- Разделяйте задачи, избегайте создания больших классов
- Избегайте тесных связей между компонентами архитектуры
- Сбалансируйте количество и размер компонентов верхнего уровня
- Стремитесь к уменьшению размера базы кода
- Автоматизируйте тестирование кода
- Пишите чистый код, избегайте «грязи» в коде, свидетельствующей о более глубоких проблемах

***Джуст Виссер** — научный руководитель компании SIG, отвечает за руководство уникальной сертифицированной лаборатории анализа программного обеспечения. Эта лаборатория производит стандартизированные исследования качества программных продуктов в соответствии с международным стандартом ISO 25010. Данная книга объединяет коллективные знания и опыт консультантов компании SIG, занимающихся оценкой и консультациями в области качества программного обеспечения с 2000 года.*

**Интернет-магазин:**
www.dmkpress.com
**Книга – почтой:**
orders@alians-kniga.ru
**Оптовая продажа:**
"Альянс-книга"
тел.(499)782-38-89
books@alians-kniga.ru



ISBN 978-5-97060-446-5