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 (Нидерданды). 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 рисками. 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.

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 представлен-

---

[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.

ных здесь рекомендаций. Например, инструменты Checkstyle (для Java, http://checkstyle.sourceforge.net/), StyleCop+ (для C#, http://stylecopplus.codeplex.com/), Pylint (для Python, http://www.pylint.org/), JSHint (для JavaScript, http://jshint.com/), RuboCop (для Ruby, https://github.com/bbatsov/rubocop) 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 автоматизации), 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 язы-

ках). 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 In-

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

formatica [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 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.

---

[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.*

«Clean Code: A Handbook of Agile Software Craftsmanship», автор Роберт С. Мартин (Robert C. Martin) (известный также как Uncle Bob)[1]

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

«Code Quality: The Open Source Perspective», автор Диомидис Спинеллис (Diomidis Spinellis)[2]

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))[3]

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.

---

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

[2] *Спинеллис Д.* Анализ программного кода на примере проектов Open Source. М.: Вильямс, 2014. ISBN: 5-8459-0604-0. – *Прим. ред.*

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


Так обозначаются советы или рекомендации.


Так обозначаются общие примечания.


Этим значком отмечаются важные замечания

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 обеспечения	(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 и 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 все 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 обслуживаемости, 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.

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 эксплуа-

[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.

тацию. 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 ис-

правление 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 обслуживаемость 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. Подразуме-

вается, 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 из-

менений 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 изменений 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.