ПИТЕР®

Robert C. Martin

Clean Code:

A Handbook of Agile Software Craftsmanship

PRENTICE
HALL

БИБЛИОТЕКА ПРОГРАММИСТА

Роберт Мартин

ЧИСТЫЙ КОД

Создание, анализ и рефакторинг

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2013

Роберт Мартин

Чистый код: создание, анализ и рефакторинг. Библиотека программиста

Перевел с английского Е. Матвеев

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Художественный редактор	*Л. Адуевская*
Корректор	*В. Листова*
Верстка	*Е. Егорова*

Мартин Р.

М29 Чистый код: создание, анализ и рефакторинг. Библиотека программиста. — СПб.: Питер, 2013. — 464 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-00487-9

Даже плохой программный код может работать. Однако если код не является «чистым», это всегда будет мешать развитию проекта и компании-разработчика, отнимая значительные ресурсы на его поддержку и «укрощение».

Эта книга посвящена хорошему программированию. Она полна реальных примеров кода. Мы будем рассматривать код с различных направлений: сверху вниз, снизу вверх и даже изнутри. Прочитав книгу, вы узнаете много нового о коде. Более того, вы научитесь отличать хороший код от плохого. Вы узнаете, как писать хороший код и как преобразовать плохой код в хороший.

Книга состоит из трех частей. В первой части излагаются принципы, паттерны и приемы написания чистого кода; приводится большой объем примеров кода. Вторая часть состоит из практических сценариев нарастающей сложности. Каждый сценарий представляет собой упражнение по чистке кода или преобразованию проблемного кода в код с меньшим количеством проблем. Третья часть книги — концентрированное выражение ее сути. Она состоит из одной главы с перечнем эвристических правил и «запахов кода», собранных во время анализа. Эта часть представляет собой базу знаний, описывающую наш путь мышления в процессе чтения, написания и чистки кода.

12+ (Для детей старше 12 лет. В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-0132350884 (англ.)
ISBN 978-5-496-00487-9

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 29.04.13. Формат 70x100/16. Усл. п. л. 37,410. Тираж 1000. Заказ
Отпечатано в полном соответствии с качеством предоставленных издательством материалов в ГППО «Псковская областная типография». 180004, Псков, ул. Ротная, 34

Содержание

Посвящается Анне-Марии —
бессмертной любви всей моей жизни

Предисловие

В Дании очень популярны леденцы Ga-Jol. Их сильный лакричный вкус отлично скрашивает нашу сырую и часто холодную погоду. Однако нас, датчан, леденцы Ga-Jol привлекают еще и мудрыми или остроумными высказываниями, напечатанными на крышке каждой коробки. Сегодня утром я купил две коробки леденцов и обнаружил на них старую датскую поговорку:

Ærlighed i små ting er ikke nogen lille ting.

«Честность в мелочах — вовсе не мелочь». Это было хорошим предзнаменованием, которое полностью соответствовало тому, о чем я собирался написать в предисловии. Мелочи важны. Эта книга посвящена вещам простым, но вовсе не малозначительным.

Бог скрывается в мелочах, сказал архитектор Людвиг Мис ван дер Роэ. Эта цитата напоминает о недавних дебатах о роли архитектуры в разработке программного обеспечения и особенно в мире гибких методологий. Мы с Бобом время от времени увлеченно вступаем в этот диалог. Да, Мис ван дер Роэ проявлял внимание и к удобству, и к неподвластным времени строительным формам, лежащим в основе великой архитектуры. С другой стороны, он также лично выбирал каждую дверную ручку для каждого спроектированного им дома. Почему? Да потому, что мелочи важны.

В наших с Бобом непрестанных «дебатах» о TDD выяснилось, что мы согласны с тем, что архитектура играет важную роль при разработке, хотя мы по-разному смотрим на то, какой смысл вкладывается в это утверждение. Впрочем, эти разногласия относительно несущественны, так как мы считаем само собой разумеющимся, что ответственные профессионалы выделяют *некоторое* время на обдумывание и планирование проекта. Появившиеся в конце 1990-х концепции проектирования, зависящего *только* от тестов и кода, давно прошли. Тем не менее внимание к мелочам является еще более важным аспектом профессионализма, чем любые грандиозные планы. Во-первых, благодаря практике в мелочах профессионалы приобретают квалификацию и репутацию для серьезных проектов. Во-вторых, даже мельчайшее проявление небрежности при строительстве — дверь, которая неплотно закрывается, или криво положенная плитка на полу, или даже захламленный стол — полностью рассеивает очарование всего

сооружения. Чтобы этого не происходило с вашими программами, код должен быть чистым.

Впрочем, архитектура — всего лишь одна из метафор для разработки программных продуктов. Она лучше всего подходит для проектов, в которых продукт «возводится» в том же смысле, в каком архитектор возводит строение. В эпоху Scrum и гибких методологий основное внимание уделяется быстрому выводу продукта на рынок. Фабрики, производящие программные продукты, должны работать на максимальной скорости. Однако этими «фабриками» являются живые люди: мыслящие, чувствующие программисты, работающие над пожеланиями пользователей или историей продукта для создания новых продуктов. Метафора производства сейчас как никогда сильна в их мировоззрениях. Скажем, методология Scrum во многом вдохновлена производственными аспектами японского автостроения с его конвейерами.

Но даже в автостроении основная часть работы связана не с производством, а с сопровождением продуктов — или его отсутствием. В программировании 80% и более того, что мы делаем, тоже изящно называется «сопровождением». На самом деле речь идет о починке. Наша работа ближе к работе домашних мастеров в строительной отрасли или автомехаников в области автостроения. Что японская теория управления говорит по этому поводу?

В 1951 году в японской промышленности появилась методология повышения качества, называвшаяся TPM (Total Productive Maintenance). Она была ориентирована прежде всего на сопровождение, а не на производство. Доктрина TPM базировалась на так называемых «принципах 5S». В сущности, принципы 5S представляют собой набор житейских правил. Кстати говоря, они также заложены в основу методологии Lean — другого модного течения на западной сцене, набирающего обороты и в программных кругах. Как указывает Дядюшка Боб в своем введении, хорошая практика программирования требует таких качеств, как сосредоточенность, присутствие духа и мышление. Проблемы не всегда решаются простым действием, максимальной загрузкой оборудования для производства в оптимальном темпе. Философия 5S состоит из следующих концепций:

- *Сэйри*, или организация. Абсолютно необходимо знать, где что находится — и в этом помогают такие методы, как грамотный выбор имен. Думаете, выбор имен идентификаторов неважен? Почитайте следующие главы.
- *Сэйтон*, или аккуратность. Старая американская поговорка гласит: всему свое место, и все оказывается на своих местах. Фрагмент кода должен находиться там, где читатель кода ожидает его найти, — а если он находится где-то в другом месте, переработайте свой код и разместите его там, где ему положено быть.
- *Сэйсо*, или чистка. Рабочее место должно быть свободно от висящих проводов, грязи, мусора и хлама. Что в этой книге говорят авторы о загромождении кода комментариями и закомментированными строками кода? Они советуют от них избавиться.

- *Сэйкэцу*, или стандартизация: группа достигает согласия по поводу того, как поддерживать чистоту на рабочем месте. Что в этой книге сказано о наличии единого стиля кодирования и набора правил в группах? Откуда берутся эти стандарты? Прочитайте — узнаете.
- *Сюцукэ*, или дисциплина. Программист должен быть достаточно дисциплинированным, чтобы следовать правилам, он должен часто размышлять о своей работе и быть готовым к изменениям.

Если вы не пожалеете усилий — да, усилий! — чтобы прочитать и применять эту книгу, вы научитесь понимать последний пункт. Мы наконец-то подошли к корням ответственного профессионализма в профессии, которая должна пристально интересоваться жизненным циклом продукта. В ходе сопровождения автомобилей и других машин по правилам TPM, аварийный ремонт (аналог проявления ошибок) является исключением. Вместо этого мы ежедневно осматриваем машины и заменяем изнашивающиеся части до того, как они сломаются, или выполняем аналоги знаменитой «смены масла каждые 10 000 миль» для предотвращения износа. Безжалостно перерабатывайте свой код. А еще можно сделать следующий шаг, который считался новаторским в движении TPM более 50 лет назад: строить машины, изначально ориентированные на удобство сопровождения. Ваш код должен не только работать, но и хорошо читаться. Как нас учит Фред Брукс, крупные блоки программного кода стоит переписывать «с нуля» каждые семь лет или около того, чтобы они не обрастали мхом. Но может быть, временную константу Брукса стоит вывести на уровень недель, дней и часов вместо годов. Именно на этом уровне живут мелочи.

В мелочах кроется огромная сила, но при этом такой подход к жизни выглядит скромно и основательно, как мы стереотипно ожидаем от любого метода с японскими корнями. Однако такой взгляд на жизнь не является чисто восточным; в западной народной мудрости можно найти немало наставлений такого рода. Цитата, приведенная ранее при описании принципа *сэйтон*, принадлежит перу министра из Огайо, который буквально рассматривал аккуратность «как средство от любого зла». Как насчет *сэйсо*? *Чистота ведет к Божественности*. Каким бы красивым ни был дом, захламленный стол портит все впечатление. А что говорят о *сюцукэ*? *Тот, кто верен в мелочах, верен во всем*. Стремление к переработке кода, укрепление позиций для последующих «серьезных» решений — вместо того, чтобы откладывать переработку «на потом»? *Ранняя пташка червяка ловит. Не откладывай на завтра то, что можно сделать сегодня*. (Фраза «последний ответственный момент» в методологии Lean имела именно такой смысл, пока не попала в руки консультантов по разработке ПО). Как насчет места малых, индивидуальных усилий в общей картине? *Из маленьких желудей вырастают большие дубы*. Интеграция простой профилактической работы в повседневную жизнь? *Яблочко на ужин, и доктор не нужен. Дорога ложка к обеду*. Чистый код уважает глубокие корни мудрости, лежащие в основе нашей культуры — той, которой она когда-то была или должна быть, и может быть при должном внимании к мелочам.

Даже в литературе по архитектуре мы находим фразы, возвращающие нас к важной роли мелочей. Вспомните дверные ручки ван дер Роэ. *Сэйри* в чистом виде. Внимание к имени каждой переменной. Имя переменной должно выбираться так же тщательно, как и имя новорожденного.

Как известно любому домовладельцу, такая забота и непрерывное стремление к улучшению никогда не приходят к концу. Архитектор Кристофер Александр — отец паттернов и языка паттернов — рассматривает каждый акт проектирования как маленький, локальный акт восстановления. С его точки зрения мастерство тонкой структуры является единственным содержанием архитектуры; более крупные формы можно оставить на долю паттернов, а их применение — на долю жильцов. Проектирование продолжается не только с пристройкой к дому новых комнат, но и с покраской, заменой старых ковров или кухонной раковины. Аналогичные принципы действуют во многих видах искусства. В поисках других мастеров, считавших, что Бог живет в мелочах, мы оказываемся в славной компании французского писателя XIX века Гюстава Флобера. Французский поэт Поль Валери говорит о том, что стихотворение никогда не бывает законченным, что оно требует постоянной переработки, а прекратить работу над ним — значит бросить его. Такое повышенное внимание к мелочам характерно для всех настоящих творцов. Возможно, принципиального нового здесь не так уж много, но эта книга напомнит вам о необходимости следовать житейским правилам, которые вы давно забросили из безразличия или стремления к стихийности, к простой «реакции на изменения».

К сожалению, описанные аспекты редко рассматриваются как краеугольные камни искусства программирования. Мы рано бросаем свой код — и не потому, что он идеален, а потому, что наша система ценностей сосредоточена на внешнем виде, а не на внутренней сущности того, что мы делаем. Невнимательность в конечном итоге обходится недешево: *фальшивая монета всегда возвращается к своему владельцу*. Исследования — ни отраслевые, ни академические — не желают опускаться до скромной области поддержания чистоты кода. В те времена, когда я работал в Исследовательской организации по производству программного обеспечения Bell Labs, в ходе исследований выяснилось, что последовательный стиль применения отступов является одним из самых статистически значимых признаков низкой плотности ошибок. Мы хотим, чтобы причиной качества была архитектура, язык программирования или что-то другое, столь же почтенное. Нас как людей, чей предполагаемый профессионализм обусловлен мастерским владением инструментами и методами проектирования, оскорбляет сама идея, что простое последовательное применение отступов может иметь такую ценность. Цитируя свою собственную книгу 17-летней давности, скажу, что такой стиль отличает совершенство от простой компетентности. Японское мировоззрение сознает критическую важность каждого рядового рабочего, и что еще важнее — систем разработки, существующих благодаря простым повседневным действиям этих рабочих. Качество возникает в результате миллиона проявлений небезразличного отношения к делу, — а не от применения какого-то великого метода,

спустившегося с небес. Простота этих проявлений не означает их примитивности и даже не свидетельствует об их легкости. Тем не менее из них возникает величие и, более того, — красота любого человеческого начинания. Забыть о них значит не быть человеком в полной мере.

Конечно, я по-прежнему выступаю за широту мышления и особенно за ценность архитектурных подходов, корни которых уходят в глубокое знание предметной области и удобство использования программных продуктов. Книга написана не об этом, или, по крайней мере, в ней эта тема не рассматривается напрямую. Она несет более тонкий посыл, глубину которого не стоит недооценивать. Она соответствует текущим мировоззрениям настоящих программистов — таких, как Питер Соммерлад (Peter Sommerlad), Кевлин Хенни (Kevlin Henney) и Джованни Аспрони (Giovanni Asproni). «Код есть архитектура» и «простой код» — так звучат их мантры. Хотя мы не должны забывать, что интерфейс и есть программа и что его структурные элементы несут много информации о структуре программы, очень важно помнить, что архитектура живет в коде. И если переработка в производственной метафоре ведет к затратам, переработка в архитектурной метафоре ведет к повышению ценности. Рассматривайте свой код как красивое воплощение благородных усилий по проектированию — как процесса, а не как статической конечной точки. Архитектурные метрики привязки и связности проявляются именно в коде. Если вы послушаете, как Ларри Константайн (Larry Constantine) описывает привязку и связность, он говорит о них в контексте кода, а не величественных абстрактных концепций, которые обычно встречаются в UML. Ричард Гэбриел (Richard Gabriel) в своем эссе «Abstraction Descant» утверждает, что абстракция — зло. Так вот, код — это антизло, а чистый код, вероятно, имеет божественную природу.

Возвращаясь к своему примеру с коробочкой Ga-Jol, подчеркну один важный момент: датская народная мудрость рекомендует нам не только обращать внимание на мелочи, но и быть *честными* в мелочах. Это означает честность в коде, честность с коллегами и, что самое важное, — честность перед самим собой по поводу состояния вашего кода. Действительно ли мы сделали все возможное для того, чтобы «оставить место лагеря чище, чем было до нашего прихода»? Переработали ли свой код перед тем, как сдавать его? Эти проблемы лежат в самом сердце системы ценностей Agile. Методология Scrum указывает, чтобы переработка кода должна стать частью концепции «готовности». Ни архитектура, ни чистый код не требуют от нас совершенства — просто будьте честны и делайте все, что можете. Человеку свойственно ошибаться; небесам свойственно прощать. В методологии Scrum все тайное становится явным. Мы выставляем напоказ свое грязное белье. Мы честно демонстрируем состояние нашего кода, а ведь код никогда не бывает идеальным. Мы становимся более человечными и приближаемая к величию в мелочах.

В нашей профессии нам отчаянно нужна вся помощь, которую мы можем получить. Если чистый пол в магазине сокращает вероятность несчастных случаев, а аккуратно разложенные инструменты повышают производительность, то я обе-

ими руками «за». Что касается этой книги, то она является лучшим практическим применением принципов Lean в области разработки программного обеспечения, которое я когда-либо видел в печатном виде. Впрочем, я иного и не ожидал от этой небольшой группы мыслящих личностей, которые в течение многих лет стремятся не только узнать что-то новое, но и делятся своими знаниями с нами в книгах, одну из которых вы сейчас держите в руках. Мир стал чуть более совершенным, чем был до того момента, когда Дядюшка Боб прислал мне рукопись.

Завершая свои высокопарные размышления, я отправляюсь наводить порядок на своем столе.

Джеймс О. Коплин
Мёрруп, Дания

Введение

С любезного разрешения Тома Холверда (Thom Holwerda)
(http://www.osnews.com/story/19266/WTFs_m)

Какая из двух дверей характерна для вашего кода? Какая дверь характерна для вашей группы или компании? Почему вы попали именно в эту комнату? В ней идет нормальный анализ кода или сразу же после выпуска программы обнаружился целый поток ужасных ошибок? Отладка идет в панике, вы просматриваете код, который, как считалось, уже работает? Клиенты уходят от вас целыми толпами, а начальство дышит в затылок? Как оказаться за *правильной* дверью, когда дела пойдут плохо? Ответ: *профессионализм*.

Профессионализм имеет две составляющие: знания и практический опыт. Вы должны узнать принципы, паттерны, приемы и эвристические правила, известные каждому профессионалу, а также «втереть» полученные знания в свои пальцы, глаза и внутренности усердной работой и практикой.

Я могу объяснить вам физику езды на велосипеде. В самом деле, классическая физика относительно прямолинейна. Сила тяжести, сила трения, ротационный момент, центр тяжести и т. д. — все это можно описать менее чем на одной странице уравнений. Этими формулами я докажу вам, что езда на велосипеде возможна, и предоставлю всю необходимую для этого информацию. Но когда вы впервые заберетесь на велосипед, вы все равно неизбежно упадете.

С программированием дело обстоит точно так же. Конечно, мы могли бы записать все «хорошие» принципы чистого кода, а потом доверить вам всю практическую работу (другими словами, позволить вам упасть, забравшись на велосипед), но какие бы из нас тогда были учителя?

Нет. В этой книге мы пойдем по другому пути.

Умение писать чистый код — *тяжелая работа*. Она не ограничивается знанием паттернов и принципов. Над кодом необходимо *попотеть*. Необходимо пытаться и терпеть неудачи. Необходимо наблюдать за тем, как другие пытаются и терпят неудачи. Необходимо видеть, как они спотыкаются и возвращаются к началу; как мучительно принимаются решения и какую цену приходится платить за неверный выбор.

Приготовьтесь основательно потрудиться во время чтения книги. Перед вами не «легкое чтиво», которое можно проглотить в самолете и перевернуть последнюю страницу перед посадкой. Книга заставит вас потрудиться, *и потрудиться усердно*. Какая работа вам предстоит? Вы будете читать код — много кода. И вам придется как следует подумать, что в этом коде правильно, а что нет. Вы будете наблюдать за тем, как мы разбираем эти модули, а потом собираем заново. Это потребует немало времени и усилий; но мы считаем, что результат того стоит.

Книга разделена на три части. В первых нескольких главах излагаются принципы, паттерны и приемы написания чистого кода. В них приводится довольно солидный объем кода, и читать их будет непросто. Весь этот материал подготовит вас ко второй части. Если вы отложите книгу после первой части — всего хорошего!

Во второй части книги трудиться придется еще больше. Она состоит из нескольких практических сценариев нарастающей сложности. Каждый сценарий представляет собой упражнение по чистке кода — или преобразовании проблемного кода в код с меньшим количеством проблем. Чтобы усвоить материал этой части, необходимо *основательно потрудиться*. Вам придется переключаться туда-сюда между текстом и листингами. Вам придется анализировать и разбирать код, с которым мы работаем, и осознать причину каждого вносимого изменения. Выделите на это время, потому что *работа займет не один день*.

Третья часть книги — концентрированное выражение ее сути. Она состоит из одной главы с перечнем эвристических правил и «запахов кода», собранных во

время анализа. В ходе очистки кода в практических сценариях мы документировали причину каждого выполняемого действия в виде эвристического правила или «запаха». Мы пытались понять нашу собственную реакцию на код в процессе его чтения и изменения; старались объяснить, почему мы чувствовали то, что чувствовали, или делали то, что делали. Результат представляет собой базу знаний, описывающую наш путь мышления в процессе чтения, написания и чистки кода.

Впрочем, без тщательного чтения всех практических сценариев из второй части книги пользы от базы знаний будет немного. В этих сценариях мы тщательно пометили каждое вносимое изменение ссылкой на соответствующее эвристическое правило. Ссылки заключаются в квадратные скобки и выглядят примерно так: **[H22]**. Это позволяет читателю видеть *контекст*, в котором применяются эвристики! Главная ценность заключается даже не в самих эвристиках, а связях между ними и конкретными решениями, принимаемыми в ходе чистки кода практических сценариев.

Чтобы помочь вам отслеживать эти связи, мы разместили в конце книги список перекрестных ссылок. В нем приведены номера страниц всех ссылок. По этому списку можно найти каждое место, в котором применялась та или иная эвристика.

Если вы прочитаете первую и третью часть, пропустив анализ практических сценариев, — считайте, что вы прочитали еще одну «легкую» книгу о написании качественного кода. Но если вы потратите время на проработку всех сценариев, проследите за каждым крошечным шагом, за каждым решением, если вы поставите себя на наше место и заставите себя думать в том же направлении, то ваше понимание этих принципов, паттернов, приемов и эвристик значительно углубится. Знания уже не будут «внешними». Они проникнут в ваши пальцы, глаза и сердце. Они станут частью вашей личности — как велосипед становится продолжением вашего тела, когда вы научитесь на нем ездить.

Благодарности

Я благодарю двух художников, Дженнифер Конке (Jeniffer Kohnke) и Анджелу Брукс (Angela Brooks). Дженнифер создала отличные остроумные рисунки в начале каждой главы, а также нарисовала портреты Кента Бека, Уорда Каннингема, Бьёрна Страуструпа, Рона Джеффриса, Грэди Буча, Дэйва Томаса, Майкла Физерса... и меня.

Анджела занималась рисунками, поясняющими материал внутри глав. За прошедшие годы она подготовила немало иллюстраций для моих книг, в том числе для книги «Agile Software Develpment: Principles, Patterns, and Practices». Кроме того, она мой первенец, и я ей горжусь.

1 Чистый код

Вы читаете эту книгу по двум причинам. Во-первых, вы программист. Во-вторых, вы хотите повысить свою квалификацию как программиста. Отлично. Хороших программистов не хватает.

Эта книга посвящена хорошему программированию. Она полна реальных примеров кода. Мы будем рассматривать код с направлений: сверху вниз, снизу

вверх, и даже изнутри. К последней странице книги вы узнаете много нового о коде. Более того, вы научитесь отличать хороший код от плохого. Вы узнаете, как писать хороший код и как преобразовать плохой код в хороший.

Да будет код

Возможно, кто-то скажет, что книга о коде отстала от времени — код сейчас уже не так актуален; вместо него внимание следует направить на модели и требования. Нам даже доводилось слышать мнение, что код как таковой скоро перестанет существовать. Что скоро весь код будет генерироваться, а не писаться вручную. Что программисты станут попросту не нужны, потому что бизнесмены будут генерировать программы по спецификациям.

Ерунда! Код никогда не исчезнет, потому что код представляет подробности требований. На определенном уровне эти подробности невозможно игнорировать или абстрагировать; их приходится определять. А когда требования определяются настолько подробно, чтобы они могли быть выполнены компьютером, это и есть программирование. А их определение есть код.

Вероятно, уровень абстракции наших языков продолжит расти. Я также ожидаю, что количество предметно-ориентированных языков продолжит расти. И это хорошо. Но код от этого существовать не перестанет. В самом деле, все определения, написанные на этих высокоуровневых, предметно-ориентированных языках, станут кодом! И этот код должен быть достаточно компактным, точным, формальным и подробным, чтобы компьютер мог понять и выполнить его.

Люди, полагающие, что код когда-нибудь исчезнет, напоминают математиков, которые надеются когда-нибудь обнаружить неформальную математическую дисциплину. Они надеются, что когда-нибудь будут построены машины, которые будут делать то, что мы хотим, а не то, что мы приказываем сделать. Такие машины должны понимать нас настолько хорошо, чтобы преобразовать набор нечетких потребностей в идеально выполняемые программы, точно отвечающие этим потребностям.

Но этого никогда не произойдет. Даже люди, со всей их интуицией и изобретательностью, не способны создавать успешные системы на основе туманных представлений своих клиентов. Если дисциплина определения требований нас чему-то научила, так это тому, что четко определенные требования так же формальны, как сам код, и могут использоваться как исполняемые тесты этого кода!

В сущности, код представляет собой язык, на котором в конечном итоге выражаются потребности. Мы можем создавать языки, близкие к потребностям. Мы можем создавать инструменты, помогающие нам обрабатывать и собирать эти потребности в формальные структуры. Но необходимая точность никогда не исчезнет — а следовательно, код останется всегда.

Плохой код

Недавно я читал предисловие к книге Кента Бека «Implementation Patterns» [Beck07]. Автор говорит: «...эта книга базируется на довольно непрочной предпосылке: что хороший код важен...» Непрочная предпосылка? Не согласен! На мой взгляд, эта предпосылка является одной из самых мощных, основополагающих и многогранных положений нашего ремесла (и я думаю, что Кенту это известно). Мы знаем, что хороший код важен, потому что нам приходилось так долго мириться с его отсутствием.

Одна компания в конце 80-х годов написала приложение-бестселлер. Приложение стало чрезвычайно популярным, многие профессионалы покупали и использовали его. Но потом циклы выпуска новых версий стали затягиваться. Ошибки не исправлялись между версиями. Время загрузки росло, а сбои происходили все чаще. Помню тот день, когда я в раздражении закрыл этот продукт и никогда не запускал его. Вскоре эта компания разорилась.

Два десятилетия спустя я встретил одного из работников той компании и спросил его, что же произошло. Ответ подтвердил мои опасения. Они торопились с выпуском продукта на рынок и не обращали внимания на качество кода. С добавлением новых возможностей код становился все хуже и хуже, пока в какой-то момент не вышел из-под контроля. *Плохой код привел к краху компании.*

Плохой код когда-нибудь мешал вашей работе? Любой сколько-нибудь опытный программист неоднократно попадал в подобную ситуацию. Мы продираемся через плохой код. Мы вязнем в хитросплетении ветвей, попадаем в скрытые ловушки. Мы с трудом прокладываем путь, надеясь получить хоть какую-нибудь подсказку, что же происходит в коде; но не видим вокруг себя ничего, кроме новых залежей невразумительного кода.

Конечно, плохой код мешал вашей работе. Почему же вы писали его? Пытались поскорее решить задачу? Торопились? Возможно. А может быть, вам казалось, что у вас нет времени качественно выполнить свою работу; что ваше начальство будет недовольно, если вы потратите время на чистку своего кода. А может, вы устали работать над программой и вам хотелось поскорее избавиться от нее. А может, вы посмотрели на список запланированных изменений и поняли, что вам необходимо поскорее «прикрутить» этот модуль, чтобы перейти к следующему. Такое бывало с каждым.

Каждый из нас смотрел на тот хаос, который он только что сотворил, и решал оставить его на завтра. Каждый с облегчением видел, что бестолковая программа работает, и решал, что рабочая мешанина — лучше, чем ничего. Каждый обещал

себе вернуться и почистить код... потом. Конечно, в те дни мы еще не знали закон Леблана: *потом равносильно никогда*.

Расплата за хаос

Если вы занимались программированием более двух-трех лет, вам наверняка доводилось вязнуть в чужом — или в своем собственном — беспорядочном ходе. Замедление может быть весьма значительным. За какие-нибудь год-два группы, очень быстро двигавшиеся вперед в самом начале проекта, начинают ползти со скоростью улитки. Каждое изменение, вносимое в код, нарушает работу кода в двух-трех местах. Ни одно изменение не проходит тривиально. Для каждого дополнения или модификации системы необходимо «понимать» все хитросплетения кода — чтобы в программе их стало еще больше. Со временем неразбериха разрастается настолько, что справиться с ней уже не удается. Выхода просто нет.

По мере накопления хаоса в коде производительность группы начинает снижаться, асимптотически приближаясь к нулю. В ходе снижения производительности начальство делает единственное, что оно может сделать: подключает к проекту новых работников в надежде повысить производительность. Но новички ничего не понимают в архитектуре системы. Они не знают, какие изменения соответствуют намерениям проектировщика, а какие им противоречат. Более того, они — и все остальные участники группы — находятся под страшным давлением со стороны начальства. В спешке они работают все небрежнее, отчего производительность только продолжает падать (рис. 1.1).

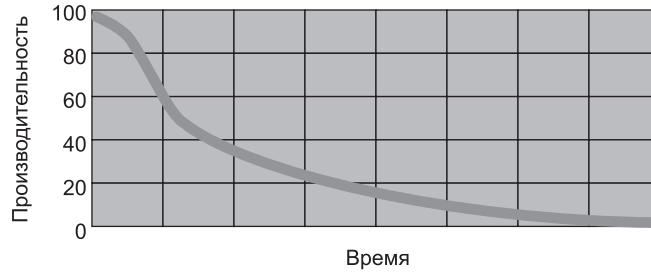


Рис. 1.1. Зависимость производительности от времени

Грандиозная переработка

В конечном итоге группа устраивает бунт. Она сообщает начальству, что не может продолжать разработку отвратительной кодовой базы, и требует переработки архитектуры. Начальство не хочет тратить ресурсы на полную переработку проекта, но не может отрицать, что производительность просто ужасна. Со временем

начальство поддается на требования разработчиков и дает разрешение на проведение грандиозной переработки.

Набирается новая «ударная группа». Все хотят в ней участвовать, потому что проект начинается «с нуля». Разработчики будут строить «на пустом месте», и создадут нечто воистину прекрасное. Но в «ударную группу» отбирают только самых лучших и умных. Всем остальным приходится сопровождать текущую систему.

Между двумя группами начинается гонка. «Ударная группа» должна построить новую систему, которая делает то же самое, что делала старая. Более того, она должна своевременно учитывать изменения, непрерывно вносимые в старую систему. Начальство не заменяет старую систему до тех пор, пока новая система не будет полностью повторять ее функциональность.

Такая гонка может продолжаться очень долго. Мне известны случаи, как она продолжалась по 10 лет. И к моменту ее завершения оказывалось, что исходный состав давно покинул «ударную группу», а текущие участники требовали переработать новую систему, потому что в ней творился сущий хаос.

Если вы сталкивались хотя бы с некоторыми частями истории, которую я сейчас поведал, то вы уже знаете, что поддержание чистоты кода не только окупает затраченное время; оно является делом профессионального выживания.

Отношение

Вам доводилось продираться через код настолько запутанный, что у вас уходили недели на то, что должно было занять несколько часов? Вы видели, как изменение, которое вроде бы должно вноситься в одной строке, приходится вносить в сотнях разных модулей? Эти симптомы стали слишком привычными.

Почему это происходит с кодом? Почему хороший код так быстро загнивает и превращается в плохой код? У нас обычно находится масса объяснений. Мы жалуемся на изменения в требованиях, противоречащие исходной архитектуре. Мы стенаем о графиках, слишком жестких для того, чтобы делать все, как положено. Мы сплетничаем о глупом начальстве, нетерпимых клиентах и бестолковых типах из отдела маркетинга. Однако вина лежит вовсе не на них, а на нас самих. Дело в нашем непрофессионализме.

Возможно, проглотить эту горькую пилюлю будет непросто. Разве мы виноваты в этом хаосе? А как же требования? График? Глупое начальство и бестолковые типы из отдела маркетинга? Разве по крайней мере часть вины не лежит на них?

Нет. Начальство и маркетологи обращаются к нам за информацией, на основании которой они выдвигают свои обещания и обязательства; но даже если они к нам не обращаются, мы не должны бояться говорить им то, что мы думаем. Пользователи обращаются к нам, чтобы мы высказали свое мнение относительно того, насколько уместны требования в системе. Руководители проектов обращаются к нам за помощью в составлении графика. Мы принимаем самое деятельное

участие в планировании проекта и несем значительную долю ответственности за любые провалы; особенно если эти провалы обусловлены плохим кодом!

«Но постойте! — скажете вы. — Если я не сделаю то, что говорит мой начальник, меня уволят». Скорее всего, нет. Обычно начальники хотят знать правду, даже если по их поведению этого не скажешь. Начальники хотят видеть хороший код, даже если они помешаны на рабочем графике. Они могут страстно защищать график и требования; но это их работа. А ваша работа — так же страстно защищать код.

Чтобы стало понятнее, представьте, что вы — врач, а ваш пациент требует прекратить дурацкое мытье рук при подготовке к операции, потому что это занимает слишком много времени[1]! Естественно, пациент — это ваш начальник; и все же врач должен наотрез отказаться подчиниться его требованиям. Почему? Потому что врач знает об опасности заражения больше, чем пациент. Было бы слишком непрофессионально (а то и преступно) подчиниться воле пациента.

Таким образом, программист, который подчиняется воле начальника, не понимающего опасность некачественного кода, проявляет непрофессионализм.

Основной парадокс

Программисты сталкиваются с основным парадоксом базовых ценностей. Каждый разработчик, имеющий сколько-нибудь значительный опыт работы, знает, что предыдущий беспорядок замедляет его работу. Но при этом все разработчики под давлением творят беспорядок в своем коде для соблюдения графика. Короче, у них нет времени, чтобы работать быстро!

Настоящие профессионалы знают, что вторая половина этого парадокса неверна. Невозможно выдержать график, устроив беспорядок. На самом деле этот беспорядок сразу же замедлит вашу работу, и график будет сорван. Единственный способ выдержать график — и единственный способ работать быстро — заключается в том, чтобы постоянно поддерживать чистоту в коде.

Искусство чистого кода?

Допустим, вы согласились с тем, что беспорядок в коде замедляет вашу работу. Допустим, вы согласились, что для быстрой работы необходимо соблюдать чистоту. Тогда вы должны спросить себя: «А как мне написать чистый код?» Бесполезно пытаться написать чистый код, если вы не знаете, что это такое!

К сожалению, написание чистого кода имеет много общего с живописью. Как правило, мы способны отличить хорошую картину от плохой, но это еще не значит,

[1] Когда Игнац Земмельвейс в 1847 году впервые порекомендовал врачам мыть руки перед осмотром пациентов, его советы были отвергнуты на том основании, что у врачей слишком много работы и на мытье рук у них нет времени.

что мы умеем рисовать. Таким образом, умение отличать чистый код от грязного еще не означает, что вы умеете писать чистый код!

Чтобы написать чистый код, необходимо сознательно применять множество приемов, руководствуясь приобретенным усердным трудом чувством «чистоты». Ключевую роль здесь играет «чувство кода». Одни с этим чувством рождаются. Другие работают, чтобы развить его. Это чувство не только позволяет отличить хороший код от плохого, но и демонстрирует стратегию применения наших навыков для преобразования плохого кода в чистый код.

Программист без «чувства кода» посмотрит на грязный модуль и распознает беспорядок, но понятия не имеет, что с ним делать. Программист с «чувством кода» смотрит на грязный модуль и видит различные варианты и возможности. «Чувство кода» поможет ему выбрать лучший вариант и спланировать последовательность преобразований, сохраняющих поведение программы и приводящих к нужному результату.

Короче говоря, программист, пишущий чистый код, — это художник, который проводит пустой экран через серию преобразований, пока он не превратится в элегантно запрограммированную систему.

Что такое «чистый код»?

Вероятно, сколько существует программистов, столько найдется и определений. Поэтому я спросил у некоторых известных, чрезвычайно опытных программистов, что они думают по этому поводу.

БЬЁРН СТРАУСТРУП, СОЗДАТЕЛЬ C++ И АВТОР КНИГИ «THE C++ PROGRAMMING LANGUAGE»

Я люблю, чтобы мой код был элегантным и эффективным. Логика должны быть достаточно прямолинейной, чтобы ошибкам было трудно спрятаться; зависимости — минимальными, чтобы упростить сопровождение; обработка ошибок — полной в соответствии с выработанной стратегией; а производительность — близкой к оптимальной, чтобы не искушать людей загрязнять код беспринципными оптимизациями. Чистый код хорошо решает одну задачу.

Бьёрн использует слово «элегантный». Хорошее слово! Словарь в моем MacBook® выдает следующие определения: доставляющий удовольствие своим изяществом и стилем; сочетающий простоту с изобретательностью. Обратите внимание на оборот «доставляющий удовольствие». Очевидно, Бьёрн считает,

что чистый код приятно читать. При чтении чистого кода вы улыбаетесь, как при виде искусно сделанной музыкальной шкатулки или хорошо сконструированной машины.

Бьёрн также упоминает об эффективности — притом дважды. Наверное, никого не удивят эти слова, произнесенные изобретателем C++, но я думаю, что здесь кроется нечто большее, чем простое стремление к скорости. Напрасные траты процессорного времени неэлегантны, они не радуют глаз. Также обратите внимание на слово «искушение», которым Бьёрн описывает последствия неэлегантности. В этом кроется глубокая истина. Плохой код *искушает*, способствуя увеличению беспорядка! Когда другие программисты изменяют плохой код, они обычно делают его еще хуже.

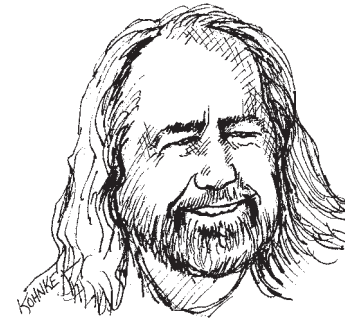
Прагматичные Дэйв Томас (Dave Thomas) и Энди Хант (Andy Hunt) высказали ту же мысль несколько иначе. Они сравнили плохой код с разбитыми окнами[1]. Здание с разбитыми окнами выглядит так, словно никому до него нет дела. Поэтому люди тоже перестают обращать на него внимание. Они равнодушно смотрят, как на доме появляются новые разбитые окна, а со временем начинают сами бить их. Они уродуют фасад дома надписями и устраивают мусорную свалку. Одно разбитое окно стало началом процесса разложения.

Бьёрн также упоминает о необходимости полной обработки ошибок. Это одно из проявлений внимания к мелочам. Упрощенная обработка ошибок — всего лишь одна из областей, в которых программисты пренебрегают мелочами. Утечка — другая область, состояния гонки — третья, непоследовательный выбор имен — четвертая... Суть в том, что чистый код уделяет пристальное внимание мелочам.

В завершение Бьёрн говорит о том, что чистый код хорошо решает одну задачу. Не случайно многие принципы проектирования программного обеспечения сводятся к этому простому наставлению. Писатели один за другим пытаются донести эту мысль. Плохой код пытается сделать слишком много всего, для него характерны неясные намерения и неоднозначность целей. Для чистого кода характерна целенаправленность. Каждая функция, каждый класс, каждый модуль фокусируются на конкретной цели, не отвлекаются от нее и не загрязняются окружающими подробностями.

ГРЭДИ БУЧ, АВТОР КНИГИ «OBJECT ORIENTED ANALYSIS AND DESIGN WITH APPLICATIONS»

> Чистый код прост и прямолинеен. Чистый код читается, как хорошо написанная проза. Чистый код никогда не затемняет намерения проектировщика; он полон четких абстракций и простых линий передачи управления.



[1] http://www.pragmaticprogrammer.com/booksellers/2004-12.html.

Грэди частично говорит о том же, о чем говорил Бьёрн, но с точки зрения *удобочитаемости*. Мне особенно нравится его замечание о том, что чистый код должен читаться, как хорошо написанная проза. Вспомните какую-нибудь хорошую книгу, которую вы читали. Вспомните, как слова словно исчезали, заменяясь зрительными образами! Как кино, верно? Лучше! Вы словно видели персонажей, слышали звуки, испытывали душевное волнение и сопереживали героям.

Конечно, чтение чистого кода никогда не сравнится с чтением «Властелина колец». И все же литературная метафора в данном случае вполне уместна. Чистый код, как и хорошая повесть, должен наглядно раскрыть интригу решаемой задачи. Он должен довести эту интригу до высшей точки, чтобы потом читатель воскликнул: «Ага! Ну конечно!», когда все вопросы и противоречия благополучно разрешатся в откровении очевидного решения.

На мой взгляд, использованный Грэди оборот «четкая абстракция» представляет собой очаровательный оксюморон! В конце концов, слово «четкий» почти всегда является синонимом для слова «конкретный». В словаре моего MacBook приведено следующее определение слова «четкий»: *краткий, решительный, фактический, без колебаний или лишних подробностей*. Несмотря на кажущееся смысловое противоречие, эти слова несут мощный информационный посыл. Наш код должен быть фактическим, а не умозрительным. Он должен содержать только то, что необходимо. Читатель должен видеть за кодом нашу решительность.

«БОЛЬШОЙ» ДЭЙВ ТОМАС, ОСНОВАТЕЛЬ OTI, КРЕСТНЫЙ ОТЕЦ СТРАТЕГИИ ECLIPSE

Чистый код может читаться и усовершенствоваться другими разработчиками, кроме его исходного автора. Для него написаны модульные и приемочные тесты. В чистом коде используются содержательные имена. Для выполнения одной операции в нем используется один путь (вместо нескольких разных). Чистый код обладает минимальными зависимостями, которые явно определены, и четким, минимальным API. Код должен быть грамотным, потому что в зависимости от языка не вся необходимая информация может быть четко выражена в самом коде.

Большой Дэйв разделяет стремление Грэди к удобочитаемости, но с одной важной особенностью. Дэйв утверждает, что чистота кода упрощает его доработку другими людьми. На первый взгляд это утверждение кажется очевидным, но его важность трудно переоценить. В конце концов, код, который легко читается, и код, который легко изменяется, — не одно и то же.

Дэйв связывает чистоту с тестами! Десять лет назад это вызвало бы множество недоуменных взглядов. Однако методология разработки через тестирование оказала огромное влияние на нашу отрасль и стала одной из самых фундамен-

тальных дисциплин. Дэйв прав. Код без тестов не может быть назван чистым, каким бы элегантным он ни был и как бы хорошо он ни читался.

Дэйв использует слово «минимальный» дважды. Очевидно, он отдает предпочтение компактному коду перед объемистым кодом. В самом деле, это положение постоянно повторяется в литературе по программированию от начала ее существования. Чем меньше, тем лучше.

Дэйв также говорил, что код должен быть *грамотным*. Это ненавязчивая ссылка на концепцию «грамотного программирования» Дональда Кнута [Knuth92]. Итак, код должен быть написан в такой форме, чтобы он хорошо читался людьми.

МАЙКЛ ФИЗЕРС, АВТОР КНИГИ «WORKING EFFECTIVELY WITH LEGACY CODE»

Я мог бы перечислить все признаки, присущие чистому коду, но существует один важнейший признак, из которого следуют все остальные. Чистый код всегда выглядит так, словно его автор над ним тщательно потрудился. Вы не найдете никаких очевидных возможностей для его улучшения. Все они уже были продуманы автором кода. Попытавшись представить возможные усовершенствования, вы снова придете к тому, с чего все началось: вы рассматриваете код, тщательно продуманный и написанный настоящим мастером, небезразличным к своему ремеслу.

Всего одно слово: тщательность. На самом деле оно составляет тему этой книги. Возможно, ее название стоило снабдить подзаголовком: «Как тщательно работать над кодом».

Майкл попал в самую точку. Чистый код — это код, над которым тщательно поработали. Кто-то не пожалел своего времени, чтобы сделать его простым и стройным. Кто-то уделил должное внимание всем мелочам и относился к коду с душой.

РОН ДЖЕФФРИС, АВТОР КНИГ «EXTREME PROGRAMMING INSTALLED» И «EXTREME PROGRAMMING ADVENTURES IN C#»

Карьера Рона началась с программирования на языке Fortran. С тех пор он писал код практически на всех языках и на всех компьютерах. К его словам стоит прислушаться.

За последние коды я постоянно руководствуюсь «правилами простого кода», сформулированными Беком. В порядке важности, простой код:

— проходит все тесты;

— не содержит дубликатов;

— выражает все концепции проектирования, заложенные в систему;

— содержит минимальное количество сущностей: классов, методов, функций и т. д.

Из всех правил я уделяю основное внимание дублированию. Если что-то делается в программе снова и снова, это свидетельствует о том, что какая-то мысленная концепция не нашла представления в коде. Я пытаюсь понять, что это такое, а затем пытаюсь выразить идею более четко.

Выразительность для меня прежде всего означает содержательность имен. Обычно я провожу переименования по несколько раз, пока не остановлюсь на окончательном варианте. В современных средах программирования — таких, как Eclipse — переименование выполняется легко, поэтому изменения меня не беспокоят. Впрочем, выразительность не ограничивается одними лишь именами. Я также смотрю, не выполняет ли объект или метод более одной операции. Если это объект, то его, вероятно, стоит разбить на два и более объекта. Если это метод, я всегда применяю к нему прием «извлечения метода»; в итоге у меня остается основной метод, который более четко объясняет, что он делает, и несколько подметодов, объясняющих, как он это делает.

Отсутствие дублирования и выразительности являются важнейшими составляющими чистого кода в моем понимании. Даже если при улучшении грязного кода вы будете руководствоваться только этими двумя целями, разница в качестве кода может быть огромной. Однако существует еще одна цель, о которой я также постоянно помню, хотя объяснить ее будет несколько сложнее.

После многолетней работы мне кажется, что все программы состоят из очень похожих элементов. Для примера возьмем операцию «найти элемент в коллекции». Независимо от того, работаем ли мы с базой данных, содержащий информацию о работниках, или хеш-таблицей с парами «ключ-значение», или массивом с однотипными объектами, на практике часто возникает задача извлечь конкретный элемент из этой коллекции. В подобных ситуациях я часто инкапсулирую конкретную реализацию в более абстрактном методе или классе. Это открывает пару интересных возможностей.

Я могу определить для нужной функциональности какую-нибудь простую реализацию (например, хеш-таблицу), но поскольку все ссылки прикрыты моей маленькой абстракцией, реализацию можно в любой момент изменить. Я могу быстро двигаться вперед, сохраняя возможность внести изменения позднее.

Кроме того, абстракция часто привлекает мое внимание к тому, что же «действительно» происходит в программе, и удерживает меня от реализации поведения коллекций там, где в действительности достаточно более простых способов получения нужной информации. Сокращение дублирования, высокая выразительность и раннее построение простых абстракций. Все это составляет чистый код в моем понимании.

В нескольких коротких абзацах Рон представил сводку содержимого этой книги. Устранение дублирования, выполнение одной операции, выразительность, простые абстракции. Все на месте.

УОРД КАННИНГЕМ, СОЗДАТЕЛЬ WIKI, СОЗДАТЕЛЬ FIT, ОДИН ИЗ СОЗДАТЕЛЕЙ ЭКСТРЕМАЛЬНОГО ПРОГРАММИРОВАНИЯ. ВДОХНОВИТЕЛЬ НАПИСАНИЯ КНИГИ «DESIGN PATTERNS». ДУХОВНЫЙ ЛИДЕР SMALLTALK И ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДА. КРЕСТНЫЙ ОТЕЦ ВСЕХ, КТО ТЩАТЕЛЬНО ОТНОСИТСЯ К НАПИСАНИЮ КОДА.

> Вы работаете с чистым кодом, если каждая функция делает примерно то, что вы ожидали. Код можно назвать красивым, если у вас также создается впечатление, что язык был создан специально для этой задачи.

Подобные заявления — отличительная способность Уорда. Вы читаете их, киваете головой и переходите к следующей теме. Это звучит настолько разумно, настолько очевидно, что не выглядит чем-то глубоким и мудрым. Вроде бы все само собой разумеется. Но давайте присмотримся повнимательнее.

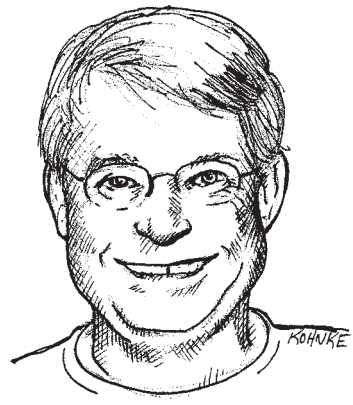
«...примерно то, что вы ожидали». Когда вы в последний раз видели модуль, который делал примерно то, что вы ожидали? Почему попадающиеся нам модули выглядят сложными, запутанными, приводят в замешательство? Разве они не нарушают это правило? Как часто вы безуспешно пытались понять логику всей системы и проследить ее в том модуле, который вы сейчас читаете? Когда в последний раз при чтении кода вы кивали головой так, как при очевидном заявлении Уорда?

Уорд считает, что чтение чистого кода вас совершенно не удивит. В самом деле, оно даже не потребует от вас особых усилий. Вы читаете код, и он делает примерно то, что вы ожидали. Чистый код очевиден, прост и привлекателен. Каждый модуль создает условия для следующего. Каждый модуль показывает, как будет написан следующий модуль. Чистые программы написаны настолько хорошо, что вы этого даже не замечаете. Благодаря автору код выглядит до смешного простым, как и все действительно выдающиеся творения.

А как насчет представления Уорда о красоте? Все мы жаловались на языки, не предназначенные для решения наших задач. Однако утверждение Уорда возлагает ответственность на нас. Он говорит, что при чтении красивого кода язык кажется созданным для решения конкретной задачи! Следовательно, мы сами должны позаботиться о том, чтобы язык казался простым! Языковые фанатики, задумайтесь! Не язык делает программы простыми. Программа выглядит простой благодаря работе программиста!

Школы мысли

А как насчет меня (Дядюшка Боб)? Что я думаю по поводу чистого кода? Эта книга расскажет вам во всех подробностях, что я и мои соратники думаем о чистом коде. Вы узнаете, как, по нашему мнению, должно выглядит чистое имя переменной, чистая функция, чистый класс и т. д. Мы излагаем свои мнения в виде беспрекословных истин и не извиняемся за свою категоричность. Для нас, на данном моменте наших карьер, они *являются* беспрекословными истинами. Они составляют нашу *школу мысли* в области чистого кода.



Мастера боевых искусств не достигли единого мнения по поводу того, какой из видов единоборств является лучшим, а какие приемы — самыми эффективными. Часто ведущие мастера создают собственную школу и набирают учеников. Так появилась школа дзю-дзюцу Грейси, основанная семьей Грейси в Бразилии. Так появилась школа дзю-дзюцу Хаккорю, основанная Окуямой Рюхо в Токио. Так появилась школа Джит Кун-до, основанная Брюсом Ли в Соединенных Штатах.

Ученики этих разных школ погружаются в учение основателя школы. Они посвящают себя изучению того, чему учит конкретный мастер, часто отказываясь от учений других мастеров. Позднее, когда уровень их мастерства возрастет, они могут стать учениками другого мастера, чтобы расширить свои познания и проверить их на практике. Некоторые переходят к совершенствованию своих навыков, открывают новые приемы и открывают собственные школы.

Ни одна из этих разных школ не обладает *абсолютной истиной*. Тем не менее в рамках конкретной школы мы действуем так, будто ее учение и арсенал приемов верны. Именно так и следует тренироваться в школе Хаккорю или Джит Кун-до. Но правильность принципов в пределах одной школы не делает ошибочными учения других школ.

Считайте, что эта книга является описанием Школы учителей Чистого кода. В ней представлены те методы и приемы, которыми мы сами пользуемся в своем искусстве. Мы утверждаем, что если вы последуете нашему учению, то это принесет вам такую же пользу, как и нам, и вы научитесь писать чистый и профессиональный код. Но не стоит думать, что наше учение «истинно» в каком-то абсолютном смысле. Существуют другие школы и мастера, которые имеют ничуть не меньше оснований претендовать на профессионализм. Не упускайте возможности учиться у них.

В самом деле, многие рекомендации в этой книге противоречивы. Вероятно, вы согласитесь не со всеми из них. Возможно, против некоторых вы будете яростно протестовать. Это нормально. Мы не претендуем на абсолютную истину. С дру-

гой стороны, приведенные в книге рекомендации являются плодами долгих, непростых размышлений. Мы пришли к ним после десятилетий практической работы, непрестанных проб и ошибок. Независимо от того, согласитесь вы с нами или нет, нашу точку зрения стоит по крайней мере узнать и уважать.

Мы — авторы

Поле `@author` комментария javadoc говорит о том, кто мы такие. Мы — авторы. А как известно, у каждого автора имеются свои читатели. Автор несет ответственность за то, чтобы хорошо изложить свои мысли читателям. Когда вы в следующий раз напишете строку кода, вспомните, что вы — автор, и пишете для читателей, которые будут оценивать плоды вашей работы.

Кто-то спросит: так ли уж часто читается наш код? Разве большая часть времени не уходит на его написание?

Вам когда-нибудь доводилось воспроизводить запись сеанса редактирования? В 80-х и 90-х годах существовали редакторы, записывавшие все нажатия клавиш (например, Emacs). Вы могли проработать целый час, а потом воспроизвести весь сеанс, словно ускоренное кино. Когда я это делал, результаты оказывались просто потрясающими. Большинство операций относилось к прокрутке и переходу к другим модулям!

Боб открывает модуль.

Он находит функцию, которую необходимо изменить.

Задумывается о последствиях.

Ой, теперь он переходит в начало модуля, чтобы проверить инициализацию переменной.

Снова возвращается вниз и начинает вводить код.

Стирает то, что только что ввел.

Вводит заново.

Еще раз стирает!

Вводит половину чего-то другого, но стирает и это!

Прокручивает модуль к другой функции, которая вызывает изменяемую функцию, чтобы посмотреть, как она вызывается.

Возвращается обратно и восстанавливает только что стертый код.

Задумывается.

Снова стирает!

Открывает другое окно и просматривает код субкласса. Переопределяется ли в нем эта функция?

. . .

В общем, вы поняли. На самом деле соотношение времени чтения и написания кода превышает 10:1. Мы постоянно читаем свой старый код, поскольку это необходимо для написания нового кода.

Из-за столь высокого соотношения наш код должен легко читаться, даже если это затрудняет его написание. Конечно, написать код, не прочитав его, невозможно, так что упрощение чтения в действительности упрощает и написание кода.

Уйти от этой логики невозможно. Невозможно написать код без предварительного чтения окружающего кода. Код, который вы собираетесь написать сегодня, будет легко или тяжело читаться в зависимости от того, насколько легко или тяжело читается окружающий код. Если вы хотите быстро справиться со своей задачей, если вы хотите, чтобы ваш код было легко писать — позаботьтесь о том, чтобы он легко читался.

Правило бойскаута

Хорошо написать код недостаточно. Необходимо поддерживать чистоту кода с течением времени. Все мы видели, как код загнивает и деградирует с течением времени. Значит, мы должны активно поработать над тем, чтобы этого не произошло.

У бойскаутов существует простое правило, которое применимо и к нашей профессии:

Оставь место стоянки чище, чем оно было до твоего прихода[1].

Если мы все будем оставлять свой код чище, чем он был до нашего прихода, то код попросту не будет загнивать. Чистка не обязана быть глобальной. Присвойте более понятное имя переменной, разбейте слишком большую функцию, устраните одно незначительное повторение, почистите сложную цепочку `if`.

Представляете себе работу над проектом, код которого *улучшается* с течением времени? Но может ли профессионал позволить себе нечто иное? Разве постоянное совершенствование не является неотъемлемой частью профессионализма?

Предыстория и принципы

Эта книга во многих отношениях является «предысторией» для книги, написанной мной в 2002 году: «Agile Software Development: Principles, Patterns, and Practices» (сокращенно PPP). Книга PPP посвящена принципам объектно-ориентированного проектирования и практическим приемам, используемым профессиональными разработчиками. Если вы еще не читали PPP, скажу, что там развивается тема, начатая в этой книге. Прочитавшие убедятся, что многие идеи перекликаются с идеями, изложенными в этой книге на уровне кода.

[1] Из прощального послания Роберта Стивенсона Смита Баден-Пауэлла скаутам: «Постарайтесь оставить этот мир чуть лучшим, чем он был до вашего прихода...»

В этой книге периодически встречаются ссылки на различные принципы проектирования. В частности, упоминается принцип единой ответственности (SRP), принцип открытости/закрытости (OCP) и принцип обращения зависимостей (DIP). Все эти принципы подробно описаны в PPP.

Заключение

Книги по искусству не обещают сделать из вас художника. Все, что они могут — познакомить вас с приемами, инструментами и направлением мысли других художников. Эта книга тоже не обещает сделать из вас хорошего программиста. Она не обещает сформировать у вас «чувство кода». Я могу лишь показать, в каком направлении мыслят хорошие программисты и какие приемы, трюки и инструменты они применяют в своей работе.

Подобно книгам по искусству, эта книга насыщена подробностями. В ней много кода — как хорошего, так и плохого. Вы увидите, как плохой код преобразуется в хороший. Вы найдете списки эвристических правил, дисциплин и методов. Вы увидите множество примеров. А дальше дело только за вами.

Помните старый анекдот о скрипаче, который заблудился по пути на концерт? Он остановил старика на углу и спросил, как попасть в Карнеги-холл. Старик посмотрел на скрипача, на зажатую у него под мышкой скрипку и сказал: «Старайся, сынок. Старайся!»

Литература

[Beck07]: Implementation Patterns, Kent Beck, Addison-Wesley, 2007.

[Knuth92]: Literate Programming, Donald E. Knuth, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

2 Содержательные имена

Тим Оттингер

Имена встречаются в программировании повсеместно. Мы присваиваем имена своим переменным, функциям, аргументам, классам и пакетам. Мы присваиваем имена исходным файлам и каталогам, в которых они хранятся. Мы присваиваем имена файлам *jar*, *war* и *ear*. Имена, имена, имена... Но то, что делается так часто, должно делаться хорошо. Далее приводятся некоторые простые правила создания хороших имен.

Имена должны передавать намерения программиста

Легко сказать: имена должны передавать намерения программиста. И все же к выбору имен следует относиться серьезно. Чтобы выбрать хорошее имя, понадобится время, но экономия окупит затраты. Итак, следите за именами в своих программах и изменяйте их, если найдете более удачные варианты. Этим вы упростите жизнь каждому, кто читает ваш код (в том числе и себе самому).

Имя переменной, функции или класса должно отвечать на все главные вопросы. Оно должно сообщить, почему эта переменная (и т. д.) существует, что она делает и как используется. Если имя требует дополнительных комментариев, значит, оно не передает намерений программиста.

```
int d; // Прошедшее время
```

Имя `d` не передает ровным счетом ничего. Оно не ассоциируется ни с временными интервалами, ни с днями. Его следует заменить другим именем, которое указывает, что именно измеряется и в каких единицах:

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Содержательные имена существенно упрощают понимание и модификацию кода. Например, что делает следующий фрагмент?

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

Почему мы не можем сразу сказать, что делает этот код? В нем нет сложных выражений. Пробелы и отступы расставлены грамотно. В коде задействованы только три переменные и две константы. В нем нет никаких хитроумных классов или полиморфных методов, только список массивов (по крайней мере на первый взгляд).

Проблема кроется не в сложности кода, а в его *неочевидности*, то есть степени, в которой контекст не следует явно из самого кода. Код подразумевает, что мы знаем ответы на вопросы:

1. Какие данные хранятся в `theList`?
2. Чем так важен элемент `theList` с нулевым индексом?
3. Какой особый смысл имеет значение 4?
4. Как будет использоваться возвращаемый список?

Ответы на все эти вопросы не следуют из примера, хотя и могли бы. Допустим, мы работаем над игрой «Сапер». Игровое поле представлено в виде списка ячеек с именем `theList`. Переименуем его в `gameBoard`.

Каждая ячейка игрового поля представлена простым массивом. Далее выясняется, что в элементе с нулевым индексом хранится код состояния, а код 4 означает «флажок установлен». Даже простое присваивание имен всем этим концепциям существенно улучшает код:

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Обратите внимание: простота кода несколько не изменилась. Новая версия содержит точно такое же количество операторов и констант, с абсолютно таким же количеством уровней вложенности. Однако код стал существенно более понятным.

Можно пойти еще дальше и написать простой класс для представления ячеек вместо использования массива `int`. В класс включается функция, передающая намерения программиста (назовем ее `isFlagged`); она скрывает «волшебные» числа. В результате мы получаем новую версию функции:

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Не изменилось ничего, кроме имен — но теперь можно легко понять, что здесь происходит. Такова сила выбора хороших имен.

Избегайте дезинформации

Программисты должны избегать ложных ассоциаций, затемняющих смысл кода. Не используйте слова со скрытыми значениями, отличными от предполагаемого. Например, переменным не стоит присваивать имена `hp`, `aix`, and `sco`, потому что они ассоциируются с платформами и разновидностями Unix. Даже если в переменной хранится длина гипотенузы и имя `hp` кажется хорошим сокращением, оно может ввести в заблуждение читателя кода.

Не обозначайте группу учетных записей именем `accountList`, если только она действительно не хранится в списке (`List`). Слово «список» имеет для программиста вполне конкретный смысл. Если записи хранятся не в `List`, а в другом контейнере,

это может привести к ложным выводам[1]. В этом примере лучше подойдет имя `accountGroup`, `bunchOfAccounts` и даже просто `accounts`.

Остерегайтесь малозаметных различий в именах. Сколько времени понадобится, чтобы заметить незначительное различие в `XYZControllerForEfficientHandlingOfStrings` в одном модуле и `XYZControllerForEfficientStorageOfStrings` где-то в другом месте? Эти имена выглядят устрашающе похожими.

Сходное представление сходных концепций — информация. Непоследовательное представление — дезинформация. Современные среды Java поддерживают удобный механизм автоматического завершения кода. Вы вводите несколько символов имени, нажимаете некую комбинацию клавиш (а иногда обходится и без этого) и получаете список возможных вариантов завершения имени. Очень удобно, если имена похожих объектов сортируются по алфавиту, и если различия предельно очевидны — ведь разработчик, скорее всего, выберет ваш объект по имени, не увидев ни ваших обширных комментариев, ни хотя бы списка методов класса.

По-настоящему устрашающие примеры дезинформирующих имен встречаются при использовании строчной «L» и прописной «O» в именах переменных, особенно в комбинациях. Естественно, проблемы возникают из-за того, что эти буквы почти не отличаются от констант «1» и «0» соответственно.

```
int a = l;
if ( O == l )
  a = O1;
else
  l = O1;
```

Возможно, некоторым читателям этот совет покажется надуманным, однако мы неоднократно видели код, в котором подобных ухищрений было предостаточно. В одном случае автор кода даже предложил использовать другой шрифт, чтобы различия стали более очевидными — в дальнейшем это решение должно было передаваться всем будущим разработчикам на словах или в письменном документе. Простое переименование решает проблему окончательно и без создания новых документов.

Используйте осмысленные различия

Когда программист пишет код исключительно для того, чтобы удовлетворить запросы компилятора или интерпретатора, он сам себе создает проблемы. Например, поскольку одно имя в одной области имени не может обозначать две разные вещи,

[1] Как будет показано ниже, даже если контейнер *действительно* представляет собой `List`, лучше обойтись без кодирования типа контейнера в имени.

возникает соблазн произвольно изменить одно из имен. Иногда для этого имя записывается заведомо неправильно и возникает удивительная ситуация: после исправления грамматической ошибки программа перестает компилироваться[1].

Недостаточно добавить в имя серию цифр или неинформативные слова, даже если компилятору этого будет достаточно. Если имена различаются, то они должны обозначать разные понятия.

«Числовые ряды» вида `(a1, a2, .. aN)` являются противоположностью сознательного присваивания имен. Такие имена не дезинформируют — они просто не несут информации и не дают представления о намерениях автора. Пример:

```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

Такая функция будет читаться намного лучше, если присвоить аргументам имена `source` и `destination`.

Неинформативные слова также применяются для создания бессодержательных различий. Допустим, у вас имеется класс `Product`. Создав другой класс с именем `ProductInfo` или `ProductData`, вы создаете разные имена, которые по сути обозначают одно и то же. `Info` и `Data` не несут полезной информации, как и артикли *a*, *an* и *the*.

Следует учесть, что использование префиксов *a* и *the* вовсе не является ошибкой, но только при условии, что они создают осмысленные различия. Например, префикс *a* может присваиваться всем локальным переменным, а префикс *the* — всем аргументам функций[2]. Проблема возникает тогда, когда вы называете переменную `theZork`, потому что в программе уже есть другая переменная с именем `zork`.

Неинформативные слова избыточны. Слово *variable* никогда не должно встречаться в именах переменных. Слово *table* никогда не должно встречаться в именах таблиц. Чем имя `NameString` лучше `Name`? Разве имя может быть, скажем, вещественным числом? Если может, то это нарушает предыдущее правило о дезинформации. Представьте, что в программе присутствуют два класса с именами `Customer` и `CustomerObject`. Что вы можете сказать о различиях между ними? Какой класс предоставляет лучший путь к истории платежей клиента?

Эта проблема встретилась нам в одном реально существующем приложении. Мы изменили имена, чтобы защитить виновных, но точная форма ошибки выглядит так:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

---

[1] Для примера можно привести совершенно отвратительную привычку создавать переменную `klass` только из-за того, что имя `class` было использовано для других целей.

[2] Дядюшка Боб действовал так при программировании на C++, но потом бросил эту привычку, потому что благодаря современным IDE она стала излишней.

Как участвующему в проекте программисту понять, какую из этих функций вызывать в конкретном случае?

При отсутствии жестких именных схем имя `moneyAmount` не отличается от `money`, `customerInfo` не отличается от `customer`, `accountData` не отличается от `account`, а `theMessage` — от `message`. Записывайте различающиеся имена так, чтобы читатель кода понимал, какой смысл заложен в этих различиях.

Используйте удобопроизносимые имена

Людям удобно работать со словами. Значительная часть нашего мозга специализируется на концепции слов, а слова по определению удобопроизносимы. Было бы обидно не использовать ту изрядную часть мозга, которая развивалась для разговорной речи. Следовательно, имена должны нормально произноситься.

Если имя невозможно нормально произнести, то при любом его упоминании в обсуждении вы выглядите полным идиотом. «Итак, за этим би-си-эр-три-си-эн-тэ у нас идет пи-эс-зэт-кью, видите?» А это важно, потому что программирование является социальной деятельностью.

В одной известной мне компании используется переменная `genymdhms` (дата генерирования, год, месяц, день, час, минуты и секунды), поэтому программисты упоминали в своих разговорах «ген-уай-эм-ди-эйч-эм-эс». У меня есть противная привычка произносить все так, как написано, поэтому я начал говорить «генъямадда-химс». Потом переменную начали так называть многие проектировщики и аналитики, и это звучало довольно глупо. Впрочем, мы делали это в шутку. Но как бы то ни было, мы столкнулись с типичным примером неудачного выбора имен. Новым разработчикам приходилось объяснять смысл переменных, после чего они начинали изъясняться дурацкими неестественными словами вместо нормальной разговорной речи. Сравните:

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /* ... */
};
```

и

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /* ... */
};
```

Теперь становится возможным осмысленный разговор: «Эй, Майк, глянь-ка на эту запись! В поле временного штампа заносится завтрашняя дата! Разве такое возможно?»

Выбирайте имена, удобные для поиска

У однобуквенных имен и числовых констант имеется один специфический недостаток: их трудно искать в большом объеме текста.

Строка MAX_CLASSES_PER_STUDENT отыскивается легко, а с числом 7 могут возникнуть проблемы. Система поиска находит эту цифру в именах файлов, в определениях констант и в различных выражениях, где значение используется с совершенно другим смыслом. Еще хуже, если константа представляет собой длинное число, в котором были случайно переставлены цифры; в программе появляется ошибка, которая одновременно скрывается от поиска.

Также не стоит присваивать имя e переменной, которая может использоваться при поиске. Самая распространенная буква английского алфавита с большой вероятностью встречается в любом текстовом фрагменте каждой программы. В этом отношении длинные имена лучше коротких, а имена, удобные для поиска, лучше констант в коде.

Лично я считаю, что однобуквенные имена могут использоваться ТОЛЬКО для локальных переменных в коротких методах. *Длина имени должна соответствовать размеру его области видимости* [N5]. Если переменная или константа может встречаться или использоваться в нескольких местах кодового блока, очень важно присвоить ей имя, удобное для поиска. Снова сравните:

```
for (int j=0; j<34; j++) {
  s += (t[j]*4)/5;
}
```

и

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

Имя sum в этом фрагменте не слишком содержательно, но по крайней мере его удобно искать. Сознательное присваивание имен увеличивает длину функции, но подумайте, насколько проще найти WORK_DAYS_PER_WEEK, чем искать все вхождения цифры 5 и фильтровать список до позиций с нужным смыслом.

Избегайте схем кодирования имен

У нас и так хватает хлопот с кодированием, чтобы искать новые сложности. Кодирование информации о типе или области видимости в именах только создает новые хлопоты по расшифровке. Вряд ли разумно заставлять каждого нового работника изучать очередной «язык» кодирования — в дополнение к изучению

(обычно немалого) объема кода, с которым он будет работать. Это только усложняет его работу при попытке решения задачи. Как правило, кодированные имена плохо произносятся и в них легко сделать опечатку.

Венгерская запись

В доисторические времена, когда в языках действовали ограничения на длину имен, мы нарушали это правило по необходимости — и не без сожалений. В Fortran первая буква имени переменной обозначала код типа. В ранних версиях BASIC имена могли состоять только из одной буквы и одной цифры. Венгерская запись (HN, Hungarian Notation) подняла эту проблему на новый уровень.

Венгерская запись играла важную роль во времена Windows C API, когда программы работали с целочисленными дескрипторами (handle), длинными указателями, указателями на `void` или различными реализациями «строк» (с разным применением и атрибутами). Компиляторы в те дни не поддерживали проверку типов, поэтому программистам были нужны «подсказки» для запоминания типов.

В современных языках существует куда более развитая система типов, а компиляторы запоминают типы и обеспечивают их соблюдение. Более того, появилась тенденция к использованию меньших классов и более коротких функций, чтобы программисты видели точку объявления каждой используемой переменной.

Java-программисту кодировать типы в именах не нужно. Объекты обладают сильной типизацией, а рабочие среды развились до такой степени, что могут выявить ошибку типа еще до начала компиляции! Таким образом, в наши дни венгерская запись и другие формы кодирования типов в именах превратились в обычные пережитки прошлого. Они усложняют изменение имени или типа переменных, функций и классов. Они затрудняют чтение кода. Наконец, они повышают риск того, что система кодирования собьет с толку читателя кода.

```
PhoneNumber phoneString;
// Имя не изменяется при изменении типа!
```

Префиксы членов классов

Префиксы `m_`, которыми когда-то снабжались переменные классов, тоже стали ненужными. Классы и функции должны быть достаточно компактными, чтобы вы могли обходиться без префиксов. Также следует использовать рабочую среду с цветовым выделением членов классов, обеспечивающим их наглядную идентификацию:

```
public class Part {
  private String m_dsc; // Текстовое описание
  void setName(String name) {
    m_dsc = name;
  }
}
```

– –

```
public class Part {
  String description;
  void setDescription(String description) {
    this.description = description;
  }
}
```

Кроме того, люди быстро учатся игнорировать префиксы (и суффиксы), чтобы видеть содержательную часть имени. Чем больше мы читаем код, тем реже замечаем префиксы. В конечном итоге префикс превращается в невидимый балласт, характерный для старого кода.

Интерфейсы и реализации

Иногда в программах встречается особый случай кодирования. Допустим, вы строите АБСТРАКТНУЮ ФАБРИКУ для создания геометрических фигур. Фабрика представляет собой интерфейс, который реализуется конкретным классом. Как их назвать? `IShapeFactory` и `ShapeFactory`? Я предпочитаю оставлять имена интерфейсов без префиксов. Префикс `I`, столь распространенный в старом коде, в лучшем случае отвлекает, а в худшем — передает лишнюю информацию. Я не собираюсь сообщать своим пользователям, что они имеют дело с интерфейсом. Им достаточно знать, что это `ShapeFactory`, то есть фабрика фигур. Следовательно, при необходимости закодировать в имени либо интерфейс, либо реализацию, я выбираю реализацию. Имя `ShapeFactoryImp`, или даже уродливое `CShapeFactory`, все равно лучше кодирования информации об интерфейсе.

Избегайте мысленных преобразований

Не заставляйте читателя мысленно преобразовывать ваши имена в другие, уже известные ему. Обычно эта проблема возникает из-за нежелания использовать понятия как из пространства задачи, так и из пространства решения.

Такая проблема часто возникает при использовании однобуквенных имен переменных. Конечно, счетчик цикла можно назвать `i`, `j` или `k` (но только не `l`!), если его область видимости очень мала, и он не конфликтует с другими именами. Это связано с тем, что однобуквенные имена счетчиков циклов традиционны. Однако в большинстве других контекстов однобуквенные имена нежелательны; в сущности, вы создаете временный заменитель, который должен быть мысленно преобразован пользователем в реальную концепцию. Нет худшей причины для выбора имени `c`, чем та, что имена `a` и `b` уже заняты.

Как правило, программисты весьма умны. А умные люди иногда любят показывать мощь интеллекта, демонстрируя свои способности к мысленному жонглированию. В конце концов, если вы помните, что переменная `r` содержит URL-адрес с удаленным хостом и схемой, преобразованный к нижнему регистру, это совершенно очевидно свидетельствует о вашем уме.

Одно из различий между умным и профессиональным программистом заключается в том, что профессионал понимает: ясность превыше всего. Профессионалы используют свою силу во благо и пишут код, понятный для других людей.

Имена классов

Имена классов и объектов должны представлять собой существительные и их комбинации: `Customer`, `WikiPage`, `Account` и `AddressParser`. Старайтесь не использовать в именах классов такие слова, как `Manager`, `Processor`, `Data` или `Info`. Имя класса не должно быть глаголом.

Имена методов

Имена методов представляют собой глаголы или глагольные словосочетания: `postPayment`, `deletePage`, `save` и т. д. Методы чтения/записи и предикаты образуются из значения и префикса `get`, `set` и `is` согласно стандарту javabean[1].

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

При перегрузке конструкторов используйте статические методы-фабрики с именами, описывающими аргументы. Например, запись

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

обычно лучше записи

```
Complex fulcrumPoint = new Complex(23.0);
```

Рассмотрите возможность принудительного использования таких методов; для этого соответствующие конструкторы объявляются приватными.

Избегайте остроумия

Если ваши имена будут излишне остроумными, то их смысл будет понятен только людям, разделяющим чувство юмора автора — и только если они помнят шутку. Все ли догадаются, что делает функция с именем `HolyHandGrenade`?[2] Конечно, это очень мило, но, возможно, в данном случае лучше

[1] http://java.sun.com/products/javabeans/docs/spec.html.

[2] «Святая ручная граната» — оружие огромной разрушительной силы из фильма «Монти Пайтон и Священный Грааль». — *Примеч. перев.*

подойдет имя DeleteItems. Отдавайте предпочтение ясности перед развлекательной ценностью.

Остроумие часто воплощается в форме просторечий или сленга. Например, не используйте имя whack() вместо kill(). Не используйте шуточки, привязанные к конкретной культуре, — например, eatMyShorts[1]() вместо abort().

Выберите одно слово для каждой концепции

Выберите одно слово для представления одной абстрактной концепции и придерживайтесь его. Например, существование в разных классах эквивалентных методов с именами fetch, retrieve и get неизбежно создаст путаницу. Как запомнить, к какому классу относится то или иное имя метода? К сожалению, чтобы запомнить, какой термин использовался в той или иной библиотеке или классе, нередко приходится помнить, какой компанией, группой или программистом эта библиотека была создана. В противном случае вы потратите массу времени на просмотр заголовков и предыдущих примеров кода.

Современные рабочие среды (такие, как Eclipse и IntelliJ) предоставляют контекстно-зависимые подсказки — скажем, список методов, которые могут вызываться для конкретного объекта. Однако следует учитывать, что в этом списке обычно не приводятся комментарии, которые вы записываете рядом с именами функций и списками параметров. И вам еще повезло, если в нем будут указаны имена параметров из объявлений функций. Имена функций должны быть законченными и логичными, чтобы программист мог сразу выбрать правильный метод без сбора дополнительной информации.

Аналогичным образом, использование терминов controller, manager и driver в одной кодовой базе тоже вызывает путаницу. Чем DeviceManager принципиально отличается от ProtocolController? Почему в двух случаях не используются одинаковые термины? Такие имена создают ложное впечатление, что два объекта обладают совершенно разными типами, а также относятся к разным классам.

Единый, согласованный лексикон окажет неоценимую помощь программистам, которые будут пользоваться вашим кодом.

Воздержитесь от каламбуров

Старайтесь не использовать одно слово в двух смыслах. В сущности, обозначение двух разных идей одним термином — это каламбур.

[1] Из мультипликационного сериала «Симпсоны». — *Примеч. перев.*

Если следовать принципу «одно слово для каждой концепции», в программе может появиться много классов, содержащих, например, метод `add`. Пока списки параметров и возвращаемые значения разных методов `add` остаются семантически эквивалентными, все хорошо.

Однако программист может решить использовать имя `add` «ради единообразия» независимо от того, выполняет ли этот метод добавление в прежнем смысле или нет. Допустим, программа содержит много классов с методами `add`, которые создают новое значение сложением или конкатенацией двух существующих значений. Вы пишете новый класс с методом, помещающим свой единственный параметр в коллекцию. Стоит ли присвоить этому методу имя `add`? На первый взгляд это выглядит последовательно, потому что в программе уже используется множество других методов `add`, но новый метод имеет другую семантику, поэтому ему лучше присвоить имя `insert` или `append`. Присваивая новому методу имя `add`, вы создаете нежелательный каламбур.

Задача автора — сделать свой код как можно более понятным. Код должен восприниматься с первого взгляда, не требуя тщательного изучения. Ориентируйтесь на модель популярной литературы, в которой сам автор должен доступно выразить свои мысли, а не на академическую модель, в которой ученик усердным трудом постигает скрытый смысл публикации.

Используйте имена из пространства решения

Не забывайте: ваш код будут читать программисты. А раз так, не стесняйтесь использовать термины из области информатики, названия алгоритмов и паттернов, математические термины и т. д. Не ограничивайтесь именами исключительно из пространства задачи; не заставляйте своих коллег постоянно бегать к клиенту и спрашивать, что означает каждое имя, когда соответствующая концепция уже знакома им под другим названием.

Имя `AccountVisitor` сообщит много полезной информации программисту, знакомому с паттерном «Посетитель» (Visitor). И какой программист не знает, что такое «очередь заданий» (`JobQueue`)? Существует множество сугубо технических понятий, с которыми имеют дело программисты. Как правило, таким понятиям разумнее всего присваивать технические имена.

Используйте имена из пространства задачи

Если для того, что вы делаете, не существует подходящего «программизма», используйте имя из пространства задачи. По крайней мере программист, занима-

ющийся сопровождением кода, сможет узнать у специалиста в предметной области, что означает это имя.

Разделение концепций из пространств задачи и решения — часть работы хорошего программиста и проектировщика. В коде, главным образом ориентированном на концепции из пространства задачи, следует использовать имена из пространства задачи.

Добавьте содержательный контекст

Лишь немногие имена содержательны сами по себе. Все остальные имена следует помещать в определенный контекст для читателя кода, заключая их в классы, функции и пространства имен с правильно выбранными названиями. В крайнем случае контекст имени можно уточнить при помощи префикса.

Допустим, в программе используются переменные с именами `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state` и `zipcode`. Вполне очевидно, что в совокупности они образуют адрес. Но что, если переменная `state` встретилась вам отдельно от других переменных внутри метода? Сразу ли вы поймете, что она является частью адреса?

Контекст можно добавить при помощи префиксов: `addrFirstName`, `addrLastName`, `addrState` и т. д. По крайней мере читатель кода поймет, что переменные являются частью более крупной структуры. Конечно, правильнее было бы создать класс с именем `Address`, чтобы даже компилятор знал, что переменные являются частью чего-то большего.

Возьмем метод из листинга 2.1. Нужен ли переменным более содержательный контекст? Имя функции определяет только часть контекста; алгоритм предоставляет все остальное. При чтении функции становится видно, что три переменные `number`, `verb` и `pluralModifier` являются компонентами сообщения `guessMessage`. К сожалению, контекст приходится вычислять. При первом взгляде на метод смысл переменных остается неясным.

Листинг 2.1. Переменные с неясным контекстом

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
      number = "no";
      verb = "are";
      pluralModifier = "s";
    } else if (count == 1) {
      number = "1";
      verb = "is";
      pluralModifier = "";
```

продолжение ⇩

Листинг 2.1 (продолжение)

```
    } else {
      number = Integer.toString(count);
      verb = "are";
      pluralModifier = "s";
    }
    String guessMessage = String.format(
      "There %s %s %s%s", verb, number, candidate, pluralModifier
    );
    print(guessMessage);
  }
```

Функция длинновата, а переменные используются на всем ее протяжении. Чтобы разделить функцию на меньшие смысловые фрагменты, следует создать класс `GuessStatisticsMessage` и сделать три переменные полями этого класса. Тем самым мы предоставим очевидный контекст для трех переменных — теперь абсолютно очевидно, что эти переменные являются частью `GuessStatisticsMessage`. Уточнение контекста также позволяет заметно улучшить четкость алгоритма за счет его деления на меньшие функции (листинг 2.2).

Листинг 2.2. Переменные с контекстом

```
public class GuessStatisticsMessage {
  private String number;
  private String verb;
  private String pluralModifier;

  public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
    return String.format(
      "There %s %s %s%s",
       verb, number, candidate, pluralModifier );
  }

  private void createPluralDependentMessageParts(int count) {
    if (count == 0) {
      thereAreNoLetters();
    } else if (count == 1) {
      thereIsOneLetter();
    } else {
      thereAreManyLetters(count);
    }
  }

  private void thereAreManyLetters(int count) {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
  }

  private void thereIsOneLetter() {
```

```
    number = "1";
    verb = "is";
    pluralModifier = "";
  }

  private void thereAreNoLetters() {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  }
}
```

Не добавляйте избыточный контекст

Если вы работаете над вымышленным приложением «Gas Station Deluxe», не стоит снабжать имя каждого класса префиксом `GSD`. В сущности, вы работаете против собственного инструментария. Введите букву «G», нажмите клавишу завершения — и вы получите длинный-предлинный список всех классов в системе. Разумно ли это? IDE пытается помочь вам, так стоит ли ей мешать?

Допустим, вы изобрели класс `MailingAddress` в учетном модуле GSD и присвоили ему имя `GSDAccountAddress`. Позднее адрес используется в приложении, обеспечивающем связь с клиентами. Будете ли вы использовать `GSDAccountAddress`? Насколько подходящим выглядит это имя? Десять из 17 символов либо избыточны, либо не относятся к делу.

Короткие имена обычно лучше длинных, если только их смысл понятен читателю кода. Не включайте в имя больше контекста, чем необходимо.

Имена `accountAddress` и `customerAddress` хорошо подходят для экземпляров класса `Address`, но для классов такой выбор неудачен. `Address` — вот хорошее имя класса. Если потребуется подчеркнуть различия между MAC-адресами, адресами портов и веб-адресами, я подумаю об использовании имен `PostalAddress`, `MAC` и `URI`. Полученные имена становятся более точными, а это, собственно, и является главной целью всего присваивания имен.

Несколько слов напоследок

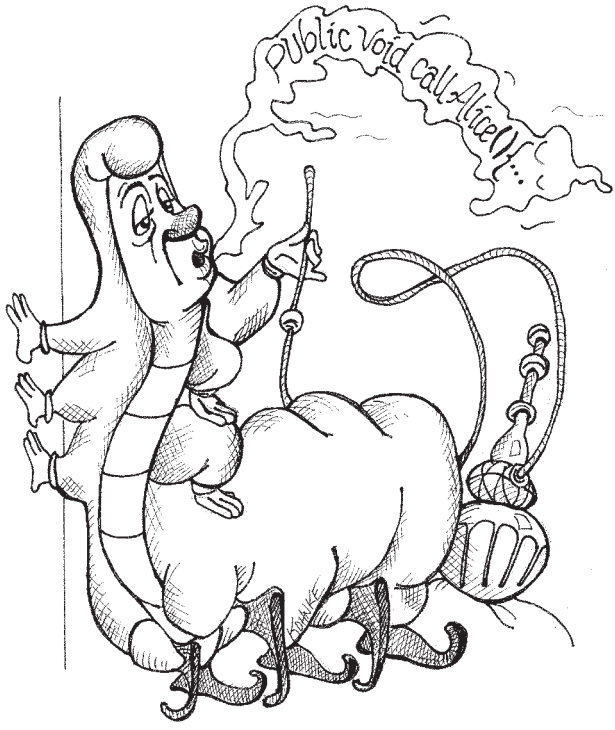
Основные трудности с выбором хороших имен обусловлены необходимостью хороших описательных навыков и единого культурного фона. Это вопрос преподавания, а не вопрос техники, экономики или управления. В результате многие специалисты, работающие в этой области, так и не научились хорошо справляться с этой задачей.

Люди также опасаются переименований из страха возражений со стороны других разработчиков. Мы не разделяем эти опасения, а изменение имен (в лучшую сторону) вызывает у нас только благодарность. Большей частью мы не запоминаем

имена классов и методов. Современные инструменты берут на себя подобные мелочи, а мы следим за тем, чтобы программный код читался как абзацы и предложения или хотя бы как таблицы и структуры данных (предложение не всегда является лучшим способом отображения данных). Возможно, своими переименованиями — как и любыми другими усовершенствованиями кода — вы кого-то удивите. Пусть это вас не останавливает.

Последуйте этим правилам и посмотрите, не станет ли ваш код более удобочитаемым. Если вы занимаетесь сопровождением чужого кода, попробуйте решить проблемы средствами рефакторинга. Это даст немедленный результат и продолжит приносить плоды в долгосрочной перспективе.

3 Функции



На заре эпохи программирования системы строились из программ, функций и подпрограмм. До наших дней дожили только функции. Они образуют первый уровень структуризации в любой программе, и их грамотная запись является основной темой этой главы.

Рассмотрим код в листинге 3.1. В FitNesse[1] трудно найти длинную функцию, но после некоторых поисков мне это все же удалось. Функция не только длинна,

[1] Тестовая программа, распространяемая с открытым кодом — www.tnese.org.

но она содержит повторяющиеся фрагменты кода, множество загадочных строк, а также странные и неочевидные типы данных и функции API. Попробуйте разобраться в ней за три минуты. Посмотрим, что вам удастся понять.

Листинг 3.1. HtmlUtil.java (FitNesse 20070619)

```
public static String testableHtml(
  PageData pageData,
  boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
      if (suiteSetup != null) {
        WikiPagePath pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
              .append(pagePathName)
              .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
    if (setup != null) {
      WikiPagePath setupPath =
        wikiPage.getPageCrawler().getFullPath(setup);
      String setupPathName = PathParser.render(setupPath);"
      buffer.append("!include -setup .")
            .append(setupPathName)
            .append("\n");
    }
  }
  buffer.append(pageData.getContent());
  if (pageData.hasAttribute("Test")) {
    WikiPage teardown =
      PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
      WikiPagePath tearDownPath =
        wikiPage.getPageCrawler().getFullPath(teardown);
      String tearDownPathName = PathParser.render(tearDownPath);
      buffer.append("\n")
            .append("!include -teardown .")
            .append(tearDownPathName)
            .append("\n");
    }
```

```
    if (includeSuiteSetup) {
      WikiPage suiteTeardown =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_TEARDOWN_NAME,
                wikiPage
        );
      if (suiteTeardown != null) {
        WikiPagePath pagePath =
          suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
              .append(pagePathName)
              .append("\n");
      }
    }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

Удалось ли вам разобраться с функцией за три минуты? Вероятно, нет. В ней происходит слишком много всего, и притом на разных уровнях абстракции. Загадочные строки и непонятные вызовы функций смешиваются в конструкциях `if` двойной вложенности, к тому же зависящих от состояния флагов.

Но после выделения нескольких методов, переименований и небольшой реструктуризации мне удалось представить смысл этой функции в девяти строках листинга 3.2. Посмотрим, удастся ли вам разобраться в ней за следующие три минуты.

Листинг 3.2. HtmlUtil.java (переработанная версия)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages(testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }

  return pageData.getHtml();
}
```

Если только вы не занимаетесь активным изучением FitNesse, скорее всего, вы не разберетесь во всех подробностях. Но по крайней мере вы поймете, что функция включает в тестовую страницу какие-то начальные и конечные блоки, а потом

генерирует код HTML. Если вы знакомы с JUnit[1], то, скорее всего, поймете, что эта функция является частью тестовой инфраструктуры на базе Web. И конечно, это правильное предположение. Прийти к такому выводу на основании листинга 3.2 несложно, но из листинга 3.1 это, мягко говоря, неочевидно.

Что же делает функцию из листинга 3.2 такой понятной и удобочитаемой? Как заставить функцию передавать намерения разработчика? Какие атрибуты функции помогут случайному читателю составить интуитивное представление о выполняемых ей задачах?

Компактность!

Первое правило: функции должны быть компактными. Второе правило: *функции должны быть еще компактнее*. Я не могу научно обосновать свое утверждение. Не ждите от меня ссылок на исследования, доказывающие, что очень маленькие функции лучше больших. Я могу всего лишь сказать, что я почти четыре десятилетия писал функции всевозможных размеров. Мне доводилось создавать кошмарных монстров в 3000 строк. Я написал бесчисленное множество функций длиной от 100 до 300 строк. И я писал функции от 20 до 30 строк. Мой практический опыт научил меня (ценой многих проб и ошибок), что функции должны быть очень маленькими. В 80-е годы считалось, что функция должна занимать не более одного экрана. Конечно, тогда экраны VT100 состояли из 24 строк и 80 столбцов, а редакторы использовали 4 строки для административных целей. В наши дни с мелким шрифтом на хорошем большом мониторе можно разместить 150 символов в строке и 100 и более строк на экране. Однако строки не должны состоять из 150 символов, а функции — из 100 строк. Желательно, чтобы длина функции не превышала 20 строк.

Насколько короткой может быть функция? В 1999 году я заехал к Кенту Беку в его дом в Орегоне. Мы посидели и позанимались программированием. В один момент он показал мне симпатичную маленькую программу Java/Swing, которую он назвал Sparkle. Программа создавала на экране визуальный эффект, очень похожий на эффект волшебной палочки феи-крестной из фильма «Золушка». При перемещении мыши с курсора рассыпались замечательные блестящие искорки, которые осыпались к нижнему краю экрана под воздействием имитируемого гравитационного поля. Когда Кент показал мне код, меня поразило, насколько компактными были все функции. Многие из моих функций в программах Swing растягивались по вертикали чуть ли не на километры. Однако каждая функция в программе Кента занимала всего две, три или четыре строки. Все функции были предельно очевидными. Каждая функция излагала свою историю, и каждая исто-

[1] Программа модульного тестирования для Java, распространяемая с открытым кодом — www.junit.org.

рия естественным образом подводила вас к началу следующей истории. *Вот* какими короткими должны быть функции[1]!

Более того, функции должны быть еще короче, чем в листинге 3.2! На деле листинг 3.2 следовало бы сократить до листинга 3.3.

Листинг 3.3. HtmlUtil.java (переработанная версия)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

Блоки и отступы

Из сказанного выше следует, что блоки в командах `if`, `else`, `while` и т. д. должны состоять из одной строки, в которой обычно содержится вызов функции. Это не только делает вмещающую функцию более компактной, но и способствует документированию кода, поскольку вызываемой в блоке функции можно присвоить удобное содержательное имя.

Кроме того, функции не должны содержать вложенных структур, так как это приводит к их увеличению. Максимальный уровень отступов в функции не должен превышать одного-двух. Разумеется, это упрощает чтение и понимание функций.

Правило одной операции

Совершенно очевидно, что функция из листинга 3.1 выполняет множество операций. Она создает буферы, производит выборку данных, ищет унаследованные страницы, строит пути, присоединяет загадочные строки, генерирует код HTML... и это еще не все. С другой стороны, в листинге 3.3 выполняется всего одна простая операция: включение в тестовую страницу начальных и конечных блоков.

Следующий совет существует в той или иной форме не менее 30 лет.

[1] Я спросил Кента, не сохранилась ли у него эта программа, но ему не удалось ее найти. Обшарил все свои старые компьютеры — тоже безуспешно. Остались лишь мои воспоминания об этой программе.

ФУНКЦИЯ ДОЛЖНА ВЫПОЛНЯТЬ ТОЛЬКО ОДНУ ОПЕРАЦИЮ. ОНА ДОЛЖНА ВЫПОЛНЯТЬ ЕЕ ХОРОШО. И НИЧЕГО ДРУГОГО ОНА ДЕЛАТЬ НЕ ДОЛЖНА.

Проблема в том, что иногда бывает трудно определить, что же считать «одной операцией». В листинге 3.3 выполняется одна операция? Легко возразить, что в нем выполняются минимум три операции:

1. Функция проверяет, является ли страница тестовой страницей.
2. Если является, то в нее включаются начальные и конечные блоки.
3. Для страницы генерируется код HTML.

Так как же? Сколько операций выполняет функция — одну или три? Обратите внимание: три этапа работы функции находятся на одном уровне абстракции под объявленным именем функции. Ее можно было бы описать в виде короткого `TO`[1]-абзаца:

- TO RenderPageWithSetupsAndTeardowns, мы проверяем, является ли страница тестовой, и если является — включаем начальные и конечные блоки. В любом случае для страницы генерируется код HTML.

Если функция выполняет только те действия, которые находятся на одном уровне под объявленным именем функции, то эта функция выполняет одну операцию. В конце концов, функции пишутся прежде всего для разложения более крупной концепции (иначе говоря, имени функции) на последовательность действий на следующем уровне абстракции.

Вполне очевидно, что листинг 3.1 содержит множество различных действий на разных уровнях абстракции. Поэтому в нем явно выполняется более одной операции. Даже листинг 3.2 содержит два уровня абстракции; это доказывается тем, что нам удалось его сократить. С другой стороны, осмысленно сократить листинг 3.3 очень трудно. Команду `if` можно вынести в функцию с именем `includeSetupsAndTeardownsIfTestPage`, но это простая переформулировка кода без изменения уровня абстракции.

Итак, чтобы определить, что функция выполняет более одной операции, попробуйте извлечь из нее другую функцию, которая бы не являлась простой переформулировкой реализации [G34].

Секции в функциях

Взгляните на листинг 4.7 на с. 98. Обратите внимание: функция `generatePrimes` разделена на секции (объявления, инициализация, отбор). Это очевидный признак того, что функция выполняет более одной операции. Функцию, выполняющую только одну операцию, невозможно осмысленно разделить на секции.

[1] В языке LOGO ключевое слово TO использовалось так же, как в Ruby и Python используется «def». Таким образом, каждая функция начиналась со слова «TO».

Один уровень абстракции на функцию

Чтобы убедиться в том, что функция выполняет «только одну операцию», необходимо проверить, что все команды функции находятся на одном уровне абстракции. Легко убедиться, что листинг 3.1 нарушает это правило. Некоторые из его концепций — например, `getHtml()` — находятся на очень высоком уровне абстракции; другие (скажем, `String pagePathName = PathParser.render(pagePath)`) — на среднем уровне. Наконец, третьи — такие, как `.append("\n")` — относятся к чрезвычайно низкому уровню абстракции.

Смешение уровней абстракции внутри функции всегда создает путаницу. Читатель не всегда понимает, является ли некоторое выражение важной концепцией или второстепенной подробностью. Что еще хуже, при их смешении функция постепенно начинает обрастать все большим количеством второстепенных подробностей.

Чтение кода сверху вниз: правило понижения

Код должен читаться как рассказ — сверху вниз [KP78, p. 37].
За каждой функцией должны следовать функции следующего уровня абстракции. Это позволяет читать код, последовательно спускаясь по уровням абстракции в ходе чтения списка функций. Я называю такой подход «правилом понижения».

Сказанное можно сформулировать и иначе: программа должна читаться так, словно она является набором `TO`-абзацев, каждый из которых описывает текущий уровень абстракции и ссылается на последующие `TO`-абзацы следующего нижнего уровня.

- Чтобы включить начальные и конечные блоки, мы сначала включаем начальные блоки, затем содержимое тестовой страницы, а затем включаем конечные блоки.
  - Чтобы включить начальные блоки, мы сначала включаем пакетные начальные блоки, если имеем дело с пакетом тестов, а затем включаем обычные начальные блоки.
  - Чтобы включить пакетные начальные блоки, мы ищем в родительской иерархии страницу `SuiteSetUp` и добавляем команду `include` с путем к этой странице.
  - Чтобы найти в родительской иерархии...

Опыт показывает, что программистов очень трудно научить следовать этому правилу и писать функции, остающиеся на одном уровне абстракции. Тем не менее освоить этот прием очень важно. Он играет ключевую роль для создания коротких функций, выполняющих только одну операцию. Построение кода по аналогии с набором последовательных `TO`-абзацев — эффективный метод поддержания единого уровня абстракции.

Взгляните на листинг 3.7 в конце этой главы. В нем приведен полный код функции `testableHtml`, переработанной в соответствии с описанными здесь принципами. Обратите внимание на то, как каждая функция «представляет» читателю следующую функцию и как каждая функция остается на едином уровне абстракции.

Команды switch

Написать компактную команду `switch` довольно сложно[1]. Даже команда `switch` всего с двумя условиями занимает больше места, чем в моем представлении должен занимать один блок или функция. Также трудно создать команду `switch`, которая делает что-то одно — по своей природе команды `switch` всегда выполняют N операций. К сожалению, обойтись без команд `switch` удается не всегда, но по крайней мере мы *можем* позаботиться о том, чтобы эти команды были скрыты в низкоуровневом классе и не дублировались в коде. И конечно, в этом нам может помочь полиморфизм.

В листинге 3.4 представлена всего одна операция, зависящая от типа работника.

Листинг 3.4. Payroll.java

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
      case COMMISSIONED:
        return calculateCommissionedPay(e);
      case HOURLY:
        return calculateHourlyPay(e);
      case SALARIED:
        return calculateSalariedPay(e);
      default:
        throw new InvalidEmployeeType(e.type);
    }
  }
```

Эта функция имеет ряд недостатков. Во-первых, она велика, а при добавлении новых типов работников она будет разрастаться. Во-вторых, она совершенно очевидно выполняет более одной операции. В-третьих, она нарушает принцип единой ответственности[2], так как у нее существует несколько возможных причин изменения. В-четвертых, она нарушает принцип открытости/закрытости[3], потому что код функции должен изменяться при каждом добавлении новых типов.

---

[1] Разумеется, сюда же относятся и длинные цепочки `if/else`.

[2] http://en.wikipedia.org/wiki/Single_responsibility_principle; http://www.objectmentor.com/resources/articles/srp.pdf.

[3] http://en.wikipedia.org/wiki/Open/closed_principle; http://www.objectmentor.com/resources/articles/ocp.pdf.

Но, пожалуй, самый серьезный недостаток заключается в том, что программа может содержать неограниченное количество других функций с аналогичной структурой, например:

```
isPayday(Employee e, Date date)
```

или

```
deliverPay(Employee e, Money pay)
```

и так далее. Все эти функции будут иметь все ту же ущербную структуру.

Решение проблемы (листинг 3.5) заключается в том, чтобы похоронить команду switch в фундаменте АБСТРАКТНОЙ ФАБРИКИ [GOF] и никому ее не показывать. Фабрика использует команду switch для создания соответствующих экземпляров потомков Employee, а вызовы функций calculatePay, isPayDay, deliverPay и т. д. проходят полиморфную передачу через интерфейс Employee.

Листинг 3.5. Employee и Factory

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
-----------------
public interface EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}
-----------------
public class EmployeeFactoryImpl implements EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
    switch (r.type) {
      case COMMISSIONED:
        return new CommissionedEmployee(r) ;
      case HOURLY:
        return new HourlyEmployee(r);
      case SALARIED:
        return new SalariedEmploye(r);
      default:
        throw new InvalidEmployeeType(r.type);
    }
  }
}
```

Мое общее правило в отношении команд switch гласит, что эти команды допустимы, если они встречаются в программе однократно, используются для создания полиморфных объектов и скрываются за отношениями наследования, чтобы оставаться невидимыми для остальных частей системы [G23]. Конечно, правил без исключений не бывает и в некоторых ситуациях приходится нарушать одно или несколько условий этого правила.

Используйте содержательные имена

В листинге 3.7 я переименовал нашу функцию `testableHtml` в `SetupTeardownIncluder.render`. Новое имя гораздо лучше, потому что оно точнее описывает, что делает функция. Кроме того, всем приватным методам были присвоены столь же содержательные имена `isTestable`, `includeSetupAndTeardownPages` и т. д. Трудно переоценить пользу хороших имен. Вспомните принцип Уорда: «Вы работаете с чистым кодом, если каждая функция в основном делает то, что вы от нее ожидали». Половина усилий по реализации этого принципа сводится к выбору хороших имен для компактных функций, выполняющих одну операцию. Чем меньше и специализированнее функция, тем проще выбрать для нее содержательное имя.

Не бойтесь использовать длинные имена. Длинное содержательное имя лучше короткого невразумительного. Выберите схему, которая позволяет легко прочитать слова в имени функции, а затем составьте из этих слов имя, которое описывает назначение функции.

Не бойтесь расходовать время на выбор имени. Опробуйте несколько разных имен и посмотрите, как читается код с каждым из вариантов. В современных рабочих средах (таких, как Eclipse и IntelliJ) задача смены имени решается тривиально. Используйте одну из этих сред и поэкспериментируйте с разными именами, пока не найдете самое содержательное.

Выбор содержательных имен прояснит архитектуру модуля и поможет вам усовершенствовать ее. Нередко поиски хороших имен приводят к полезной реструктуризации кода.

Будьте последовательны в выборе имен. Используйте в именах функций те же словосочетания, глаголы и существительные, которые используются в ваших модулях. Для примера можно взять имена `includeSetupAndTeardownPages`, `includeSetupPages`, `includeSuiteSetupPage` и `includeSetupPage`. Благодаря единой фразеологии эти имена рассказывают связную историю. В самом деле, если бы я показал вам только эту последовательность, вы бы спросили: «А где же `includeTeardownPages`, `includeSuiteTeardownPage` и `includeTeardownPage`?» Вспомните — «...в основном делает то, что вы от нее ожидали».

Аргументы функций

В идеальном случае количество аргументов функции равно нулю (нуль-арная функция). Далее следуют функции с одним аргументом (унарные) и с двумя аргументами (бинарные). Функций с тремя аргументами (тернарных) следует по возможности избегать. Необходимость функций с большим количеством аргументов (полиарных) должна быть подкреплена очень вескими доводами — и все равно такие функции лучше не использовать.

Аргументы усложняют функции и лишают их значительной части концептуальной мощи. Именно по этой причине я почти полностью избавился от них в этом примере. Возьмем хотя бы переменную `StringBuffer`. Ее можно было бы передать в аргументе (вместо того, чтобы делать ее переменной экземпляра), но тогда читателям кода пришлось бы интерпретировать ее каждый раз, когда она встречается в коде. Когда вы читаете историю, рассказываемую модулем, вызов `includeSetupPage()` выглядит намного более понятным, чем вызов `includeSetupPageInto(newPageContent)`. Аргумент и имя функции находятся на разных уровнях абстракции, а читателю приходится помнить о подробностях (то есть `StringBuffer`), которые на данный момент не особенно важны.

Аргументы создают еще больше проблем с точки зрения тестирования. Только представьте, как трудно составить все тестовые сценарии, проверяющие правильность работы кода со всеми комбинациями аргументов. Если аргументов нет — задача тривиальна. При одном аргументе все обходится без особых сложностей. С двумя аргументами ситуация усложняется. Если же аргументов больше двух, задача тестирования всех возможных комбинаций выглядит все более устрашающе.

Выходные аргументы запутывают ситуацию еще быстрее, чем входные. Читая код функции, мы обычно предполагаем, что функция получает информацию в аргументах, и выдает ее в возвращаемом значении. Как правило, никто не ожидает, что функция будет возвращать информацию в аргументах. Таким образом, выходные аргументы часто заставляют нас браться за чтение функции заново.

Если уж обойтись без аргументов никак не удается, постарайтесь хотя бы ограничиться одним входным аргументом. Смысл вызова `SetupTeardownIncluder.render(pageData)` вполне прозрачен — понятно, что мы собираемся сгенерировать данные для объекта `pageData`.

Стандартные унарные формы

Существует два очень распространенных случая вызова функции с одним аргументом. Первая — проверка некоторого условия, связанного с аргументом, как в вызове `boolean fileExists("MyFile")`. Вторая — обработка аргумента, его преобразование и возвращение. Например, вызов `InputStream fileOpen("MyFile")` преобразует имя файла в формате `String` в возвращаемое значение `InputStream`. Выбирайте имена, которые четко отражают различия, и всегда используйте две формы в логически непротиворечивом контексте. (См. далее «Разделение команд и запросов»).

Несколько менее распространенным, но все равно очень полезным частным случаем функции с одним аргументом является *событие*. В этой форме имеется входной аргумент, а выходного аргумента нет. Предполагается, что программа интерпретирует вызов функции как событие и использует аргумент для изменения состояния системы, например, `void passwordAttemptFailedNtimes(int attempts)`. Будьте внимательны при использовании данной формы. Читателю должно быть предельно ясно, что перед ним именно событие. Тщательно выбирайте имена и контексты.

Старайтесь избегать унарных функций, не относящихся к этим формам, например `void includeSetupPageInto(StringBuffer pageText)`. Преобразования, в которых вместо возвращаемого значения используется выходной аргумент, сбивают читателя с толку. Если функция преобразует свой входной аргумент, то результат должен передаваться в возвращаемом значении. В самом деле, вызов `StringBuffer transform(StringBuffer in)` лучше вызова `void transform(StringBuffer out)`, даже если реализация в первом случае просто возвращает входной аргумент. По крайней мере она соответствует основной форме преобразования.

Аргументы-флаги

Аргументы-флаги уродливы. Передача логического значения функции — воистину ужасная привычка. Она немедленно усложняет сигнатуру метода, громко провозглашая, что функция выполняет более одной операции. При истинном значении флага выполняется одна операция, а при ложном — другая!

В листинге 3.7 у нас нет выбора, потому что вызывающая сторона уже передает этот флаг, а я хотел ограничить область переработки границами функции. Тем не менее вызов метода `render(true)` откровенно сбивает с толку бедного читателя. Если навести указатель мыши на вызов и увидеть `render(boolean isSuite)`, ситуация слегка проясняется, но ненамного. Эту функцию следовало бы разбить на две: `renderForSuite()` и `renderForSingleTest()`.

Бинарные функции

Функцию с двумя аргументами понять сложнее, чем унарную функцию. Например, вызов `writeField(name)` выглядит более доступно, чем `writeField(outputStream, name)`[1]. Хотя смысл обеих форм понятен, первая форма просто проскальзывает под нашим взглядом, моментально раскрывая свой смысл. Во второй форме приходится сделать непродолжительную паузу, пока вы не поймете, что первый

[1] Я только что завершил переработку модуля, использовавшего бинарную форму. Мне удалось преобразовать `outputStream` в поле класса и привести все вызовы `writeField` к унарной форме. Результат получился гораздо более наглядным.

параметр должен игнорироваться. И конечно, это в конечном итоге создает проблемы, потому что никакие части кода игнорироваться не должны. Именно в проигнорированных частях чаще всего скрываются ошибки.

Конечно, в некоторых ситуациях форма с двумя аргументами оказывается уместной. Например, вызов `Point p = new Point(0,0);` абсолютно разумен. Точка в декартовом пространстве естественным образом создается с двумя аргументами. В самом деле, вызов `new Point(0)` выглядел бы довольно странно. Однако два аргумента в нашем случае являются упорядоченными компонентами одного значения! Напротив, `outputStream` и `name` не имеют ни естественной связи, ни естественного порядка.

Даже с очевидными бинарными функциями вида `assertEquals(expected, actual)` возникают проблемы. Сколько раз вы помещали `actual` туда, где должен был находиться аргумент `expected`? Эти два аргумента не имеют естественного порядка. Последовательность `expected, actual` — не более чем условное правило, которое запоминается не сразу.

Бинарные функции не являются абсолютным злом, и вам почти наверняка придется писать их. Тем не менее следует помнить, что за их использование приходится расплачиваться, а вам стоит воспользоваться всеми доступными средствами для их преобразования в унарные. Например, можно сделать метод `writeField` членом класса `outStream`, чтобы использовать запись `outputStream.writeField(name)`. Другой вариант — преобразование `outputStream` в поле текущего класса, чтобы переменную не приходилось передавать при вызове. Также можно создать новый класс `FieldWriter`, который получает `outputStream` в конструкторе и содержит метод `write`.

Тернарные функции

Разобраться в функции с тремя аргументами значительно сложнее, чем в бинарной функции. Проблемы соблюдения порядка аргументов, приостановки чтения и игнорирования увеличиваются более чем вдвое. Я рекомендую очень хорошо подумать, прежде чем создавать тернарную функцию.

Для примера возьмем стандартную перегруженную версию `assertEquals` с тремя аргументами: `assertEquals(message, expected, actual)`. Сколько раз вы читали значение `message` и думали, что перед вами `expected`? Я сталкивался с этой конкретной тернарной функцией и задерживался на ней много раз. Более того, *каждый раз, когда я ее вижу*, мне приходится делать новый заход и вспоминать о необходимости игнорировать `message`.

С другой стороны, следующая тернарная функция не столь коварна: `assertEquals(1.0, amount, .001)`. Хотя и она не воспринимается с первого раза, в данном случае эта трудность оправдана. Всегда полезно лишний раз вспомнить, что равенство вещественных значений — понятие относительное.

Объекты как аргументы

Если функция должна получать более двух или трех аргументов, весьма вероятно, что некоторые из этих аргументов стоит упаковать в отдельном классе. Рассмотрим следующие два объявления:

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

Сокращение количества аргументов посредством создания объектов может показаться жульничеством, но это не так. Если переменные передаются совместно как единое целое (как переменные `x` и `y` в этом примере), то, скорее всего, вместе они образуют концепцию, заслуживающую собственного имени.

Списки аргументов

Иногда функция должна получать переменное количество аргументов. Для примера возьмем метод `String.format`:

```
String.format("%s worked %.2f hours.", name, hours);
```

Если все переменные аргументы считаются равноправными, как в этом примере, то их совокупность эквивалентна одному аргументу типа `List`. По этой причине функция `String.format` фактически является бинарной. И действительно, следующее объявление `String.format` подтверждает это:

```
public String format(String format, Object... args)
```

Следовательно, в данном случае действуют уже знакомые правила. Функции с переменным списком аргументов могут быть унарными, бинарными и даже тернарными, но использовать большее количество аргументов было бы ошибкой.

```
void monad(Integer... args);
void dyad(String name, Integer... args);
void triad(String name, int count, Integer... args);
```

Глаголы и ключевые слова

Выбор хорошего имени для функции способен в значительной мере объяснить смысл функции, а также порядок и смысл ее аргументов. В унарных функциях сама функция и ее аргумент должны образовывать естественную пару «глагол/существительное». Например, вызов вида `write(name)` смотрится весьма информативно. Читатель понимает, что чем бы ни было «имя» (`name`), оно куда-то «записывается» (`write`). Еще лучше запись `writeField(name)`, которая сообщает, что «имя» записывается в «поле» какой-то структуры.

Последняя запись является примером использования ключевых слов в имени функции. В этой форме имена аргументов кодируются в имени функции. Например, `assertEquals` можно записать в виде `assertExpectedEqualsActual(expected, actual)`. Это в значительной мере решает проблему запоминания порядка аргументов.

Избавьтесь от побочных эффектов

Побочные эффекты суть ложь. Ваша функция обещает делать что-то одно, но делает что-то другое, *скрытое от пользователя*. Иногда она вносит неожиданные изменения в переменные своего класса — скажем, присваивает им значения параметров, переданных функции, или глобальных переменных системы. В любом случае такая функция является коварной и вредоносной ложью, которая часто приводит к созданию противоестественных временных привязок и других зависимостей.

Для примера возьмем безвредную на первый взгляд функцию из листинга 3.6. Функция использует стандартный алгоритм для проверки пары «имя пользователя/пароль». Она возвращает `true` в случае совпадения или `false` при возникновении проблем. Но у функции также имеется побочный эффект. Сможете ли вы обнаружить его?

Листинг 3.6. UserValidator.java

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password)
{
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
  }
}
```

Разумеется, побочным эффектом является вызов `Session.initialize()`. Имя `checkPassword` сообщает, что функция проверяет пароль. Оно ничего не говорит о том, что функция инициализирует сеанс. Таким образом, тот, кто поверит имени функции, рискует потерять текущие сеансовые данные, когда он решит проверить данные пользователя.

Побочный эффект создает временную привязку. А именно, функция `checkPassword` может вызываться только в определенные моменты времени (когда инициализация сеанса может быть выполнена безопасно). Несвоевременный вызов может привести к непреднамеренной потере сеансовых данных. Временные привязки создают массу проблем, особенно когда они прячутся в побочных эффектах. Если без временной привязки не обойтись, этот факт должен быть четко оговорен в имени функции. В нашем примере функцию можно было бы переименовать в `checkPasswordAndInitializeSession`, хотя это безусловно нарушает правило «одной операции».

Выходные аргументы

Аргументы естественным образом интерпретируются как входные данные функции. Каждый, кто занимался программированием более нескольких лет, наверняка сталкивался с необходимостью дополнительной проверки аргументов, которые на самом деле оказывались выходными, а не входными. Пример:

```
appendFooter(s);
```

Присоединяет ли эта функция s в качестве завершающего блока к чему-то другому? Или она присоединяет какой-то завершающий блок к s? Является ли s входным или выходным аргументом? Конечно, можно посмотреть на сигнатуру функции и получить ответ:

```
public void appendFooter(StringBuffer report)
```

Вопрос снимается, но только после проверки объявления. Все, что заставляет обращаться к сигнатуре функции, нарушает естественный ритм чтения кода. Подобных «повторных заходов» следует избегать.

До наступления эпохи объектно-ориентированного программирования без выходных аргументов иногда действительно не удавалось обойтись. Но в ОО-языках эта проблема в целом исчезла, потому что сама функция может вызываться для выходного аргумента. Иначе говоря, функцию appendFooter лучше вызывать в виде

```
report.appendFooter();
```

В общем случае выходных аргументов следует избегать. Если ваша функция должна изменять чье-то состояние, пусть она изменяет состояние своего объекта-владельца.

Разделение команд и запросов

Функция должна что-то делать или отвечать на какой-то вопрос, но не одновременно. Либо функция изменяет состояние объекта, либо возвращает информацию об этом объекте. Совмещение двух операций часто создает путаницу. Для примера возьмем следующую функцию:

```
public boolean set(String attribute, String value);
```

Функция присваивает значение атрибуту с указанным именем и возвращает true, если присваивание прошло успешно, или false, если такой атрибут не существует. Это приводит к появлению странных конструкций вида

```
if (set("username", "unclebob"))...
```

Представьте происходящее с точки зрения читателя кода. Что проверяет это условие? Что атрибут "username" содержит ранее присвоенное значение "unclebob"? Или что проверяет атрибуту "username" успешно присвоено значение "unclebob"? Смысл невозможно вывести из самого вызова, потому что мы не знаем, чем в данном случае является слово set — глаголом или прилагательным.

Автор предполагал, что set является глаголом, но в контексте команды if это имя скорее воспринимается как прилагательное. Таким образом, команда читается в виде «Если атрибуту username ранее было присвоено значение unclebob», а не «присвоить атрибуту username значение unclebob, и если все прошло успешно, то...» Можно было бы попытаться решить проблему, переименовав функцию set в setAndCheckIfExists, но это не особенно улучшает удобочитаемость команды if. Полноценное решение заключается в отделении команды от запроса, чтобы в принципе исключить любую неоднозначность.

```
if (attributeExists("username")) {
  setAttribute("username", "unclebob");
  ...
}
```

Используйте исключения вместо возвращения кодов ошибок

Возвращение кодов ошибок функциями-командами является неочевидным нарушением принципа разделения команд и запросов. Оно поощряет использование команд в предикатных выражениях if:

```
if (deletePage(page) == E_OK)
```

Такие конструкции не страдают от смешения глаголов с прилагательными, но они приводят к созданию структур слишком глубокой вложенности. При возвращении кода ошибки возникает проблема: вызывающая сторона должна немедленно отреагировать на ошибку.

```
if (deletePage(page) == E_OK) {
  if (registry.deleteReference(page.name) == E_OK) {
    if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
      logger.log("page deleted");
    } else {
      logger.log("configKey not deleted");
    }
  } else
{
    logger.log("deleteReference from registry failed");
  }
} else {
  logger.log("delete failed");
  return E_ERROR;
}
```

С другой стороны, если вместо возвращения кодов ошибок используются исключения, то код обработки ошибок изолируется от ветви нормального выполнения и упрощается:

```
try {
  deletePage(page);
```

```
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
  }
  catch (Exception e) {
    logger.log(e.getMessage());
  }
```

Изолируйте блоки try/catch

Блоки `try/catch` выглядят весьма уродливо. Они запутывают структуру кода и смешивают обработку ошибок с нормальной обработкой. По этой причине тела блоков `try` и `catch` рекомендуется выделять в отдельные функции.

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
}

private void deletePageAndAllReferences(Page page) throws Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
  logger.log(e.getMessage());
}
```

В этом примере функция `delete` специализируется на обработке ошибок. В этой функции легко разобраться, а потом забыть о ней. Функция `deletePageAndAllReferences` специализируется на процессе полного удаления страницы. Читая ее, можно не обращать внимания на обработку ошибок. Таким образом, код нормального выполнения отделяется от кода обработки ошибок, а это упрощает его понимание и модификацию.

Обработка ошибок как одна операция

Функции должны выполнять одну операцию. Обработка ошибок — это одна операция. Значит, функция, обрабатывающая ошибки, ничего другого делать не должна. Отсюда следует, что если в функции присутствует ключевое слово `try`, то оно должно быть первым словом в функции, а после блоков `catch/finally` ничего другого быть не должно (как в предыдущем примере).

Магнит зависимостей Error.java

Возвращение кода ошибки обычно подразумевает, что в программе имеется некий класс или перечисление, в котором определяются все коды ошибок.

```
public enum Error {
  OK,
  INVALID,
  NO_SUCH,
  LOCKED,
  OUT_OF_RESOURCES,
  WAITING_FOR_EVENT;
}
```

Подобные классы называются *магнитами зависимостей*; они должны импортироваться и использоваться многими другими классами. При любых изменениях перечисления `Error` все эти классы приходится компилировать и развертывать заново[1]. Это обстоятельство создает негативную нагрузку на класс `Error`. Программистам не хочется добавлять новые ошибки, чтобы не создавать себе проблем со сборкой и развертыванием. Соответственно, вместо добавления новых кодов ошибок они предпочитают использовать старые.

Если вместо кодов ошибок использовать исключения, то новые исключения определяются *производными* от класса исключения. Их включение в программу не требует перекомпиляции или повторного развертывания[2].

Не повторяйтесь[3]

Внимательно присмотревшись к листингу 3.1, можно заметить, что один из алгоритмов повторяется в нем четыре раза: по одному разу для `SetUp`, `SuiteSetUp`, `TearDown` и `SuiteTearDown`. Обнаружить это дублирование нелегко, потому что четыре вхождения алгоритма перемешаны с другим кодом, а в дублировании фрагментов имеются некоторые различия. Тем не менее дублирование создает проблемы, потому что оно увеличивает объем кода, а при изменении алгоритма вам придется вносить изменения сразу в четырех местах. Также вчетверо возрастает вероятность ошибки.

---

[1] Люди, считавшие, что они смогут обойтись без перекомпиляции и повторного развертывания, были пойманы и сурово наказаны.

[2] Пример принципа открытости/закрытости (OCP) [PPP02].

[3] Принцип DRY [PRAG].

В листинге 3.7 дублирование устраняется при помощи метода include. Снова прочитайте код и обратите внимание, насколько проще читается весь модуль после устранения дублирования.

Дублирование иногда считается корнем всего зла в программировании. Было создано много приемов и методологий, направленных на контроль и устранение дублирования. Возьмем хотя бы нормальные формы баз данных Кодда, предназначенные для устранения дубликатов в данных. Или другой пример: объектно-ориентированные языки помогают сконцентрировать в базовых классах код, который в других обстоятельствах мог бы дублироваться в разных местах. Структурное программирование, аспектно-ориентированное программирование, компонентно-ориентированное программирование — все эти технологии отчасти являются стратегиями борьбы с дублированием. Похоже, с момента изобретения подпрограмм все новшества в разработке программного обеспечения были направлены исключительно на борьбу с дублированием в исходном коде.

Структурное программирование

Некоторые программисты следуют правилам структурного программирования, изложенным Эдгаром Дейкстрой [SP72]. Дейкстра считает, что каждая функция и каждый блок внутри функции должны иметь одну точку входа и одну точку выхода. Выполнение этого правила означает, что функция должна содержать только одну команду return, в циклах не должны использоваться команды break или continue, а команды goto не должны использоваться никогда и ни при каких условиях.

Хотя мы с симпатией относимся к целям и методам структурного программирования, в очень компактных функциях эти правила не приносят особой пользы. Только при увеличении объема функций их соблюдение обеспечивает существенный эффект.

Итак, если ваши функции остаются очень компактными, редкие вкрапления множественных return, команд break и continue не принесут вреда, а иногда даже повышают выразительность по сравнению с классической реализацией с одной точкой входа и одной точкой выхода. С другой стороны, команда goto имеет смысл только в больших функциях, поэтому ее следует избегать.

Как научиться писать такие функции?

Написание программ сродни любому другому виду письменной работы. Когда вы пишете статью или доклад, вы сначала излагаете свои мысли, а затем «причесываете» их до тех пор, пока они не будут хорошо читаться. Первый вариант может

быть неуклюжим и нелогичным; вы переделываете, дополняете и уточняете его, пока он не будет читаться так, как вам хочется.

Когда я пишу свои функции, они получаются длинными и сложными. В них встречаются многоуровневые отступы и вложенные циклы. Они имеют длинные списки аргументов. Имена выбираются хаотично, а в коде присутствуют дубликаты. Но у меня также имеется пакет модульных тестов для всех этих неуклюжих строк до последней.

Итак, я начинаю «причесывать» и уточнять свой код, выделять новые функции, изменять имена и устранять дубликаты. Я сокращаю методы и переупорядочиваю их. Иногда приходится ломать целые классы, но при этом слежу за тем, чтобы все тесты выполнялись успешно.

В конечном итоге у меня остаются функции, построенные по правилам, изложенным в этой главе. Я не записываю их так с самого начала. И вообще не думаю, что кому-нибудь это под силу.

Завершение

Каждая система строится в контексте языка, отражающего специфику предметной области и разработанного программистами для описания этой системы. В этом языке функции играют роль глаголов, а классы — существительных. Не стоит полагать, что мы возвращаемся к кошмарной древней практике, по которой существительные и глаголы в документе с требованиями становились первыми кандидатами для классов и функций системы. Скорее речь идет о гораздо более древней истине. Искусство программирования является (и всегда было) искусством языкового проектирования.

Опытные программисты рассматривают систему как историю, которую они должны рассказать, а не как программу, которую нужно написать. Они используют средства выбранного ими языка программирования для конструирования гораздо более богатого и выразительного языка, подходящего для этого повествования. Частью этого предметно-ориентированного языка является иерархия функций, которые описывают все действия, выполняемые в рамках системы. В результате искусной рекурсии эти действия формулируются на том самом предметно-ориентированном языке, который они определяют для изложения своей маленькой части истории.

Эта глава была посвящена механике качественного написания функций. Если вы будете следовать этим правилам, ваши функции будут короткими, удачно названными и хорошо организованными. Но никогда не забывайте, что ваша настоящая цель — «рассказать историю» системы, а написанные вами функции должны четко складываться в понятный и точный язык, который поможет вам в этом.

Листинг 3.7. SetupTeardownIncluder.java

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

public class SetupTeardownIncluder {
  private PageData pageData;
  private boolean isSuite;
  private WikiPage testPage;
  private StringBuffer newPageContent;
  private PageCrawler pageCrawler;

  public static String render(PageData pageData) throws Exception {
    return render(pageData, false);
  }

  public static String render(PageData pageData, boolean isSuite)
    throws Exception {
    return new SetupTeardownIncluder(pageData).render(isSuite);
  }

  private SetupTeardownIncluder(PageData pageData) {
    this.pageData = pageData;
    testPage = pageData.getWikiPage();
    pageCrawler = testPage.getPageCrawler();
    newPageContent = new StringBuffer();
  }

  private String render(boolean isSuite) throws Exception {
    this.isSuite = isSuite;
    if (isTestPage())
      includeSetupAndTeardownPages();
    return pageData.getHtml();
  }

  private boolean isTestPage() throws Exception {
    return pageData.hasAttribute("Test");
  }

  private void includeSetupAndTeardownPages() throws Exception {
    includeSetupPages();
    includePageContent();
    includeTeardownPages();
    updatePageContent();
  }

  private void includeSetupPages() throws Exception {
    if (isSuite)
      includeSuiteSetupPage();
    includeSetupPage();
  }
```

```
private void includeSuiteSetupPage() throws Exception {
  include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
}

private void includeSetupPage() throws Exception {
  include("SetUp", "-setup");
}

private void includePageContent() throws Exception {
  newPageContent.append(pageData.getContent());
}

private void includeTeardownPages() throws Exception {
  includeTeardownPage();
  if (isSuite)
    includeSuiteTeardownPage();
}

private void includeTeardownPage() throws Exception {
  include("TearDown", "-teardown");
}

private void includeSuiteTeardownPage() throws Exception {
  include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
}

private void updatePageContent() throws Exception {
  pageData.setContent(newPageContent.toString());
}

private void include(String pageName, String arg) throws Exception {
  WikiPage inheritedPage = findInheritedPage(pageName);
  if (inheritedPage != null) {
    String pagePathName = getPathNameForPage(inheritedPage);
    buildIncludeDirective(pagePathName, arg);
  }
}

private WikiPage findInheritedPage(String pageName) throws Exception {
  return PageCrawlerImpl.getInheritedPage(pageName, testPage);
}

private String getPathNameForPage(WikiPage page) throws Exception {
  WikiPagePath pagePath = pageCrawler.getFullPath(page);
  return PathParser.render(pagePath);
}

private void buildIncludeDirective(String pagePathName, String arg) {
  newPageContent
    .append("\n!include ")
    .append(arg)
    .append(" .")
```

продолжение ⇩

Листинг 3.7 (продолжение)

```
        .append(pagePathName)
        .append("\n");
    }
}
```

Литература

[KP78]: Kernighan and Plaugher, The Elements of Programming Style, 2d. ed., McGraw-Hill, 1978.

[PPP02]: Robert C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, 2002.

[GOF]: Design Patterns: Elements of Reusable Object Oriented Software, Gamma et al., Addison-Wesley, 1996.

[PRAG]: The Pragmatic Programmer, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

[SP72]: Structured Programming, O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Academic Press, London, 1972.

4 Комментарии

Не комментируйте плохой код — перепишите его.
Брайан У. Керниган и П. Дж. Плауэр[1]

Ничто не помогает так, как уместный комментарий. Ничто не загромождает модуль так, как бессодержательные и безапелляционные комментарии. Ничто не приносит столько вреда, как старый, утративший актуальность комментарий, распространяющий ложь и дезинформацию.

Комментарии — не список Шиндлера. Не стоит относиться к ним как к «абсолютному добру». На самом деле комментарии в лучшем случае являются неизбежным злом. Если бы языки программирования были достаточно выразительными или если бы мы умели искусно пользоваться этими языками для выражения

[1] [KP78], p. 144.

своих намерений, то потребность в комментариях резко снизилась бы, а может, и вовсе сошла «на нет».

Грамотное применение комментариев должно компенсировать нашу неудачу в выражении своих мыслей в коде. Обратите внимание на слово «неудачу». Я абсолютно серьезно. Комментарий — всегда признак неудачи. Мы вынуждены использовать комментарии, потому что нам не всегда удается выразить свои мысли без них, однако гордиться здесь нечем.

Итак, вы оказались в ситуации, в которой необходимо написать комментарий? Хорошенько подумайте, нельзя ли пойти по другому пути и выразить свои намерения в коде. Каждый раз, когда вам удается это сделать, — похлопайте себя по плечу. Каждый раз, когда вы пишете комментарий, — поморщитесь и ощутите свою неудачу.

Почему я так настроен против комментариев? Потому что они лгут. Не всегда и не преднамеренно, но это происходит слишком часто. Чем древнее комментарий, чем дальше он расположен от описываемого им кода, тем больше вероятность того, что он просто неверен. Причина проста: программисты не могут нормально сопровождать комментарии.

Программный код изменяется и эволюционирует. Его фрагменты перемещаются из одного места в другое, раздваиваются, размножаются и сливаются. К сожалению, комментарии не всегда сопровождают их — и не всегда *могут* сопровождать их. Слишком часто комментарии отделяются от описываемого ими кода и превращаются в пометки непонятной принадлежности, с постоянно снижающейся точностью. Посмотрите, что произошло с этим комментарием и той строкой, которую он должен описывать:

```
MockRequest request;
private final String HTTP_DATE_REGEXP =
    "[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s"+
    "[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT";
private Response response;
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;
// Пример: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Другие переменные экземпляра (вероятно, добавленные позднее) вклинились между константой `HTTP_DATE_REGEXP` и пояснительным комментарием.

На это можно возразить, что программисты должны быть достаточно дисциплинированными, чтобы поддерживать в своем коде актуальные, точные и релевантные комментарии. Согласен, должны. Но я бы предпочел, чтобы вместо этого программист постарался сделать свой код настолько четким и выразительным, чтобы комментарии были попросту не нужны.

Неточные комментарии гораздо вреднее, чем полное отсутствие комментариев. Они обманывают и сбивают с толку. Они создают у программиста невыполнимые ожидания. Они устанавливают устаревшие правила, которые не могут (или не должны) соблюдаться в будущем.

Истину можно найти только в одном месте: в коде. Только код может правдиво сообщить, что он делает. Это единственный источник действительно достоверной информации. Таким образом, хотя комментарии иногда необходимы, мы потратим немало усилий для того, чтобы свести их использование к минимуму.

Комментарии не компенсируют плохого кода

Одной из распространенных причин для написания комментариев является низкое качество кода. Вы пишете модуль и видите, что код получился запутанным и беспорядочным. Вы знаете, что разобраться в нем невозможно. Поэтому вы говорите себе: «О, да это стоит прокомментировать!» Нет! Лучше исправьте свой код!

Ясный и выразительный код с минимумом комментариев гораздо лучше громоздкого, сложного кода с большим количеством комментариев. Не тратьте время на написание комментариев, объясняющих созданную вами путаницу, — лучше потратьте его на исправление.

Объясните свои намерения в коде

И все же в некоторых ситуациях код оказывается не лучшим средством для объяснений. К сожалению, многие программисты воспринимают этот факт иначе: они полагают, что код *никогда* не является хорошим средством для объяснений. А это, разумеется, неправда. С каким бы кодом вы предпочли работать — с таким:

```
// Проверить, положена ли работнику полная премия
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

Или с таким:

```
if (employee.isEligibleForFullBenefits())
```

Чтобы объяснить большую часть ваших намерений в коде, достаточно нескольких секунд. Нередко задача сводится с созданию функции, которая сообщает то же, что и комментарий, который вы собираетесь написать.

Хорошие комментарии

Впрочем, необходимые и полезные комментарии все же существуют. Мы рассмотрим несколько примеров, которые, на мой взгляд, стоят затраченных на них битов. И все же следует помнить, что по-настоящему хороший комментарий — тот, без которого вам удастся обойтись.

Юридические комментарии

Иногда корпоративные стандарты кодирования заставляют нас вставлять комментарии по юридическим соображениям. Например, заявление об авторских правах — необходимая информация, которая вполне может размещаться в комментарии в начале каждого файла с исходным кодом.

Ниже приведен стандартный заголовок комментария, который вставляется в начало каждого исходного файла в FitNesse. К счастью, наша IDE автоматически сворачивает этот комментарий, чтобы он не загромождал экран.

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// Публикуется на условиях лицензии GNU General Public License версии 2 и выше.
```

Такие комментарии не должны представлять собой комментарии или юридические трактаты. Вместо того чтобы перечислять в комментарии все условия, по возможности ограничьтесь ссылкой на стандартную лицензию или другой внешний документ.

Информативные комментарии

Иногда бывает полезно включить в комментарий пояснение к коду. Возьмем следующий комментарий, объясняющий возвращаемое значение абстрактного метода:

```
// Возвращает тестируемый экземпляр Responder.
protected abstract Responder responderInstance();
```

Такие комментарии бывают полезными, но там, где это возможно, информацию лучше передавать в имени функции. Например, в данном примере вполне можно обойтись и без комментария — достаточно переименовать функцию в `responderBeingTested`.

А вот другой, более уместный пример:

```
// Поиск по формату: kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
       "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

На этот раз комментарий сообщает, что регулярное выражение предназначено для идентификации времени и даты, отформатированных функцией `SimpleDateFormat.format` с заданной форматной строкой. И все же код стал бы лучше (и понятнее), если бы мы переместили этот код в специальный класс, преобразующий форматы даты и времени. Тогда комментарий, вероятно, стал бы излишним.

Представление намерений

Иногда комментарий выходит за рамки полезной информации о реализации и описывает намерения, заложенные в решение. В следующем примере мы видим интересный пример архитектурного решения, документированного в комментарии. Автор решил, что при сравнении двух объектов объекты его класса должны находиться в порядке сортировки выше, чем объекты любого другого класса.

```
public int compareTo(Object o)
{
  if(o instanceof WikiPagePath)
  {
    WikiPagePath p = (WikiPagePath) o;
    String compressedName = StringUtil.join(names, "");
    String compressedArgumentName = StringUtil.join(p.names, "");
    return compressedName.compareTo(compressedArgumentName);
  }
  return 1; // Больше, потому что относится к правильному типу.
}
```

Или другой, еще лучший пример. Возможно, вы не согласитесь с тем, как программист решает проблему, но по крайней мере вы знаете, что он пытается сделать.

```
public void testConcurrentAddWidgets() throws Exception {
  WidgetBuilder widgetBuilder =
    new WidgetBuilder(new Class[]{BoldWidget.class});
    String text = "'''bold text'''";
    ParentWidget parent =
      new BoldWidget(new MockWidgetRoot(), "'''bold text'''");
    AtomicBoolean failFlag = new AtomicBoolean();
    failFlag.set(false);

    // Мы пытаемся спровоцировать "состояние гонки",
    // создавая большое количество программных потоков.
    for (int i = 0; i < 25000; i++) {
      WidgetBuilderThread widgetBuilderThread =
        new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);
      Thread thread = new Thread(widgetBuilderThread);
      thread.start();
    }
    assertEquals(false, failFlag.get());
}
```

Прояснение

Иногда смысл загадочного аргумента или возвращаемого значения бывает удобно преобразовать в удобочитаемую форму. В общем случае лучше подумать, как сделать так, чтобы этот аргумент или возвращаемое значение говорили сами за себя; но если они являются частью стандартной библиотеки или используются в коде, который вы не можете изменить, то пояснительный комментарий может быть весьма полезным.

```
public void testCompareTo() throws Exception
{
  WikiPagePath a = PathParser.parse("PageA");
  WikiPagePath ab = PathParser.parse("PageA.PageB");
  WikiPagePath b = PathParser.parse("PageB");
  WikiPagePath aa = PathParser.parse("PageA.PageA");
  WikiPagePath bb = PathParser.parse("PageB.PageB");
```

```
  WikiPagePath ba = PathParser.parse("PageB.PageA");
  assertTrue(a.compareTo(a) == 0);    // a == a
  assertTrue(a.compareTo(b) != 0);    // a != b
  assertTrue(ab.compareTo(ab) == 0);  // ab == ab
  assertTrue(a.compareTo(b) == -1);   // a < b
  assertTrue(aa.compareTo(ab) == -1); // aa < ab
  assertTrue(ba.compareTo(bb) == -1); // ba < bb
  assertTrue(b.compareTo(a) == 1);    // b > a
  assertTrue(ab.compareTo(aa) == 1);  // ab > aa
  assertTrue(bb.compareTo(ba) == 1);  // bb > ba
}
```

Конечно, при этом возникает существенный риск, что пояснительный комментарий окажется неверным. Просмотрите код примера и убедитесь, как трудно проверить его правильность. Это объясняет как необходимость пояснений, так и связанный с ними риск. Итак, прежде чем писать такие комментарии, убедитесь в том, что лучшего способа не существует, и еще внимательнее следите за их правильностью.

Предупреждения о последствиях

Иногда бывает полезно предупредить других программистов о нежелательных последствиях от каких-либо действий. Например, следующий комментарий объясняет, почему конкретный тестовый сценарий был отключен:

```
// Не запускайте, если только не располагаете
// излишками свободного времени.
public void _testWithReallyBigFile()
{
  writeLinesToFile(10000000);
  response.setBody(testFile);
  response.readyToSend(this);
  String responseString = output.toString();
  assertSubString("Content-Length: 1000000000", responseString);
  assertTrue(bytesSent > 1000000000);
}
```

Конечно, в наше время тестовый сценарий следовало бы отключить при помощи атрибута `@Ignore` с соответствующей пояснительной строкой: `@Ignore("Слишком долго выполняется")`. Но до появления JUnit 4 запись с начальным символом подчеркивания перед именем метода считалась стандартной. Комментарий, при всей его несерьезности, хорошо доносит свое сообщение до читателя.

А вот другой, более выразительный пример:

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
  // Класс SimpleDateFormat не является потоково-безопасным,
  // поэтому экземпляры должны создаваться независимо друг от друга.
```

```
  SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM  yyyy HH:mm:ss z");
  df.setTimeZone(TimeZone.getTimeZone("GMT"));
  return df;
}
```

Возможно, вы возразите, что у задачи есть и более удачные решения. Пожалуй, я соглашусь с вами. Однако комментарий в том виде, в котором он здесь приведен, выглядит абсолютно разумно. По крайней мере он помешает излишне ретивому программисту использовать статический инициализатор по соображениям эффективности.

Комментарии TODO

Иногда бывает полезно оставить заметки «на будущее» в форме комментариев `//TODO`. В следующем примере комментарий TODO объясняет, почему функция имеет вырожденную реализацию и что она должна делать в будущем.

```
// TODO - На данный момент эта функция не используется.
// Ситуация изменится при переходе к отладочной модели.
protected VersionInfo makeVersion() throws Exception
{
  return null;
}
```

Комментарии TODO напоминают о том, что, по мнению программиста, сделать необходимо, но по какой-то причине нельзя сделать прямо сейчас. Например, комментарий может напомнить о необходимости удаления устаревшей функции или предложить кому-то другому поучаствовать в решении проблемы — скажем, придумать более удачное имя или внести изменения, зависящие от запланированного события. Впрочем, чем бы ни был комментарий TODO, это не повод оставлять плохой код в системе.

В наши дни в любой хорошей рабочей среде имеется функция поиска всех комментариев TODO, так что потеря таких комментариев маловероятна. И все же код не должен загромождаться лишними комментариями TODO. Регулярно просматривайте их и удаляйте те, которые потеряли актуальность.

Усиление

Комментарий может подчеркивать важность обстоятельства, которое на первый взгляд кажется несущественным.

```
String listItemContent = match.group(3).trim();
// Вызов trim() очень важен. Он удаляет начальные пробелы,
// чтобы строка успешно интерпретировалась как список.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

Комментарии Javadoc в общедоступных API

С хорошо документированным общедоступным API приятно и легко работать. Документация Javadoc для стандартной библиотеки Java убедительно доказывает это утверждение. Без нее писать Java-программы было бы в лучшем случае непросто.

Если вы разрабатываете API для общего пользования, несомненно, для него следует написать хорошие комментарии Javadoc. Однако не забывайте об остальных советах этой главы. Комментарии Javadoc могут быть такими же и недостоверными и лживыми, как и любые другие комментарии.

Плохие комментарии

Большинство комментариев относится именно к этой категории. Обычно такие комментарии представляют собой «подпорки» для некачественного кода или оправдания сомнительных решений, а их текст напоминает рассуждения вслух самого программиста.

Бормотание

Не стоит лепить комментарии «на скорую руку» только потому, что вам кажется, что это уместно или этого требует процесс. Если уж вы решаете написать комментарий, не жалейте времени и напишите лучший из всех возможных комментариев.

Например, следующий фрагмент я обнаружил в FitNesse. В самом деле, комментарий здесь бы пригодился. Но автор то ли торопился, то ли не придал особого значения тому, что он пишет. Его бормотание оставляет читателя в недоумении:

```
public void loadProperties()
{
  try
  {
      String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
      FileInputStream propertiesStream = new FileInputStream(propertiesPath);
      loadedProperties.load(propertiesStream);
  }
  catch(IOException e)
  {
    // Если нет файла свойств, загружаются настройки по умолчанию
  }
}
```

Что означает комментарий в блоке `catch`? Очевидно, он что-то означал для автора, но для читателя этот смысл не доходит. Видимо, если мы получаем `IOException`, это означает, что файл свойств отсутствует; в этом случае должны загружаться

все настройки по умолчанию. Но кто загружает эти настройки? Были ли они загружены перед вызовом `loadProperties.load`? Или вызов `loadProperties.load` перехватывает исключение, загружает настройки по умолчанию, а затем передает исключение нам, чтобы мы могли его проигнорировать? Или `loadProperties.load` загружает настройки по умолчанию до того, как вы попытались загрузить файл? Автор пытался успокоить себя относительно того факта, что он оставил блок `catch` пустым? Или — и это самая пугающая возможность — автор хотел напомнить себе, что позднее нужно вернуться и написать код загрузки настроек по умолчанию?

Чтобы разобраться в происходящем, нам остается только изучить код других частей системы. Любой комментарий, смысл которого приходится искать в других модулях, не несет полезной информации и не стоит битов, затраченных на его написание.

Избыточные комментарии

В листинге 4.1 приведена простая функция с совершенно лишним заголовочным комментарием. Вероятно, чтение комментария займет больше времени, чем чтение самого кода.

Листинг 4.1. waitForClose

```
// Вспомогательный метод; возвращает управление, когда значение this.closed истинно.
// Инициирует исключение при достижении тайм-аута.
public synchronized void waitForClose(final long timeoutMillis)
throws Exception
{
  if(!closed)
  {
    wait(timeoutMillis);
    if(!closed)
      throw new Exception("MockResponseSender could not be closed");
  }
}
```

Какой цели достигает этот комментарий? Конечно, он несет не больше информации, чем программный код. Он не объясняет код, не предоставляет обоснований и не раскрывает намерений. Он читается не проще, чем сам код. Более того, комментарий уступает коду в точности и навязывает читателю эту неточность взамен истинного понимания. Он напоминает жуликоватого торговца подержанными машинами, уверяющего, что вам незачем заглядывать под капот.

А теперь рассмотрим легион бесполезных, избыточных комментариев Javadoc из листинга 4.2, позаимствованных из Tomcat. Эти комментарии только загромождают код и скрывают его смысл. Никакой пользы для документирования от них нет. Что еще хуже, я привел только несколько начальных комментариев — в этом модуле их намного больше.

Листинг 4.2. ContainerBase.java (Tomcat)

```
public abstract class ContainerBase
  implements Container, Lifecycle, Pipeline,
  MBeanRegistration, Serializable {

  /**
   * Задержка процессора для этого компонента.
   */
  protected int backgroundProcessorDelay = -1;

  /**
   * Поддержка событий жизненного цикла для этого компонента.
   */
  protected LifecycleSupport lifecycle =
    new LifecycleSupport(this);

  /**
   * Слушатели контейнерных событий для этого контейнера.
   */
  protected ArrayList listeners = new ArrayList();

  /**
   * Реализация загрузчика, связанная с контейнером.
   */
  protected Loader loader = null;

  /**
   * Реализация журнального компонента, связанная с контейнером.
   */
  protected Log logger = null;

  /**
   * Имя журнального компонента.
   */
  protected String logName = null;

  /**
   * Реализация менеджера, связанная с контейнером.
   */
  protected Manager manager = null;

  /**
   * Кластер, связанный с контейнером.
   */
  protected Cluster cluster = null;

  /**
   * Удобочитаемое имя контейнера.
   */
  protected String name = null;
```

```
/**
 * Родительский контейнер, по отношению к которому
 * данный контейнер является дочерним.
 */
protected Container parent = null;

/**
 * Загрузчик родительского класса, задаваемый при назначении загрузчика.
 */
protected ClassLoader parentClassLoader = null;

/**
 * Объект Pipeline, связанный с данным контейнером.
 */
protected Pipeline pipeline = new StandardPipeline(this);

/**
 * Объект Realm, связанный с контейнером.
 */
protected Realm realm = null;

/**
 * Объект ресурсов DirContext, связанный с контейнером
 */
protected DirContext resources = null;
```

Недостоверные комментарии

Иногда с самыми лучшими намерениями программист делает в комментариях заявления, неточные и не соответствующие истине. Еще раз взгляните на совершенно лишний, но при этом слегка вводящий в заблуждение комментарий из листинга 4.1.

А вы нашли, в чем этот комментарий обманывает читателя? Метод не возвращает управление, *когда* значение `this.closed` становится истинным. Он возвращает управление, если значение `this.closed` истинно; в противном случае метод ожидает истечения тайм-аута, а затем инициирует исключение, если значение `this.closed` так и не стало истинным.

Эта крошечная дезинформация в комментарии, который читается хуже, чем сам код, может заставить другого программиста вызвать функцию в предположении, что она вернет управление сразу же, как только значение `this.closed` станет истинным. После этого бедный программист будет долго отлаживать программу, пытаясь понять, почему его код выполняется так медленно.

Обязательные комментарии

Правила, говорящие, что каждая функция должна иметь комментарий Javadoc или что каждая переменная должна быть помечена комментарием, — обычная глупость. Такие комментарии только загромождают код, распространяют недостоверную информацию и вызывают общую путаницу и дезориентацию.

Например, требование обязательного комментария Javadoc для каждой функции приводит к появлению монстров вроде листинга 4.3. Бессмысленные комментарии не приносят никакой пользы. Они только запутывают код, повышая риск обмана и недоразумений.

Листинг 4.3.

```
/**
 *
 * @param title  Название диска
 * @param author Автор диска
 * @param tracks Количество дорожек на диске
 * @param durationInMinutes Продолжительность воспроизведения в минутах
 */
public void addCD(String title, String author,
                  int tracks, int durationInMinutes) {
  CD cd = new CD();
  cd.title = title;
  cd.author = author;
  cd.tracks = tracks;
  cd.duration = duration;
  cdList.add(cd);
}
```

Журнальные комментарии

Некоторые программисты добавляют комментарий в начало модуля при каждом его редактировании. Такие комментарии накапливаются, образуя своего рода журнал всех вносимых изменений. Я видел модули, в которых эти журнальные записи растягивались на десятки страниц.

```
* Изменения (начиная с 11 октября 2001)
* -------------------------
* 11.10.2001  : Реорганизация класса и его перемещение в новый пакет
*               com.jrefinery.date (DG);
* 05.11.2001  : Добавление метода getDescription(), устранение класса
*               NotableDate (DG);
* 12.11.2001  : С устранением класса NotableDate IBD требует включения
*               метода setDescription() (DG); исправление ошибок
*               в методах getPreviousDayOfWeek(), getFollowingDayOfWeek()
*               и getNearestDayOfWeek() (DG);
* 05.12.2001  : Исправление ошибки в классе SpreadsheetDate (DG);
```

```
 * 29.05.2002  : Перемещение констант месяцев в отдельный интерфейс
 *               (MonthConstants) (DG);
 * 27.08.2002  : Исправление ошибки в методе addMonths() с подачи N???levka Petr
(DG);
 * 03.10.2002  : Исправление ошибок по сообщениям Checkstyle (DG);
 * 13.03.2003  : Реализация Serializable (DG);
 * 29.05.2003  : Исправление ошибки в методе addMonths (DG);
 * 04.09.2003  : Реализация Comparable.  Обновление isInRange javadocs (DG);
 * 05.01.2005  : Исправление ошибки в методе addYears() (1096282) (DG);
```

Когда-то создание и сопровождение журнальных записей в начале каждого модуля было оправдано. У нас еще не было систем управления исходным кодом, которые делали это за нас. В наши дни длинные журналы только загромождают и усложняют код. Их следует полностью удалить из ваших программ.

Шум

Также в программах нередко встречаются комментарии, не содержащие ничего, кроме «шума». Они лишь утверждают очевидное, не предоставляя никакой новой информации.

```
/**
 * Конструктор по умолчанию.
 */
protected AnnualDateRule() {
}
```

Да неужели? А как насчет этого:

```
/** День месяца. */
    private int dayOfMonth;
```

И наконец, апофеоз избыточности:

```
/**
 * Возвращает день месяца.
 *
 * @return день месяца.
 */
public int getDayOfMonth() {
  return dayOfMonth;
}
```

Эти комментарии настолько бесполезны, что мы учимся не обращать на них внимания. В процессе чтения кода наш взгляд просто скользит мимо них. Рано или поздно код вокруг таких комментариев изменяется, и они начинают лгать.

Первый комментарий в листинге 4.4 кажется уместным. Он объясняет, почему блок `catch` игнорируется. Но второй комментарий не несет полезной информации. Видимо, программист настолько вышел из себя при написании этих блоков `try/catch` в этой функции, что ему понадобилось «выпустить пар».

Листинг 4.4. startSending

```
private void startSending()
{
  try
  {
    doSending();
  }
  catch(SocketException e)
  {
    // Нормально. Кто-то прервал запрос.
  }
  catch(Exception e)
  {
    try
    {
      response.add(ErrorResponder.makeExceptionString(e));
      response.closeAll();
    }
    catch(Exception e1)
    {
      // Ну хватит уже!
    }
  }
}
```

Вместо того чтобы давать выход чувствам в бесполезном комментарии, программисту следовало понять, что раздражение можно было снять улучшением структуры кода. Ему стоило направить свою энергию на выделение последнего блока try/catch в отдельную функцию, как показано в листинге 4.5.

Листинг 4.5. startSending (переработанная версия)

```
private void startSending()
{
  try
  {
    doSending();
  }
  catch(SocketException e)
  {
    // Нормально. Кто-то прервал запрос.
  }
  catch(Exception e)
  {
    addExceptionAndCloseResponse(e);
  }
}

private void addExceptionAndCloseResponse(Exception e)
{
  try
```

```
  {
    response.add(ErrorResponder.makeExceptionString(e));
    response.closeAll();
  }
  catch(Exception e1)
  {
  }
}
```

Искушение создать очередной «шумовой комментарий» следует заменить решимостью очистить код. Вы сами увидите, что это сделает вашу работу более приятной и эффективной.

Опасный шум

Комментарии Javadoc тоже бывают «шумовыми». Какую пользу приносят следующие комментарии (из хорошо известной библиотеки, распространяемой с открытым кодом)? Ответ: никакой. Это избыточные шумовые комментарии, вызванные неуместным желанием как-то документировать свои действия.

```
/** Имя. */
private String name;
/** Версия. */
private String version;
/** Название лицензии. */
private String licenceName;
/** Версия. */
private String info;
```

Прочитайте эти комментарии повнимательнее. Заметили ошибку копирования/вставки? Если авторы не следят за ними в момент написания (или вставки), то как можно ожидать, что эти комментарии принесут пользу читателю?

Не используйте комментарии там, где можно использовать функцию или переменную

Возьмем следующий фрагмент кода:

```
// Зависит ли модуль из глобального списка <mod> от подсистемы,
// частью которой является наш код?
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))
```

Его можно было бы перефразировать без комментария в следующем виде:

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

Возможно (хотя и маловероятно), автор исходного кода сначала написал комментарий, а затем — соответствующий ему код. Но после этого автор должен был переработать свой код, как это сделал я, чтобы комментарий можно было удалить.

Позиционные маркеры

Некоторые программисты любят отмечать определенные позиции в исходных файлах. Например, недавно я обнаружил в одной из просматриваемых программ следующую строку:

```
// Действия //////////////////////////////////
```

В отдельных случаях объединение функций под такими заголовками имеет смысл. Но в общем случае они составляют балласт, от которого следует избавиться — особенно от назойливой серии косых черт в конце.

Взгляните на дело под таким углом: заголовки привлекают внимание только в том случае, если они встречаются не слишком часто. Используйте их умеренно и только тогда, когда они приносят ощутимую пользу. При слишком частом употреблении заголовков читатель воспринимает их как фоновый шум и перестает обращать на них внимание.

Комментарии за закрывающей фигурной скобкой

Иногда программисты размещают специальные комментарии за закрывающими фигурными скобками, как в листинге 4.6. Применение таких комментариев оправдано в длинных функциях с многоуровневой вложенностью, но они только загромождают компактные специализированные функции, которым мы отдает предпочтение. Итак, если у вас возникает желание прокомментировать закрывающие фигурные скобки, лучше постарайтесь укоротить свои функции.

Листинг 4.6. wc.java

```
public class wc {
  public static void main(String[] args) {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String line;
    int lineCount = 0;
    int charCount = 0;
    int wordCount = 0;
    try {
      while ((line = in.readLine()) != null) {
        lineCount++;
        charCount += line.length();
        String words[] = line.split("\\W");
        wordCount += words.length;
      } //while
      System.out.println("wordCount = " + wordCount);
      System.out.println("lineCount = " + lineCount);
      System.out.println("charCount = " + charCount);
    } // try
    catch (IOException e) {
      System.err.println("Error:" + e.getMessage());
    } //catch
  } //main
}
```

Ссылки на авторов

```
/* Добавлено Риком */
```

Системы контроля исходного кода отлично запоминают, кто и когда внес то или иное исправление. Нет необходимости загрязнять код подобными ссылками. Может показаться, что такие комментарии помогают другим определить, с кем следует обсуждать данный фрагмент кода. Однако в действительности эти комментарии остаются в коде на долгие годы и со временем становятся все менее точными и актуальными.

И снова лучшим источником подобной информации является система контроля исходного кода.

Закомментированный код

В программировании редко встречаются привычки более отвратительные, чем закрытие комментариями неиспользуемого кода. Никогда не делайте этого!

```
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount()));
```

У других программистов, видящих закомментированный код, не хватает храбрости удалить его. Они полагают, что код оставлен не зря и слишком важен для удаления. В итоге закомментированный код скапливается, словно осадок на дне бутылки плохого вина.

Следующий код взят из общих модулей Apache:

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
  writeEnd();
  this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
}
else {
    this.pngBytes = null;
}
return this.pngBytes;
```

Почему эти две строки кода закомментированы? Они важны? Их оставили как напоминание о будущих изменениях? Или это «хлам», который кто-то закомментировал сто лет назад и не удосужился убрать из программы?

В 60-е годы закомментированный код мог быть действительно полезен. Но с тех пор у нас давно появились хорошие системы контроля исходного кода. Эти системы запоминают изменения в коде за нас. Нам уже не нужно закрывать их комментариями. Просто удалите ненужный код. Он никуда не исчезнет. Честное слово.

Комментарии HTML

Как видно из следующего фрагмента, HTML в комментариях к исходному коду выглядит отвратительно. Он затрудняет чтение комментариев именно там, где они должны легко читаться — в редакторе/IDE. Если комментарии должны извлекаться внешним инструментом (например, Javadoc) для отображения в вебстранице, то за украшение комментариев соответствующим кодом HTML должен отвечать этот инструмент, а не программист.

```
/**
 * Задача для запуска тестов.
 * Задача запускает тесты fitnesse и публикует результаты.
 * <p/>
 * <pre>
 * Usage:
 * &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
 *     classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot;
 *     classpathref=&quot;classpath&quot; /&gt;
 * OR
 * &lt;taskdef classpathref=&quot;classpath&quot;
 *             resource=&quot;tasks.properties&quot; /&gt;
 * <p/>
 * &lt;execute-fitnesse-tests
 *     suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;
 *     fitnesseport=&quot;8082&quot;
 *     resultsdir=&quot;${results.dir}&quot;
 *     resultshtmlpage=&quot;fit-results.html&quot;
 *     classpathref=&quot;classpath&quot; /&gt;
 * </pre>
 */
```

Нелокальная информация

Если вы должны написать комментарий, проследите за тем, чтобы он описывал находящийся поблизости код. Не излагайте информацию системного уровня в контексте локального комментария. Примером служит приведенный ниже комментарий Javadoc. Не считая того факта, что комментарий ужасающе избыточен, в него также включена информация о порте по умолчанию, притом что функция никоим образом не может управлять этим значением. И конечно, ничто не гарантирует, что комментарий будет изменен при изменении кода, в котором это значение определяется.

```
/**
 * Порт, на котором будет работать fitnesse. По умолчанию <b>8082</b>.
 *
 * @param fitnessePort
 */
public void setFitnessePort(int fitnessePort)
{
  this.fitnessePort = fitnessePort;
}
```

Слишком много информации

Не включайте в комментарии интересные исторические дискуссии или описания подробностей, не относящиеся к делу. Следующий комментарий был извлечен из модуля, который должен был проверять, что функция кодирует и декодирует данные в формате base64. Читателю кода совершенно не нужна заумная информация, содержащаяся в этом комментарии, — вполне достаточно номера RFC.

```
/*
 RFC 2045 - Multipurpose Internet Mail Extensions (MIME)
 Часть 1: Формат тел сообщений
 раздел 6.8.  Кодирование данных Base64
 В процессе кодирования 24-разрядные группы входных битов представляются
 в виде выходных строк из 4 закодированных символов. Слева направо 24-разрядная
 входная группа образуется посредством конкатенации 38-разрядных входных групп.
 Далее эти 24 бита интерпретируются как 4 конкатенированных 6-разрядных группы,
 каждая из которых преобразуется в одну цифру алфавита base64. При кодировании
 потока битов в кодировке base64 предполагается, что битовый поток упорядочивается
 от старшего значащего бита. Иначе говоря, первым битом потока будет старший бит
 первого 8-битового байта, а восьмым - младший бит первого 8-битого байта и т. д.
 */
```

Неочевидные комментарии

Связь между комментарием и кодом, который он описывает, должна быть очевидной. Если уж вы берете на себя хлопоты, связанные с написанием комментария, то по крайней мере читатель должен посмотреть на комментарий и на код и понять, о чем говорится в комментарии. Для примера возьмем следующий комментарий из общих модулей Apache:

```
/*
 * Начать с массива, размер которого достаточен для хранения
 * всех пикселов (плюс байты фильтра), плюс еще 200 байт
 * для данных заголовка
 */
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

Что такое «байты фильтра»? Они как-то связаны с +1? Или с *3? И с тем и с другим? Один пиксел соответствует одному байту? И почему 200? Цель комментария — объяснить код, который не объясняет сам себя. Плохо, когда сам комментарий нуждается в объяснениях.

Заголовки функций

Короткие функции не нуждаются в долгих описаниях. Хорошо выбранное имя компактной функции, которая выполняет одну операцию, обычно лучше заголовка с комментарием.

Заголовки Javadoc во внутреннем коде

При всей полезности комментариев Javadoc для API общего пользования не применяйте их в коде, не предназначенном для общего потребления. Генерирование страниц Javadoc для внутренних классов и функций системы обычно не приносит реальной пользы, а формализм комментариев Javadoc только отвлекает читателя.

Пример

Модуль в листинге 4.7 был написан для первого учебного курса «XP Immersion». Предполагалось, что он является примером плохого кодирования и стиля комментирования. Кент Бек переработал этот код в куда более приятную форму перед несколькими десятками увлеченных слушателей. Позднее я приспособил этот пример для своей книги «Agile Software Development, Principles, Patterns, and Practices» и статьи в журнале «Software Development». Любопытно, что в то время многие из нас считали этот модуль «хорошо документированным». Теперь мы видим, что он представляет собой ералаш. Посмотрим, сколько разных ошибок комментирования вам удастся найти.

Листинг 4.7. GeneratePrimes.java

```
/**
 * Класс генерирует простые числа в диапазоне до максимального значения,
 *  заданного пользователем, по алгоритму "Решета Эратосфена".
 * <p>
 * Эратосфен Киренский, 276 год до н.э., Ливия --  * 194 год до н.э., Александрия.
 * Первый ученый, вычисливший длину земного меридиана. Известен своими работами
 * о календарях с високосным годом, заведовал Александрийской библиотекой.
 * <p>
 * Алгоритм весьма прост. Берем массив целых чисел, начиная с 2, и вычеркиваем
 * из него все числа, кратные 2. Находим следующее невычеркнутое число
 * и вычеркиваем все его кратные. Повторяем до тех пор, пока не дойдем
 * до квадратного корня верхней границы диапазона.
 *
 * @author Альфонс
 * @version 13 февраля 2002 и
 */
import java.util.*;

public class GeneratePrimes
{
  /**
   * @param maxValue - верхняя граница диапазона.
   */
  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue >= 2) // Единственно допустимый случай
```

```
    {
      // Объявления
      int s = maxValue + 1; // Размер массива
      boolean[] f = new boolean[s];
      int i;

      // Инициализировать массив значениями true.
      for (i = 0; i < s; i++)
        f[i] = true;
      // Удалить числа, заведомо не являющиеся простыми.
      f[0] = f[1] = false;

      // Отсев
      int j;
      for (i = 2; i < Math.sqrt(s) + 1; i++)
      {
        if (f[i]) // Если элемент i не вычеркнут, вычеркнуть кратные ему.
        {
          for (j = 2 * i; j < s; j += i)
            f[j] = false; // Кратные числа не являются простыми.
        }
      }

      // Сколько простых чисел осталось?
      int count = 0;
      for (i = 0; i < s; i++)
      {
        if (f[i])
          count++; // Приращение счетчика
      }

      int[] primes = new int[count];

      // Переместить простые числа в результат
      for (i = 0, j = 0; i < s; i++)
      {
        if (f[i])             // Если простое
          primes[j++] = i;
      }

      return primes;  // Вернуть простые числа
    }
    else // maxValue < 2
      return new int[0]; // Вернуть пустой массив при недопустимых входных данных.
  }
}
```

В листинге 4.8 приведена переработанная версия того же модуля. Обратите внимание: применение комментариев стало намного более ограниченным. Во всем модуле осталось всего два комментария пояснительного характера.

Листинг 4.8. PrimeGenerator.java (переработанная версия)

```
/**
 * Класс генерирует простые числа до максимального значения, заданного
 * пользователем, по алгоритму "Решета Эратосфена".
 * Берем массив целых чисел, начиная с 2, и вычеркиваем
 * из него все числа, кратные 2. Находим следующее невычеркнутое число
 * и вычеркиваем все числа, кратные ему. Повторяем до тех пор, пока из массива
 * не будут вычеркнуты все кратные.
 */

public class PrimeGenerator
{
  private static boolean[] crossedOut;
  private static int[] result;

  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      uncrossIntegersUpTo(maxValue);
      crossOutMultiples();
      putUncrossedIntegersIntoResult();
      return result;
    }
  }

  private static void uncrossIntegersUpTo(int maxValue)
  {
    crossedOut = new boolean[maxValue + 1];
    for (int i = 2; i < crossedOut.length; i++)
      crossedOut[i] = false;
  }

  private static void crossOutMultiples()
  {
    int limit = determineIterationLimit();
    for (int i = 2; i <= limit; i++)
      if (notCrossed(i))
        crossOutMultiplesOf(i);
  }

  private static int determineIterationLimit()
  {
    // Каждое кратное в массиве имеет простой множитель, больший либо равный
    // квадратному корню из размера массива. Следовательно, вычеркивать элементы,
    // кратные числам, превышающих квадратный корень, не нужно.
    double iterationLimit = Math.sqrt(crossedOut.length);
    return (int) iterationLimit;
  }
  private static void crossOutMultiplesOf(int i)
  {
```

```
          multiple < crossedOut.length;
          multiple += i)
        crossedOut[multiple] = true;
    }

    private static boolean notCrossed(int i)
    {
      return crossedOut[i] == false;
    }

    private static void putUncrossedIntegersIntoResult()
    {
      result = new int[numberOfUncrossedIntegers()];
      for (int j = 0, i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
          result[j++] = i;
    }

    private static int numberOfUncrossedIntegers()
    {
      int count = 0;
      for (int i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
          count++;

      return count;
    }
}
```

Можно возразить, что первый комментарий избыточен, потому что он практически полностью повторяет код самой функции `generatePrimes`. И все же я считаю, что этот комментарий упрощает понимание алгоритма пользователем, поэтому я склонен оставить его.

Второй комментарий почти стопроцентно необходим. Он объясняет смысл использования квадратного корня как верхней границы цикла. Мне не удалось найти ни простого имени переменной, ни другой структуры кода, которые бы наглядно передавали это обстоятельство. С другой стороны, само использование квадратного корня может быть иллюзией. Действительно ли ограничение цикла квадратным корнем способно сэкономить время? Не уйдет ли на его вычисление больше времени, чем я экономлю? Об этом стоит подумать. Использование квадратного корня в качестве верхней границы цикла тешит мои наклонности старого хакера, работавшего на C и ассемблере, но я не уверен, что оно оправдает время и усилия, необходимые читателям кода для его понимания.

Литература

[KP78]: Kernighan and Plaugher, The Elements of Programming Style, 2d. ed., McGraw-Hill, 1978.

5 Форматирование

Мы хотим, чтобы читатель, заглянувший «под капот» программы, был поражен увиденным — нашей аккуратностью, логичностью и вниманием к мелочам. Мы хотим, чтобы на него произвела впечатление стройность кода. Мы хотим, чтобы он уважительно поднял брови при просмотре модулей. Мы хотим, чтобы наша работа выглядела профессионально. Если вместо этого читатель видит беспорядочную массу кода, словно написанного шайкой пьяных матросов, то он заключит, что такое же неуважение к мелочам проникло и во все остальные аспекты проекта.

Вы должны позаботиться о том, чтобы ваш код был хорошо отформатирован. Выберите набор простых правил, определяющих формат кода, и последовательно

применяйте их в своей работе. Если вы работаете в составе группы, то группа должна выработать согласованный набор правил форматирования, соблюдаемых всеми участниками. Также полезно иметь средства автоматизации, которые применяют правила форматирования за вас.

Цель форматирования

Прежде всего я твердо заявляю: форматирование кода важно. Оно слишком важно, чтобы не обращать на него внимания, и слишком важно, чтобы относиться к нему с религиозным пылом. Форматирование кода направлено на передачу информации, а передача информации является первоочередной задачей профессионального разработчика.

Возможно, вы думали, что первоочередная задача профессионального разработчика – «сделать так, чтобы программа заработала». Надеюсь, к этому моменту книга уже заставила вас отказаться от этих представлений. Функциональность, созданная сегодня, вполне может измениться в следующей версии, но удобочитаемость вашего кода окажет сильное воздействие на все изменения, которые когда-либо будут внесены. Стиль кодирования и удобочитаемость создают прецеденты, которые продолжают влиять на сопровождаемость и расширяемость кода уже после того, как исходный код изменился до неузнаваемости. Стиль и дисциплина программирования продолжают жить, даже если ваш код остался в прошлом. Так какие же аспекты форматирования помогают нам лучше передать свои мысли?

Вертикальное форматирование

Начнем с вертикальных размеров. Насколько большим должен быть исходный файл? В Java размер файла тесно связан с размером класса. Мы поговорим о размерах классов, когда речь пойдет о классах, а пока давайте займемся размером файлов.

Насколько большими должны быть исходные файлы Java? Оказывается, существует широчайший диапазон размеров и весьма заметные различия в стиле. Некоторые из этих различий показаны на рис. 5.1.

На рисунке изображены семь разных проектов: Junit, FitNesse, TestNG, Time and Money (Tam), JDepend, Ant и Tomcat. Отрезки, проходящие через прямоугольники, показывают минимальную и максимальную длину файла в каждом проекте. Прямоугольник изображает приблизительно одну треть (стандартное отклонение[1]) от диапазона длин файлов. Середина прямоугольника соответствует

[1] Прямоугольник представляет диапазон «сигма/2» выше и ниже среднего значения. Да, я знаю, что распределение длин файлов не является нормальным, поэтому стандартное отклонение не может считаться математически точным. Но я и не стремлюсь к точности. Я хочу лишь дать представление о происходящем.

среднему арифметическому. Таким образом, средний размер файла в проекте FitNesse составляет около 65 строк, а около трети файлов имеет размер от 40 до 100+ строк. Наибольший файл FitNesse занимает около 400 строк, а наименьший — всего 6 строк. Обратите внимание: на графике используется логарифмическая шкала, поэтому незначительные изменения в вертикальной координате подразумевают очень большие изменения в абсолютном размере.

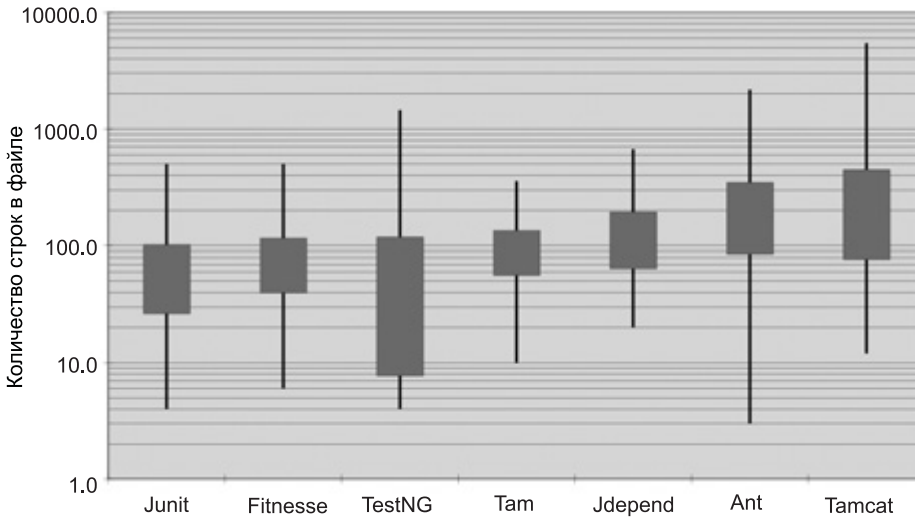


Рис. 5.1. Распределение длин файлов по логарифмической шкале (высота прямоугольника = сигма)

Junit, FitNesse и Time and Money состоят из относительно небольших файлов. Ни один размер файла не превышает 500 строк, а большинство файлов не превышает 200 строк. Напротив, в Tomcat и Ant встречаются файлы из нескольких тысяч строк, а около половины имеет длину более 200 строк.

Что это означает для нас? То, что достаточно серьезную систему (объем FitNesse приближается к 50 000 строк) можно построить из файлов, типичная длина которых составляет 200 строк, с верхним пределом в 500 строк. Хотя это не должно считаться раз и навсегда установленным правилом, такие показатели весьма желательны. Маленькие файлы обычно более понятны, чем большие.

Газетная метафора

Представьте себе хорошо написанную газетную статью. Естественно, статья читается по вертикали. В самом начале обычно располагается заголовок с общей темой статьи; он помогает вам решить, представляет ли статья интерес для вас. В первом абзаце приводится краткое изложение сюжета на уровне общих концепций, без приведения каких-либо подробностей. По мере продвижения к кон-

цу статьи объем детализации непрерывно растет, пока вы не узнаете все даты, имена, цитаты и т. д.

Исходный файл должен выглядеть как газетная статья. Имя файла должно быть простым, но содержательным. Одного имени должно быть достаточно для того, чтобы читатель понял, открыл ли он нужный модуль или нет. Начальные блоки исходного файла описывают высокоуровневые концепции и алгоритмы. Степень детализации увеличивается при перемещении к концу файла, а в самом конце собираются все функции и подробности низшего уровня в исходном файле.

Газета состоит из множества статей, в большинстве своем очень коротких. Другие статьи чуть длиннее. И лишь немногие статьи занимают всю газетную страницу. Собственно, именно этим газеты так удобны. Если бы они состояли из одной длинной статьи с неупорядоченной подборкой фактов, дат и имен, то мы бы просто не смогли их читать.

Вертикальное разделение концепций

Практически весь код читается слева направо и сверху вниз. Каждая строка представляет выражение или условие, а каждая группа строк представляет законченную мысль. Эти мысли следует отделять друг от друга пустыми строками.

Для примера возьмем листинг 5.1. Объявление пакета, директива(-ы) импорта и все функции разделяются пустыми строками. Это чрезвычайно простое правило оказывает глубокое воздействие на визуальную структуру кода. Каждая пустая строка становится зрительной подсказкой, указывающей на начало новой самостоятельной концепции. В ходе просмотра листинга ваш взгляд привлекает первая строка, следующая за пустой строкой.

Листинг 5.1. BoldWidget.java

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );

  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }
```

продолжение ⇩

Листинг 5.1 (продолжение)

```
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Удаление пустых строк, как в листинге 5.2, имеет весьма тяжелые последствия для удобочитаемости кода.

Листинг 5.2. BoldWidget.java

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));}
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Эффект становится еще более заметным, если на секунду отвести глаза от листинга. В первом примере группировка строк сразу бросается в глаза, а второй пример выглядит как сплошная каша, притом что два листинга различаются только вертикальными разделителями.

Вертикальное сжатие

Если вертикальные пропуски разделяют концепции, то вертикальное сжатие подчеркивает тесные связи. Таким образом, строки кода, между которыми существует тесная связь, должны быть «сжаты» по вертикали. Обратите внимание на то, как бесполезные комментарии в листинге 5.3 нарушают группировку двух переменных экземпляров.

Листинг 5.3

```
public class ReporterConfig {

  /**
   * Имя класса слушателя
   */
```

```
  private String m_className;
  /**
   * Свойства слушателя
   */
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Листинг 5.4 читается гораздо проще. Он нормально воспринимается «с одного взгляда» — по крайней мере, для меня. Я смотрю на него и сразу вижу, что передо мной класс с двумя переменными и одним методом; для этого мне не приходится вертеть головой или бегать по строчкам глазами. В предыдущем листинге для достижения того же уровня понимания приходится потрудиться намного больше.

Листинг 5.4

```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Вертикальные расстояния

Вам когда-нибудь доводилось метаться по классу, прыгая от одной функции к другой, прокручивая исходный файл вверх-вниз, пытаясь разобраться, как функции связаны друг с другом и как они работают, — только для того, чтобы окончательно заблудиться в его запутанных нагромождениях? Когда-нибудь искали определение функции или переменной по цепочкам наследования? Все это крайне неприятно, потому что вы стараетесь понять, как работает система, а вместо этого вам приходится тратить время и интеллектуальные усилия на поиски и запоминание местонахождения отдельных фрагментов.

Концепции, тесно связанные друг с другом, должны находиться поблизости друг от друга по вертикали [G10]. Разумеется, это правило не работает для концепций, находящихся в разных файлах. Но тесно связанные концепции и не должны находиться в разных файлах, если только это не объясняется очень вескими доводами. Кстати, это одна из причин, по которой следует избегать защищенных переменных.

Если концепции связаны настолько тесно, что они находятся в одном исходном файле, их вертикальное разделение должно показывать, насколько они важны для понимания друг друга. Не заставляйте читателя прыгать туда-сюда по исходным файлам и классам.

Объявления переменных. Переменные следует объявлять как можно ближе к месту использования. Так как мы имеем дело с очень короткими функциями, локальные переменные должны перечисляться в начале каждой функции, как в следующем примере из Junit 4.3.

```
private static void readPreferences() {
  InputStream is= null;
  try {
    is= new FileInputStream(getPreferencesFile());
    setPreferences(new Properties(getPreferences()));
    getPreferences().load(is);
  } catch (IOException e) {
    try {
      if (is != null)
        is.close();
      } catch (IOException e1) {
    }
  }
}
```

Управляющие переменные циклов обычно объявляются внутри конструкции цикла, как в следующей симпатичной маленькой функции из того же источника.

```
public int countTestCases() {
  int count= 0;
  for (Test each : tests)
    count += each.countTestCases();
  return count;
}
```

В отдельных случаях переменная может объявляться в начале блока или непосредственно перед циклом в длинной функции. Пример такого объявления представлен в следующем фрагменте очень длинной функции из TestNG.

```
...
for (XmlTest test : m_suite.getTests()) {
      TestRunner tr = m_runnerFactory.newTestRunner(this, test);
      tr.addListener(m_textReporter);
      m_testRunners.add(tr);

      invoker = tr.getInvoker();

      for (ITestNGMethod m : tr.getBeforeSuiteMethods()) {
        beforeSuiteMethods.put(m.getMethod(), m);
      }

      for (ITestNGMethod m : tr.getAfterSuiteMethods()) {
        afterSuiteMethods.put(m.getMethod(), m);
      }
    }
...
```

Переменные экземпляров, напротив, должны объявляться в начале класса. Это не увеличивает вертикальное расстояние между переменными, потому что

в хорошо спроектированном классе они используются многими, если не всеми, методами класса.

Размещение переменных экземпляров становилось причиной ожесточенных споров. В C++ обычно применялось так называемое *правило ножниц*, при котором все переменные экземпляров размещаются внизу. С другой стороны, в Java они обычно размещаются в начале класса. Я не вижу причин для использования каких-либо других конвенций. Здесь важно то, что переменные экземпляров должны объявляться в одном хорошо известном месте. Все должны знать, где следует искать объявления.

Для примера рассмотрим странный класс TestSuite из JUnit 4.3.1. Я основательно сократил этот класс, чтобы лучше выразить свою мысль. Где-то в середине листинга вдруг обнаруживаются объявления двух переменных экземпляров. Если бы автор сознательно хотел спрятать их, трудно найти более подходящее место. Читатель кода может наткнуться на эти объявления только случайно (как я).

```
public class TestSuite implements Test {
  static public Test createTest(Class<? extends TestCase> theClass,
                                String name) {
    ...
  }

  public static Constructor<? extends TestCase>
    getTestConstructor(Class<? extends TestCase> theClass)
    throws NoSuchMethodException {
  ...
  }

  public static Test warning(final String message) {
  ...
  }

  private static String exceptionToString(Throwable t) {
  ...
  }

  private String fName;

  private Vector<Test> fTests= new Vector<Test>(10);

  public TestSuite() {
  }

  public TestSuite(final Class<? extends TestCase> theClass) {
   ...
  }

  public TestSuite(Class<? extends TestCase>  theClass, String name) {
  ...
  }
  ... ... ... ... ...
}
```

Зависимые функции. Если одна функция вызывает другую, то эти функции должны располагаться вблизи друг от друга по вертикали, а вызывающая функция должна находиться над вызываемой (если это возможно). Тем самым формируется естественная структура программного кода. Если это правило будет последовательно соблюдаться, читатели кода будут уверены в том, что определения функций следуют неподалеку от их вызовов. Для примера возьмем фрагмент FitNesse из листинга 5.5. Обратите внимание на то, как верхняя функция вызывает нижние, и как они, в свою очередь, вызывают функции более низкого уровня. Такая структура позволяет легко найти вызываемые функции и значительно улучшает удобочитаемость всего модуля.

Листинг 5.5. WikiPageResponder.java

```
public class WikiPageResponder implements SecureResponder {
  protected WikiPage page;
  protected PageData pageData;
  protected String pageTitle;
  protected Request request;
  protected PageCrawler crawler;

  public Response makeResponse(FitNesseContext context, Request request)
    throws Exception {
    String pageName = getPageNameOrDefault(request, "FrontPage");
    loadPage(pageName, context);
    if (page == null)
      return notFoundResponse(context, request);
    else
      return makePageResponse(context);
  }

  private String getPageNameOrDefault(Request request, String defaultPageName)
  {
    String pageName = request.getResource();
    if (StringUtil.isBlank(pageName))
      pageName = defaultPageName;
    return pageName;
  }

  protected void loadPage(String resource, FitNesseContext context)
    throws Exception {
    WikiPagePath path = PathParser.parse(resource);
    crawler = context.root.getPageCrawler();
    crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
    page = crawler.getPage(context.root, path);
    if (page != null)
      pageData = page.getData();
  }

  private Response notFoundResponse(FitNesseContext context, Request request)
    throws Exception {
```

```
    return new NotFoundResponder().makeResponse(context, request);
  }

  private SimpleResponse makePageResponse(FitNesseContext context)
    throws Exception {
    pageTitle = PathParser.render(crawler.getFullPath(page));
    String html = makeHtml(context);

    SimpleResponse response = new SimpleResponse();
    response.setMaxAge(0);
    response.setContent(html);
    return response;
  }
...
```

Заодно этот фрагмент дает хороший пример хранения констант на соответствующем уровне [G35]. Константу «FrontPage» можно было бы объявить в функции `getPageNameOrDefault`, но тогда хорошо известная и ожидаемая константа оказалась бы погребенной в функции неуместно низкого уровня. Лучше переместить эту константу вниз – от того места, где ее следовало бы ввести, к месту ее фактического использования.

Концептуальное родство. Некоторые фрагменты кода *требуют*, чтобы их разместили вблизи от других фрагментов. Такие фрагменты обладают определенным концептуальным родством. Чем сильнее родство, тем меньше должно быть вертикальное расстояние между ними.

Как мы уже видели, родство может быть основано на прямой зависимости (когда одна функция вызывает другую) или на использовании переменных в функциях. Однако существуют и другие разновидности родства. Например, родство возникает в том случае, если группа функций выполняет аналогичные операции. Возьмем следующий фрагмент кода из Junit 4.3.1:

```
public class Assert {
  static public void assertTrue(String message,
boolean condition) {
    if (!condition)
      fail(message);
    }

    static public void assertTrue(boolean condition) {
      assertTrue(null, condition);
    }

    static public void assertFalse(String message, boolean condition) {
      assertTrue(message, !condition);
    }
```

```
    static public void assertFalse(boolean condition) {
      assertFalse(null, condition);
    }
...
```

Эти функции обладают сильным концептуальным родством, потому что они используют единую схему выбора имен и выполняют разные варианты одной базовой операции. Тот факт, что они вызывают друг друга, вторичен. Даже без него эти функции все равно следовало бы разместить поблизости друг от друга.

Вертикальное упорядочение

Как правило, взаимозависимые функции должны размещаться в нисходящем порядке. Иначе говоря, вызываемая функция должна располагаться ниже вызывающей функции. Так формируется логичная структура модуля исходного кода – от высокого уровня к более низкому.

Как и в газетных статьях, читатель ожидает, что самые важные концепции будут изложены сначала, причем с минимальным количеством второстепенных деталей. Низкоуровневые подробности естественно приводить в последнюю очередь. Это позволяет нам бегло просматривать исходные файлы, извлекая суть из нескольких начальных функций, без погружения в подробности. Листинг 5.5 имеет именно такую структуру. Возможно, еще лучшие примеры встречаются в листинге 15.5 на с. 299 и в листинге 3.7 на с. 75.

Горизонтальное форматирование

Насколько широкой должна быть строка? Чтобы ответить на этот вопрос, мы проанализируем ширину строк в типичных программах. Как и в предыдущем случае, будут проанализированы семь разных проектов. На рис. 5.2 показано распределение длин строк во всех семи проектах. Закономерность впечатляет, особенно около 45 символов. Фактически каждый размер от 20 до 60 соответствует примерно одному проценту от общего количества строк. Целых 40 процентов! Возможно, еще 30 процентов составляют строки с длиной менее 10 символов. Помните, что на графике используется логарифмическая шкала, поэтому разброс в области свыше 80 символов очень важен. Программисты явно предпочитают более короткие строки.

Это наводит на мысль, что строки лучше делать по возможности короткими. Установленное Холлеритом старое ограничение в 80 символов выглядит излишне жестким; я ничего не имеют против строк длиной в 100 и даже 120 символов. Но более длинные строки, вероятно, вызваны небрежностью программиста.

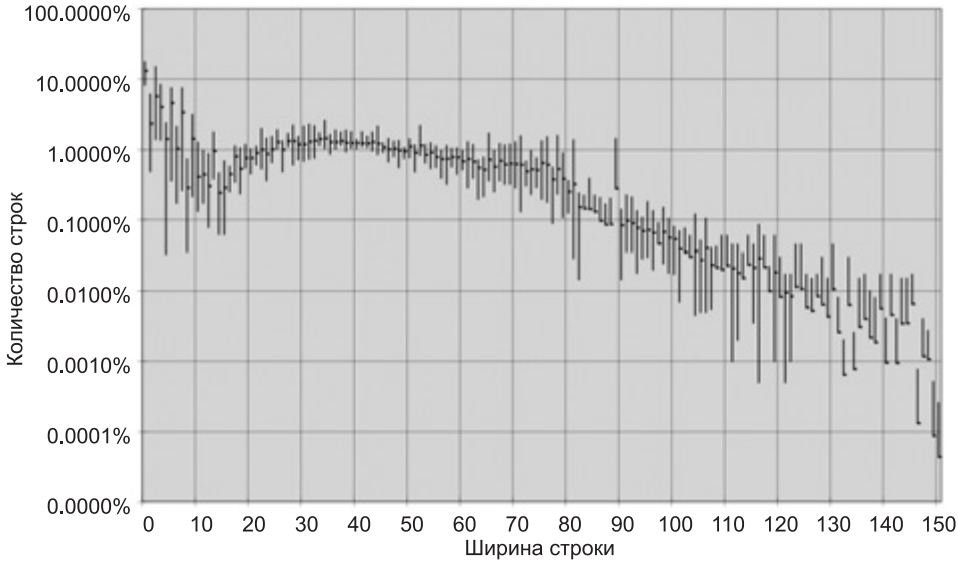


Рис. 5.2. Распределение ширины строк в Java

Прежде я использовал это правило, чтобы мне не приходилось прокручивать программный код вправо. Но современные мониторы стали настолько широкими, а молодые программисты выбирают настолько мелкие шрифты, что на экране помещается до 200 символов. Не делайте этого. Лично я установил себе «верхнюю планку» в 120 символов.

Горизонтальное разделение и сжатие

Горизонтальные пропуски используются для группировки взаимосвязанных элементов и разделения разнородных элементов. Рассмотрим следующую функцию:

```
private void measureLine(String line) {
  lineCount++;
  int lineSize = line.length();
  totalChars += lineSize;
  lineWidthHistogram.addLine(lineSize, lineCount);
  recordWidestLine(lineSize);
}
```

Знаки присваивания окружены пробелами, обеспечивающими их визуальное выделение. Операторы присваивания состоят из двух основных элементов: левой и правой частей. Пробелы наглядно подчеркивают это разделение.

С другой стороны, я не стал отделять имена функций от открывающих скобок. Это обусловлено тем, что имя функции тесно связано с ее аргументами. Пробелы изолируют их вместо того, чтобы объединять. Я также разделил аргументы

в скобках пробелами, чтобы выделить запятые и подчеркнуть, что аргументы не зависят друг от друга.

Пробелы также применяются для визуального обозначения приоритета операторов:

```
public class Quadratic {
  public static double root1(double a, double b, double c) {
    double determinant = determinant(a, b, c);
    return (-b + Math.sqrt(determinant)) / (2*a);
  }

  public static double root2(int a, int b, int c) {
    double determinant = determinant(a, b, c);
    return (-b - Math.sqrt(determinant)) / (2*a);
  }

  private static double determinant(double a, double b, double c) {
    return b*b - 4*a*c;
  }
}
```

Обратите внимание, как хорошо читаются формулы. Между множителями нет пробелов, потому что они обладают высоким приоритетом. Слагаемые разделяются пробелами, так как сложение и вычитание имеют более низкий приоритет.

К сожалению, в большинстве программ форматирования кода приоритет операторов не учитывается, и во всех случаях применяются одинаковые пропуски. Нетривиальные изменения расстояний, как в приведенном примере, теряются после переформатирования кода.

Горизонтальное выравнивание

Когда я был ассемблерным программистом[1], горизонтальное выравнивание использовалось для визуального выделения некоторых структур. Когда я перешел на C, C++, а в конце концов и на Java, я продолжал выравнивать все имена переменных в группах объявлений или все правосторонние значения в группах команд присваивания. Мой код выглядел примерно так:

```
public class FitNesseExpediter implements ResponseSender
{
  private   Socket          socket;
  private   InputStream     input;
  private   OutputStream    output;
  private   Request         request;
  private   Response        response;
  private   FitNesseContext context;
  protected long            requestParsingTimeLimit;
  private   long            requestProgress;
```

[1] Кого я пытаюсь обмануть? Я так и остался ассемблерным программистом. Парня можно разлучить с «металлом», но в душе «металл» все равно живет!

```
  private   long            requestParsingDeadline;
  private   boolean         hasError;

  public FitNesseExpediter(Socket s,
                           FitNesseContext context) throws Exception
  {
    this.context =            context;
    socket =                  s;
    input =                   s.getInputStream();
    output =                  s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

Однако потом я обнаружил, что такое выравнивание не приносит пользы. Оно визуально выделяет совсем не то, что требуется, и отвлекает читателя от моих истинных намерений. Например, в приведенном выше списке объявлений читатель просматривает имена переменных, не обращая внимания на их типы. Аналогичным образом в списке команд присваивания возникает соблазн просмотреть правосторонние значения, не замечая оператора присваивания. Ситуация усугубляется тем, что средства автоматического форматирования обычно удаляют подобное выравнивание.

Поэтому в итоге я отказался от этого стиля форматирования. Сейчас я отдаю предпочтение невыровненным объявлениям и присваиваниям, как в следующем фрагменте, потому что они помогают выявить один важный дефект. Если в программе встречаются длинные списки, нуждающиеся в выравнивании, то проблема кроется в длине списка, а не в отсутствии выравнивания. Длина списков объявлений в классе FitNesseExpediter наводит на мысль, что этот класс необходимо разделить.

```
public class FitNesseExpediter implements ResponseSender
{
  private Socket socket;
  private InputStream input;
  private OutputStream output;
  private Request request;
  private Response response;
  private FitNesseContext context;
  protected long requestParsingTimeLimit;
  private long requestProgress;
  private long requestParsingDeadline;
  private boolean hasError;

  public FitNesseExpediter(Socket s, FitNesseContext context) throws Exception
  {
    this.context = context;
    socket = s;
    input = s.getInputStream();
    output = s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

Отступы

Исходный файл имеет иерархическую структуру. В нем присутствует информация, относящаяся к файлу в целом; к отдельным классам в файле; к методам внутри классов; к блокам внутри методов и рекурсивно – к блокам внутри блоков. Каждый уровень этой иерархии образует область видимости, в которой могут объявляться имена и в которой интерпретируются исполняемые команды.

Чтобы создать наглядное представление этой иерархии, мы снабжаем строки исходного кода отступами, размер которых соответствует их позиции в иерархии. Команды уровня файла (такие, как большинство объявлений классов) отступов не имеют. Методы в классах сдвигаются на один уровень вправо от уровня класса. Реализации этих методов сдвигаются на один уровень вправо от объявления класса. Реализации блоков сдвигаются на один уровень вправо от своих внишних блоков и т. д.

Программисты широко используют эту схему расстановки отступов в своей работе. Чтобы определить, к какой области видимости принадлежат строки кода, они визуально группируют строки по левому краю. Это позволяет им быстро пропускать области видимости, не относящиеся к текущей ситуации (например, реализации команд `if` и `while`). У левого края ищутся объявления новых методов, новые переменные и даже новые классы. Без отступов программа становится практически нечитаемой для людей. Следующие программы идентичны с синтаксической и семантической точки зрения:

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }
```

```
public class FitNesseServer implements SocketServer {
  private FitNesseContext context;
  public FitNesseServer(FitNesseContext context) {
    this.context = context;
  }

  public void serve(Socket s) {
    serve(s, 10000);
  }

  public void serve(Socket s, long requestTimeout) {
    try {
      FitNesseExpediter sender = new FitNesseExpediter(s, context);
      sender.setRequestParsingTimeLimit(requestTimeout);
      sender.start();
```

```
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Наше зрение быстро охватывает структуру файла с отступами. Мы почти мгновенно находим переменные, конструкторы и методы. Всего за несколько секунд можно понять, что класс предоставляет простой интерфейс для работы с сокетом, с поддержкой тайм-аута. С другой стороны, разобраться в версии без отступов без тщательного анализа практически невозможно.

Нарушения отступов. Иногда возникает соблазн нарушить правила расстановки отступов в коротких командах `if`, коротких циклах `while` или коротких функциях. Но каждый раз, когда я поддавался этому искушению, я почти всегда возвращался и расставлял отступы, как положено. Таким образом, я стараюсь не сворачивать блоки в одну строку, как в этом фрагменте:

```
public class CommentWidget extends TextWidget
{
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

  public CommentWidget(ParentWidget parent, String text){super(parent, text);}
  public String render() throws Exception {return ""; }
}
```

Вместо этого я предпочитаю развернутые блоки с правильными отступами:

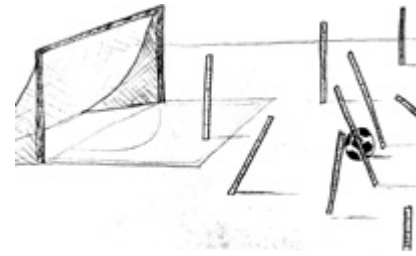
```
public class CommentWidget extends TextWidget {
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

  public CommentWidget(ParentWidget parent, String text) {
    super(parent, text);
  }

  public String render() throws Exception {
    return "";
  }
}
```

Вырожденные области видимости

Иногда тело цикла `while` или команды `for` не содержит команд, то есть является вырожденным, как в следующем фрагменте. Я не люблю такие структуры и стараюсь избегать их. А когда это невозможно, я по крайней мере слежу за тем, чтобы пустое тело имело правильные отступы и было заключено в фигурные скобки. Вы не представляете, как часто меня обманывала точка с запятой, молчаливо прячущаяся в конце цикла `while` в той же строке. Если не сделать эту точку хорошо заметной, разместив ее в отдельной строке, ее попросту слишком сложно разглядеть:

```
while (dis.read(buf, 0, readBufferSize) != -1)
  ;
```

Правила форматирования в группах



У каждого программиста есть свои любимые правила форматирования, но если он работает в группе, то должен руководствоваться групповыми правилами.

Группа разработчиков согласует единый стиль форматирования, который в дальнейшем применяется всеми участниками. Код программного продукта должен быть оформлен в едином стиле. Он не должен выглядеть так, словно был написан несколькими личностями, расходящимися во мнениях по поводу оформления.

В начале работы над проектом FitNesse в 2002 году я провел встречу с группой для выработки общего стиля программирования. На это потребовалось около 10 минут. Мы решили, где будем расставлять фигурные скобки, каким будет размер отступов, по какой схеме будут присваиваться имена классов, переменных и методов и т. д. Затем эти правила были закодированы в системе форматирования кода нашей рабочей среды, и в дальнейшем мы неуклонно придерживались их. Это были не те правила, которые предпочитаю лично я; это были правила, выбранные группой. И я, как участник группы, неуклонно соблюдал их при написании кода в проекте FitNesse.

Хорошая программная система состоит из набора удобочитаемых документов, оформленных в едином, согласованном стиле. Читатель должен быть уверен в том, что форматные атрибуты, встречающиеся в одном исходном файле, будут иметь точно такой же смысл в других файлах. Ни в коем случае не усложняйте исходный код, допуская его оформление в нескольких разных стилях.

Правила форматирования от дядюшки Боба

Правила, которые использую лично я, очень просты; они представлены в коде листинга 5.6. Перед вами пример того, как сам код становится лучшим документом, описывающим стандарты кодирования.

Листинг 5.6. CodeAnalyzer.java

```
public class CodeAnalyzer implements JavaFileAnalysis {
  private int lineCount;
  private int maxLineWidth;
  private int widestLineNumber;
  private LineWidthHistogram lineWidthHistogram;
  private int totalChars;

  public CodeAnalyzer() {
    lineWidthHistogram = new LineWidthHistogram();
  }
```

```
public static List<File> findJavaFiles(File parentDirectory) {
  List<File> files = new ArrayList<File>();
  findJavaFiles(parentDirectory, files);
  return files;
}

private static void findJavaFiles(File parentDirectory, List<File> files) {
  for (File file : parentDirectory.listFiles()) {
    if (file.getName().endsWith(".java"))
      files.add(file);
    else if (file.isDirectory())
      findJavaFiles(file, files);
  }
}

public void analyzeFile(File javaFile) throws Exception {
  BufferedReader br = new BufferedReader(new FileReader(javaFile));
  String line;
  while ((line = br.readLine()) != null)
    measureLine(line);
}

private void measureLine(String line) {
  lineCount++;
  int lineSize = line.length();
  totalChars += lineSize;
  lineWidthHistogram.addLine(lineSize, lineCount);
  recordWidestLine(lineSize);
}

private void recordWidestLine(int lineSize) {
  if (lineSize > maxLineWidth) {
    maxLineWidth = lineSize;
    widestLineNumber = lineCount;
  }
}

public int getLineCount() {
  return lineCount;
}

public int getMaxLineWidth() {
  return maxLineWidth;
}

public int getWidestLineNumber() {
  return widestLineNumber;
}

public LineWidthHistogram getLineWidthHistogram() {
  return lineWidthHistogram;
}
```

продолжение ⇩

Листинг 5.6 (продолжение)

```
  public double getMeanLineWidth() {
    return (double)totalChars/lineCount;
  }

  public int getMedianLineWidth() {
    Integer[] sortedWidths = getSortedWidths();
    int cumulativeLineCount = 0;
    for (int width : sortedWidths) {
      cumulativeLineCount += lineCountForWidth(width);
      if (cumulativeLineCount > lineCount/2)
        return width;
    }
    throw new Error("Cannot get here");
  }

  private int lineCountForWidth(int width) {
    return lineWidthHistogram.getLinesforWidth(width).size();
  }

  private Integer[] getSortedWidths() {
    Set<Integer> widths = lineWidthHistogram.getWidths();
    Integer[] sortedWidths = (widths.toArray(new Integer[0]));
    Arrays.sort(sortedWidths);
    return sortedWidths;
  }
}
```

6 Объекты и структуры данных

Существует веская причина для ограничения доступа к переменным в программах: мы не хотим, чтобы другие программисты зависели от них. Мы хотим иметь возможность свободно менять тип или реализацию этих переменных так, как считаем нужным. Тогда почему же многие программисты автоматически включают в свои объекты методы чтения/записи, предоставляя доступ к приватным переменным так, словно они являются открытыми?

Абстракция данных

Давайте сравним между собой листинги 6.1 и 6.2. В обоих случаях код представляет точку на декартовой плоскости. Однако в одном случае реализация открыта, а в другом она полностью скрыта от внешнего пользователя.

Листинг 6.1. Конкретная реализация Point

```
public class Point {
  public double x;
  public double y;
}
```

Листинг 6.2. Абстрактная реализация Point

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

Элегантность решения из листинга 6.2 заключается в том, что внешний пользователь не знает, какие координаты использованы в реализации — прямоугольные или полярные. А может, еще какие-нибудь! Тем не менее интерфейс безусловно напоминает структуру данных.

Однако он представляет нечто большее, чем обычную структуру данных. Его методы устанавливают политику доступа к данным. Пользователь может читать значения координат независимо друг от друга, но присваивание координат должно выполняться одновременно, в режиме атомарной операции.

С другой стороны, листинг 6.1 явно реализован в прямоугольных координатах, а пользователь вынужден работать с этими координатами независимо. Более того, такое решение раскрывает реализацию даже в том случае, если бы переменные были объявлены приватными, и мы использовали одиночные методы чтения/записи.

Скрытие реализации не сводится к созданию прослойки функций между переменными. Скрытие реализации направлено на формирование абстракций! Класс не просто ограничивает доступ к переменным через методы чтения/записи. Вместо этого он предоставляет абстрактные интерфейсы, посредством которых пользователь оперирует с *сущностью* данных. Знать, как эти данные реализованы, ему при этом не обязательно.

Возьмем листинги 6.3 и 6.4. В первом случае для получения информации о запасе топлива используются конкретные физические показатели, а во втором — абстрактные проценты. В первом, конкретном случае можно быть уверенным в том, что методы представляют собой обычные методы доступа к переменным. Во втором, абстрактном случае пользователь не имеет ни малейшего представления о фактическом формате данных.

Листинг 6.3. Конкретная реализация Vehicle

```
public interface Vehicle {
  double getFuelTankCapacityInGallons();
  double getGallonsOfGasoline();
}
```

Листинг 6.4. Абстрактная реализация Vehicle

```
Abstract Vehicle
public interface Vehicle {
  double getPercentFuelRemaining();
}
```

В обоих примерах вторая реализация является предпочтительной. Мы не хотим раскрывать подробности строения данных. Вместо этого желательно использовать представление данных на абстрактном уровне. Задача не решается простым использованием интерфейсов и/или методов чтения/записи. Чтобы найти лучший способ представления данных, содержащихся в объекте, необходимо серьезно поразмыслить. Бездумное добавление методов чтения и записи — худший из всех возможных вариантов.

Антисимметрия данных/объектов

Два предыдущих примера показывают, чем объекты отличаются от структур данных. Объекты скрывают свои данные за абстракциями и предоставляют функции, работающие с этими данными. Структуры данных раскрывают свои данные и не имеют осмысленных функций. А теперь еще раз перечитайте эти определения. Обратите внимание на то, как они дополняют друг друга, фактически являясь противоположностями. Различия могут показаться тривиальными, но они приводят к далеко идущим последствиям.

Возьмем процедурный пример из листинга 6.5. Класс `Geometry` работает с тремя классами геометрических фигур. Классы фигур представляют собой простые структуры данных, лишенные какого-либо поведения. Все поведение сосредоточено в классе `Geometry`.

Листинг 6.5. Процедурные фигуры

```
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}
```

продолжение ⇩

Листинг 6.5 (продолжение)

```
public class Geometry {
  public final double PI = 3.141592653589793;
  public double area(Object shape) throws NoSuchShapeException
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

Объектно-ориентированный программист недовольно поморщится и пожалуется на процедурную природу реализации — и будет прав. Но возможно, его презрительная усмешка не обоснована. Подумайте, что произойдет при включении в `Geometry` функции `perimeter()`. Классы фигур остаются неизменными! И все остальные классы, зависящие от них, тоже остаются неизменными! С другой стороны, при добавлении новой разновидности фигур мне придется изменять все функции `Geometry`, чтобы они могли работать с ней. Перечитайте еще раз. Обратите внимание на то, что эти два условия диаметрально противоположны.

Теперь рассмотрим объектно-ориентированное решение из листинга 6.6. Метод `area()` является полиморфным, класс `Geometry` становится лишним. Добавление новой фигуры не затрагивает ни одну из существующих *функций*, но при добавлении новой функции приходится изменять все *фигуры*![1]

Листинг 6.6. Полиморфные фигуры

```
Polymorphic Shapes
public class Square implements Shape {
  private Point topLeft;
  private double side;

  public double area() {
    return side*side;
  }
}
```

[1] У проблемы существуют обходные решения, хорошо известные опытным объектно-ориентированным программистам: например, паттерн ПОСЕТИТЕЛЬ или двойная диспетчеризация. Но у этих приемов имеются собственные издержки, к тому же они обычно возвращают структуру к состоянию процедурной программы.

```
public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}

public class Circle implements Shape {
  private Point center;
  private double radius;
  public final double PI = 3.141592653589793;

  public double area() {
    return PI * radius * radius;
  }
}
```

И снова мы наблюдаем взаимодополняющую природу этих двух определений. В этом проявляется основополагающая дихотомия между объектами и структурами данных.

Процедурный код (код, использующий структуры данных) позволяет легко добавлять новые функции без изменения существующих структур данных. Объектно-ориентированный код, напротив, упрощает добавление новых классов без изменения существующих функций.

Обратные утверждения также истинны.

Процедурный код усложняет добавление новых структур данных, потому что оно требует изменения всех функций. Объектно-ориентированный код усложняет добавление новых функций, потому что для этого должны измениться все классы.

Таким образом, то, что сложно в ОО, просто в процедурном программировании, а то, что сложно в процедурном программировании, просто в ОО!

В любой сложной системе возникают ситуации, когда вместо новых функций в систему требуется включить новые типы данных. Для таких ситуаций объекты и объектно-ориентированное программирование особенно уместны. Впрочем, бывает и обратное — вместо новых типов данных требуется добавить новые функции. Тогда лучше подходит процедурный код и структуры данных.

Опытные программисты хорошо знают: представление о том, что все данные должны представляться в виде объектов — *миф*. Иногда предпочтительны простые структуры данных и процедуры, работающие с ними.

Закон Деметры

Хорошо известное эвристическое правило, называемое *законом Деметры*[1], гласит, что модуль не должен знать внутреннее устройство тех объектов, с которыми он работает. Как мы видели в предыдущем разделе, объекты скрывают свои данные и предоставляют операции для работы с ними. Это означает, что объект не должен раскрывать свою внутреннюю структуру через методы доступа, потому что внутреннюю структуру следует скрывать.

В более точной формулировке закон Деметры гласит, что метод *f* класса *C* должен ограничиваться вызовом методов следующих объектов:

- *C*;
- объекты, созданные *f*;
- объекты, переданные *f* в качестве аргумента;
- объекты, хранящиеся в переменной экземпляра *C*.

Метод *не должен* вызывать методы объектов, возвращаемых любыми из разрешенных функций. Другими словами, разговаривать можно с друзьями, но не с чужаками.

Следующий код нарушает закон Деметры (среди прочего), потому что он вызывает функцию `getScratchDir()` для возвращаемого значения `getOptions()`, а затем вызывает `getAbsolutePath()` для возвращаемого значения `getScratchDir()`.

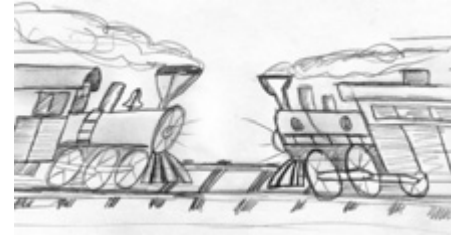
```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Крушение поезда

Подобная структура кода часто называется «крушением поезда», потому что цепочки вызовов напоминают сцепленные вагоны поезда. Такие конструкции считаются проявлением небрежного стиля программирования и их следует избегать [G36]. Обычно цепочки лучше разделить в следующем виде:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

Нарушают ли эти два фрагмента закон Деметры? Несомненно, вмещающий модуль знает, что объект контекста `ctxt` содержит значения параметров, в число которых входит и временный каталог, обладающий абсолютным путем. Это довольно большой объем информации



[1] http://en.wikipedia.org/wiki/Law_of_Demeter

для одной функции. Вызывающая функция должна знать, как перемещаться между множеством разных объектов.

Нарушает ли этот код закон Деметры или нет? Все зависит от того, чем являются `ctxt`, `Options` и `ScratchDir` — объектами или структурами данных. Если это объекты, то их внутренняя структура должна скрываться, поэтому необходимость информации об их строении является явным нарушением закона Деметры. С другой стороны, если `ctxt`, `Options` и `ScratchDir` представляют собой обычные структуры данных, не обладающие поведением, то они естественным образом раскрывают свою внутреннюю структуру, а закон Деметры на них не распространяется.

Применение функций доступа затрудняет ситуацию. Если бы код был записан следующим образом, вероятно, у нас не возникало бы вопросов по поводу нарушения закона Деметры:

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

Ситуация существенно упростилась бы, если бы структуры данных просто содержали открытые переменные без функций, а объекты — приватные переменные с открытыми функциями. Однако некоторые существующие инфраструктуры и стандарты (например, Beans) требуют, чтобы даже простые структуры данных имели методы чтения и записи.

Гибриды

Вся эта неразбериха иногда приводит к появлению гибридных структур — наполовину объектов, наполовину структур данных. Гибриды содержат как функции для выполнения важных операций, так и открытые переменные или открытые методы чтения/записи, которые во всех отношениях делают приватные переменные открытыми. Другим внешним функциям предлагается использовать эти переменные так, как в процедурных программах используются структуры данных[1].

Подобные гибриды усложняют как добавление новых функций, так и новых структур данных. Они объединяют все худшее из обеих категорий. Не используйте гибриды. Они являются признаком сумбурного проектирования, авторы которого не уверены (или еще хуже, не знают), что они собираются защищать: функции или типы.

Скрытие структуры

А если `ctxt`, `options` и `scratchDir` представляют собой объекты с реальным поведением? Поскольку объекты должны скрывать свою внутреннюю структуру, мы не сможем перемещаться между ними. Как же в этом случае узнать абсолютный путь временного каталога?

[1] Иногда это называется «функциональной завистью» (Feature Envy) — из [Refactoring].

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

или

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

Первый вариант приведет к разрастанию набора методов объекта `ctxt`. Второй вариант предполагает, что `getScratchDirectoryOption()` возвращает структуру данных, а не объект. Ни один из вариантов не вызывает энтузиазма.

Если `ctxt` является объектом, то мы должны приказать ему выполнить некую операцию, а не запрашивать у него информацию о его внутреннем устройстве. Зачем нам понадобился абсолютный путь к временному каталогу? Что мы собираемся с ним делать? Рассмотрим следующий фрагмент того же модуля (расположенный на много строк ниже):

```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

Смешение разных уровней детализации [G34][G6] выглядит немного пугающе. Точки, косые черты, расширения файлов и объекты `File` не должны так беспечно перемешиваться между собой и с окружающим кодом. Но если не обращать на это внимания, мы видим, что абсолютный путь временного каталога определялся для создания временного файла с заданным именем.

Так почему бы не приказать объекту `ctxt` выполнить эту операцию?

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

Выглядит вполне разумно! Такое решение позволяет объекту `ctxt` скрыть свое внутреннее строение, а текущей функции не приходится нарушать закон Деметры, перемещаясь между объектами, о которых ей знать не положено.

Объекты передачи данных

Квинтэссенцией структуры данных является класс с открытыми переменными и без функций. Иногда такие структуры называются *объектами передачи данных*, или DTO (Data Transfer Object). Структуры DTO чрезвычайно полезны, особенно при работе с базами данных, разборе сообщений из сокетов и т. д. С них часто начинается серия фаз преобразования низкоуровневых данных, полученных из базы, в объекты кода приложения.

Несколько большее распространение получила форма bean-компонентов, представленная в листинге 6.7. Bean-компоненты состоят из приватных переменных, операции с которыми осуществляются при помощи методов чтения/записи. Подобная форма псевдоинкапсуляции поднимает настроение некоторым блюстителям чистоты ОО, но обычно не имеет других преимуществ.

Листинг 6.7. address.java

```
public class Address {
  private String street;
  private String streetExtra;
```

```
  private String city;
  private String state;
  private String zip;

  public Address(String street, String streetExtra,
                 String city, String state, String zip) {
    this.street = street;
    this.streetExtra = streetExtra;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  public String getStreet() {
    return street;
  }

  public String getStreetExtra() {
    return streetExtra;
  }

  public String getCity() {
    return city;
  }

  public String getState() {
    return state;
  }

  public String getZip() {
    return zip;
  }
}
```

Активные записи

Активные записи (Active Records) составляют особую разновидность DTO. Они тоже представляют собой структуры данных с открытыми переменными (или переменными с bean-доступом), но обычно в них присутствуют навигационные методы — такие, как `save` или `find`. Активные записи чаще всего являются результатами прямого преобразования таблиц баз данных или других источников данных.

К сожалению, разработчики часто пытаются интерпретировать такие структуры данных, как объекты, и включают в них методы, реализующие бизнес-логику. Однако такой подход нежелателен, так как он создает гибрид между структурой данных и объектом.

Конечно, проблема решается иначе: активные записи интерпретируются как структуры данных, а в программе создаются отдельные объекты, которые содержат бизнес-логику и скрывают свои внутренние данные (которые, возможно, представляют собой обычные экземпляры класса активной записи).

Заключение

Объекты предоставляют поведение и скрывают данные. Это позволяет программисту легко добавлять новые виды объектов, не изменяя существующего поведения. С другой стороны, объекты усложняют добавление нового поведения к существующим объектам. Структуры данных предоставляют данные, но не обладают сколько-нибудь значительным поведением. Они упрощают добавление нового поведения в существующие структуры данных, но затрудняют добавление новых структур данных в существующие функции.

Если в некоторой системе нас прежде всего интересует гибкость в добавлении новых типов данных, то в этой части системы предпочтение отдается объектной реализации. В других случаях нам нужна гибкость расширения поведения, и тогда в этой части используются типы данных и процедуры. Хороший программист относится к этой проблеме без предубеждения и выбирает то решение, которое лучше всего подходит для конкретной ситуации.

Литература

[Refactoring]: Refactoring: Improving the Design of Existing Code, Martin Fowler et al., Addison-Wesley, 1999.

7 Обработка ошибок

Майкл Физерс

На первый взгляд глава, посвященная обработке ошибок, в книге о чистом коде выглядит немного странно. Обработка ошибок — одна из тех рутинных вещей, которыми нам всем приходится заниматься при программировании. Программа может получить аномальные входные данные, на устройстве могут произойти сбои. Короче говоря, выполнение программы может пойти по неверному пути, и если это случается, мы, программисты, должны позаботиться, чтобы наш код сделал то, что ему положено сделать.

Однако связь этих двух тем — обработки ошибок и чистого кода — очевидна. Во многих кодовых базах обработка ошибок выходит на первый план. Я вовсе не хочу сказать, что код не делает ничего полезного, кроме обработки ошибок; я имею в виду, что из-за разбросанной повсюду обработки ошибок практически невозможно понять, что же делает код. Обработка ошибок важна, но если они заслоняют собой логику программы — значит, она реализована неверно.

В этой главе представлены некоторые соображения и приемы, которые помогают писать чистый и надежный код, то есть код, в котором ошибки обрабатываются стильно и элегантно.

Используйте исключения вместо кодов ошибок

В далеком прошлом многие языки программирования не поддерживали механизма обработки исключений. В таких языках возможности обработки и получения информации об ошибках были ограничены. Программа либо устанавливала флаг ошибки, либо возвращала код, который проверялся вызывающей стороной. Оба способа продемонстрированы в листинге 7.1.

Листинг 7.1. DeviceController.java

```
public class DeviceController {
  ...
  public void sendShutDown() {
    DeviceHandle handle = getHandle(DEV1);
    // Проверить состояние устройства
    if (handle != DeviceHandle.INVALID) {
      // Сохранить состояние устройства в поле записи
      retrieveDeviceRecord(handle);
      // Если устройство не приостановлено, отключить его
      if (record.getStatus() != DEVICE_SUSPENDED) {
        pauseDevice(handle);
        clearDeviceWorkQueue(handle);
        closeDevice(handle);
      } else {
        logger.log("Device suspended.  Unable to shut down");
      }
    } else {
      logger.log("Invalid handle for: " + DEV1.toString());
    }
  }
  ...
}
```

У обоих решений имеется общий недостаток: они загромождают код на стороне вызова. Вызывающая сторона должна проверять ошибки немедленно после вызова. К сожалению, об этом легко забыть. По этой причине при обнаружении ошибки лучше инициировать исключение. Код вызова становится более понятным, а его логика не скрывается за кодом обработки ошибок.

В листинге 7.2 представлен тот же код с выдачей исключений в методах, способных обнаруживать ошибки.

Обратите внимание, насколько чище стал код. Причем дело даже не в эстетике. Качество кода возросло, потому что два аспекта, которые прежде были тесно переплетены — алгоритм отключения устройства и обработка ошибок, — теперь изолированы друг от друга. Вы можете рассмотреть их по отдельности и разобраться в каждом из них независимо.

Листинг 7.2. DeviceController.java (с исключениями)

```
public class DeviceController {
  ...

  public void sendShutDown() {
    try {
      tryToShutDown();
    } catch (DeviceShutDownError e) {
      logger.log(e);
    }
  }

  private void tryToShutDown() throws DeviceShutDownError {
    DeviceHandle handle = getHandle(DEV1);
    DeviceRecord record = retrieveDeviceRecord(handle);

    pauseDevice(handle);
    clearDeviceWorkQueue(handle);
    closeDevice(handle);
  }

  private DeviceHandle getHandle(DeviceID id) {
    ...
    throw new DeviceShutDownError("Invalid handle for: " + id.toString());
    ...
  }

  ...
}
```

Начните с написания команды try-catch-finally

У исключений есть одна интересная особенность: они определяют область видимости в вашей программе. Размещая код в секции `try` команды `try-catch-finally`, вы утверждаете, что выполнение программы может прерваться в любой точке, а затем продолжиться в секции `catch`.

Блоки `try` в каком-то отношении напоминают транзакции. Секция `catch` должна оставить программу в целостном состоянии, что бы и произошло в секции `try`. По этой причине написание кода, который может инициировать исключения, рекомендуется начинать с конструкции `try-catch-finally`. Это поможет вам определить, чего должен ожидать пользователь кода, что бы ни произошло в коде `try`.

Допустим, требуется написать код, который открывает файл и читает из него сериализованные объекты.

Начнем с модульного теста, который проверяет, что при неудачном обращении к файлу будет выдано исключение:

```
@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {
  sectionStore.retrieveSection("invalid - file");
}
```

Для теста необходимо создать следующую программную заглушку:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  // Пусто, пока не появится реальная реализация
  return new ArrayList<RecordedGrip>();
}
```

Тест завершается неудачей, потому что код не инициирует исключения. Затем мы изменяем свою реализацию так, чтобы она попыталась обратиться к несуществующему файлу. При попытке выполнения происходит исключение:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  try {
    FileInputStream stream = new FileInputStream(sectionName)
  } catch (Exception e) {
    throw new StorageException("retrieval error", e);
  }
  return new ArrayList<RecordedGrip>();
}
```

Теперь тест проходит успешно, потому что мы перехватили исключение. На этой стадии можно переработать код. Тип перехватываемого исключения сужается до типа, реально инициируемого конструктором `FileInputStream`, то есть `FileNotFoundException`:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  try {
    FileInputStream stream = new FileInputStream(sectionName);
    stream.close();
  } catch (FileNotFoundException e) {
    throw new StorageException("retrieval error", e);
  }
  return new ArrayList<RecordedGrip>();
}
```

Определив область видимости при помощи структуры `try-catch`, мы можем использовать методологию TDD для построения остальной необходимой логики. Эта логика размещается между созданием `FileInputStream` и закрытием, а в ее коде можно считать, что все операции были выполнены без ошибок.

Попробуйте написать тесты, принудительно инициирующие исключения, а затем включите в обработчик поведение, обеспечивающее прохождение тестов. Это заставит вас построить транзакционную область видимости блока `try` и поможет сохранить ее транзакционную природу.

Используйте непроверяемые исключения

Время споров прошло. Java-программисты годами обсуждали преимущества и недостатки проверяемых исключений (checked exceptions). Когда проверяемые исключения появились в первой версии Java, всем казалось, что это отличная идея. В сигнатуре каждого метода должны быть перечислены все исключения, которые могут передаваться вызывающей стороне. Фактически исключения становились частью типа метода. Если сигнатура не соответствовала тому, что происходит в коде, то программа просто не компилировалась.

В то время мы с энтузиазмом относились к проверяемым исключениям; в самом деле, они бывают полезными. Но сейчас стало ясно, что они не являются необходимыми для создания надежных программ. В C# нет проверяемых исключений, и несмотря на все доблестные попытки, в C++ они так и не появились. Их также нет в Python и Ruby. Тем не менее на всех этих языках можно писать надежные программы. А раз так, нам приходится решать, оправдывают ли проверяемые исключения ту цену, которую за них приходится платить.

Какую цену, спросите вы? Цена проверяемых исключений — нарушение принципа открытости/закрытости [Martin]. Если вы инициируете проверяемое исключение из метода своего кода, а `catch` находится тремя уровнями выше, то это исключение должно быть объявлено в сигнатурах всех методов между вашим методом и `catch`. Следовательно, изменение на низком уровне программного продукта приводит к изменениям сигнатур на многих более высоких уровнях. Измененные модули приходится строить и развертывать заново, притом что в программе не изменилось ничего, что было бы существенно для них.

Представьте иерархию вызовов большой системы. Функции верхнего уровня вызывают функции нижележащего уровня, которые, в свою очередь, вызывают функции низких уровней и т. д. Теперь допустим, что одна из низкоуровневых функций изменилась таким образом, что она должна инициировать исключение. Если это исключение является проверяемым, то в сигнатуру функции должна быть добавлена секция `throws`. Но тогда каждая функция, вызывающая нашу измененную функцию, тоже должна быть изменена с перехватом нового исключения или присоединением соответствующей секции `throws` к ее сигнатуре. И так до бесконечности. В итоге мы имеем каскад изменений, пробивающихся с нижних уровней программного продукта на верхние уровни! При этом нарушается инкапсуляция, потому что все функции на пути инициирования должны располагать подробной информацией об этом низкоуровневом исключении. Учитывая, что главной целью исключений является возможность обработки ошибок «на расстоянии», такое нарушение инкапсуляции проверяемыми исключениями выглядит особенно постыдно.

Проверяемые исключения иногда могут пригодиться при написании особо важных библиотек: программист обязан перехватить их. Но в общем случае разработки приложений проблемы, создаваемые зависимостями, перевешивают преимущества.

Передавайте контекст с исключениями

Каждое исключение, инициируемое в программе, должно содержать достаточно контекстной информации для определения источника и местонахождения ошибки. В Java из любого исключения можно получить данные трассировки стека; однако по трассировке невозможно узнать, с какой целью выполнялась операция, завершившаяся неудачей.

Создавайте содержательные сообщения об ошибках и передавайте их со своими исключениями. Включайте в них сведения о сбойной операции и типе сбоя. Если в приложении ведется журнал, передайте информацию, достаточную для регистрации ошибки из секции `catch`.

Определяйте классы исключений в контексте потребностей вызывающей стороны

Существует много способов классификации ошибок. Например, ошибки можно классифицировать по источнику, то есть по компоненту, в котором они произошли. Также возможна классификация по типу: сбои устройств, сетевые сбои, ошибки программирования и т. д. Однако при определении классов исключений в приложениях думать необходимо прежде всего о том, *как они будут перехватываться*.

Рассмотрим пример неудачной классификации исключений. Далее приводится конструкция `try-catch-finally` для сторонней библиотечной функции. Она учитывает все исключения, которые могут быть инициированы при вызовах:

```
ACMEPort port = new ACMEPort(12);

try {
  port.open();
} catch (DeviceResponseException e) {
  reportPortError(e);
  logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
  reportPortError(e);
  logger.log("Unlock exception", e);
} catch (GMXError e) {
  reportPortError(e);
  logger.log("Device response exception");
} finally {
  …
}
```

Конструкция содержит множество повторений, и это неудивительно. В большинстве ситуаций при обработке исключений выполняются относительно стан-

дартные действия, не зависящие от их реальной причины. Мы должны сохранить ошибку и убедиться в том, что работа программы может быть продолжена. В этом случае, поскольку выполняемая работа остается более или менее постоянной независимо от исключения, код можно существенно упростить — для этого мы создаем «обертку» для вызываемой функции API и обеспечиваем возвращение стандартного типа исключения:

```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportError(e);
  logger.log(e.getMessage(), e);
} finally {
  …
}
```

Класс `LocalPort` представляет собой простую обертку, которая перехватывает и преобразует исключения, инициированные классом `ACMEPort`:

```
public class LocalPort {
  private ACMEPort innerPort;

  public LocalPort(int portNumber) {
    innerPort = new ACMEPort(portNumber);
  }

  public void open() {
    try {
      innerPort.open();
    } catch (DeviceResponseException e) {
      throw new PortDeviceFailure(e);
    } catch (ATM1212UnlockedException e) {
      throw new PortDeviceFailure(e);
    } catch (GMXError e) {
      throw new PortDeviceFailure(e);
    }
  }
  …
}
```

Обертки — вроде той, которую мы определили для `ACMEPort`, — бывают очень полезными. Более того, инкапсуляция вызовов сторонних API принадлежит к числу стандартных приемов. Создавая обертку для стороннего вызова, вы сокращаете до минимума зависимость от него в своем коде: в будущем вы можете переключиться на другую библиотеку без сколько-нибудь заметных проблем. Обертки также упрощают имитацию сторонних вызовов в ходе тестирования кода.

Последнее преимущество оберток заключается в том, что вы не ограничиваетесь архитектурными решениями разработчика API. Вы можете определить тот API, который вам удобен. В предыдущем примере мы определили для всех сбоев порта один тип исключения, и код от этого стал намного чище.

Часто в определенной области кода бывает достаточно одного класса исключения. Информация, передаваемая с исключением, позволяет различать разные виды ошибок. Используйте разные классы исключений только в том случае, если вы намерены перехватывать одни исключения, разрешая прохождение других типов.

Определите нормальный путь выполнения

Выполнение рекомендаций из предыдущих разделов обеспечивает хорошее разделение бизнес-логики и кода обработки ошибок. Основной код программы начинает выглядеть как простой алгоритм, не отягощенный посторонними вставками. Однако в результате код обнаружения ошибок смещается на периферию вашей программы. Вы создаете обертки для внешних API, чтобы иметь возможность инициировать собственные исключения, и определяете обработчик, который находится над основным кодом и позволяет справиться с любым прерыванием вычислений. Обычно такое решение отлично работает, но в некоторых ситуациях прерывание нежелательно.

Рассмотрим конкретный пример. В следующем, довольно неуклюжем фрагменте суммируются командировочные расходы на питание:

```
try {
  MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
  m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
  m_total += getMealPerDiem();
}
```

Если работник предъявил счет по затратам на питание, то сумма включается в общий итог. Если счет отсутствует, то работнику за этот день начисляется определенная сумма. Исключение загромождает логику программы. А если бы удалось обойтись без обработки особого случая? Это позволило бы заметно упростить код:

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

Можно ли упростить код до такой формы? Оказывается, можно. Мы можем изменить класс `ExpenseReportDAO`, чтобы он всегда возвращал объект `MealExpense`. При отсутствии предъявленного счета возвращается объект `MealExpense`, у которого в качестве затрат указана стандартная сумма, начисляемая за день:

```
public class PerDiemMealExpenses implements MealExpenses {
  public int getTotal() {
    // Вернуть стандартные ежедневные затраты на питание
  }
}
```

Такое решение представляет собой реализацию паттерна ОСОБЫЙ СЛУЧАЙ [Fowler]. Программист создает класс или настраивает объект так, чтобы он обрабатывал особый случай за него. Это позволяет избежать обработки исключительного поведения в клиентском коде. Все необходимое поведение инкапсулируется в объекте особого случая.

Не возвращайте null

На мой взгляд, при любых обсуждениях обработки ошибок необходимо упомянуть о неправильных действиях программистов, провоцирующих ошибки. На первом месте в этом списке стоит возвращение `null`. Я видел бесчисленное множество приложений, в которых едва ли не каждая строка начиналась с проверки `null`. Характерный пример:

```
public void registerItem(Item item) {
  if (item != null) {
    ItemRegistry registry = persistentStore.getItemRegistry();
    if (registry != null) {
      Item existing = registry.getItem(item.getID());
      if (existing.getBillingPeriod().hasRetailOwner()) {
        existing.register(item);
      }
    }
  }
}
```

Если ваша кодовая база содержит подобный код, возможно, вы не видите в нем ничего плохого, но это не так! Возвращая `null`, мы фактически создаем для себя лишнюю работу, а для вызывающей стороны — лишние проблемы. Стоит пропустить всего одну проверку `null`, и приложение «уходит в штопор».

А вы заметили, что во второй строке вложенной команды `if` проверка `null` отсутствует? Что произойдет во время выполнения, если значение `persistentStore` окажется равным `null`? Произойдет исключение `NullPointerException`; либо кто-то перехватит его на верхнем уровне, либо не перехватит. В обоих случаях все будет плохо. Как реагировать на исключение `NullPointerException`, возникшее где-то в глубинах вашего приложения?

Легко сказать, что проблемы в приведенном коде возникли из-за пропущенной проверки `null`. В действительности причина в другом: этих проверок *слишком много*. Если у вас возникает желание вернуть `null` из метода, рассмотрите возможность выдачи исключения или возвращения объекта «особого случая». Если ваш код вызывает метод стороннего API, способный вернуть `null`, создайте для него обертку в виде метода, который инициирует исключение или возвращает объект особого случая.

Довольно часто объекты особых случаев легко решают проблему. Допустим, у вас имеется код следующего вида:

```
List<Employee> employees = getEmployees();
if (employees != null) {
```

```
  for(Employee e : employees) {
    totalPay += e.getPay();
  }
}
```

Сейчас метод `getEmployees` может возвращать `null`, но так ли это необходимо? Если изменить `getEmployee` так, чтобы метод возвращал пустой список, код станет чище:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
  totalPay += e.getPay();
}
```

К счастью, в Java существует метод `Collections.emptyList()`, который возвращает заранее определенный неизменяемый список, и мы можем воспользоваться им для своих целей:

```
public List<Employee> getEmployees() {
  if( .. there are no employees .. )
    return Collections.emptyList();
}
```

Такое решение сводит к минимуму вероятность появления `NullPointerException`, а код становится намного чище.

Не передавайте null

Возвращать `null` из методов плохо, но передавать `null` при вызове еще хуже. По возможности избегайте передачи `null` в своем коде (исключение составляют разве что методы сторонних API, при вызове которых без нее не обойтись).

Следующий пример поясняет, почему не следует передавать `null`. Возьмем простой метод для вычисления метрики по двум точкам:

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    return (p2.x – p1.x) * 1.5;
  }
  …
}
```

Что произойдет, если при вызове будет передан аргумент `null`?

```
calculator.xProjection(null, new Point(12, 13));
```

Конечно, возникнет исключение `NullPointerException`.

Как исправить его? Можно создать новый тип исключения и инициировать его в методе:

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    if (p1 == null || p2 == null) {
```

```
      throw InvalidArgumentException(
        "Invalid argument for MetricsCalculator.xProjection");
    }
    return (p2.x – p1.x) * 1.5;
  }
}
```

Стало лучше? Пожалуй, лучше, чем `NullPointerException`, но вспомните: для `InvalidArgumentException` приходится определять обработчик. Что должен делать этот обработчик? Возьметесь предложить хорошую идею?

Существует другая альтернатива: можно воспользоваться набором проверочных директив `assert`:

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    assert p1 != null : "p1 should not be null";
    assert p2 != null : "p2 should not be null";
    return (p2.x – p1.x) * 1.5;
  }
}
```

Неплохо с точки зрения документирования, но проблема не решена. Если при вызове передать `null`, произойдет ошибка времени выполнения.

В большинстве языков программирования не существует хорошего способа справиться со случайной передачей `null` с вызывающей стороны. А раз так, разумно запретить передачу `null` по умолчанию. В этом случае вы будете знать, что присутствие `null` в списке аргументов свидетельствует о возникшей проблеме; это будет способствовать уменьшению количества ошибок, сделанных по неосторожности.

Заключение

Чистый код хорошо читается, но он также должен быть надежным. Эти цели не конфликтуют друг с другом. Чтобы написать надежный и чистый код, следует рассматривать обработку ошибок как отдельную задачу, решаемую независимо от основной логики программы. В зависимости от того, насколько нам это удастся, мы сможем прорабатывать ее реализацию независимо от основной логики программы, а это окажет существенное положительное влияние на удобство сопровождения нашего кода.

Литература

[Martin]: Agile Software Development: Principles, Patterns, and Practices, Robert C. Martin, Prentice Hall, 2002.

8 Границы

Джеймс Гренинг

Редко когда весь программный код наших систем находится под нашим полным контролем. Иногда нам приходится покупать пакеты сторонних разработчиков или использовать открытый код. В других случаях мы зависим от других групп нашей компании, производящих компоненты или подсистемы для нашего проекта. И этот внешний код мы должны каким-то образом четко интегрировать со своим кодом. В этой главе рассматриваются приемы и методы «сохранения чистоты» границ нашего программного кода.

Использование стороннего кода

Между поставщиком и пользователем интерфейса существует естественная напряженность. Поставщики сторонних пакетов и инфраструктур стремятся к универсальности, чтобы их продукты работали в разных средах и были обращены к широкой аудитории. С другой стороны, пользователи желают получить интерфейс, специализирующийся на их конкретных потребностях. Эта напряженность приводит к появлению проблем на границах наших систем.

Для примера возьмем класс `java.util.Map`. Как видно из рис. 8.1, `Map` имеет очень широкий интерфейс с многочисленными возможностями. Конечно, мощь и гибкость контейнера полезны, но они также создают некоторые неудобства. Допустим, наше приложение строит объект `Map` и передает его другим сторонам. При этом мы не хотим, чтобы получатели `Map` удаляли данные из полученного контейнера. Но в самом начале списка стоит метод `clear()`, и любой пользователь `Map` может стереть текущее содержимое контейнера. А может быть, наша архитектура подразумевает, что в контейнере должны храниться объекты только определенного типа, но `Map` не обладает надежными средствами ограничения типов сохраняемых объектов. Любой настойчивый пользователь сможет разместить в `Map` элементы любого типа.

- `clear() void - Map`
- `containsKey(Object key) boolean - Map`
- `containsValue(Object value) boolean - Map`
- `entrySet() Set - Map`
- `equals(Object o) boolean - Map`
- `get(Object key) Object - Map`
- `getClass() Class<? extends Object> - Object`
- `hashCode() int - Map`
- `isEmpty() boolean - Map`
- `keySet() Set - Map`
- `notify() void - Object`
- `notifyAll() void - Object`
- `put(Object key, Object value) Object - Map`
- `putAll(Map t) void - Map`
- `remove(Object key) Object - Map`
- `size() int - Map`
- `toString() String - Object`
- `values() Collection - Map`
- `wait() void - Object`
- `wait(long timeout) void - Object`
- `wait(long timeout, int nanos) void - Object`

Рис. 8.1. Методы Map

Если в приложении требуется контейнер `Map` с элементами `Sensor`, его можно создать следующим образом:

```
Map sensors = new HashMap();
```

Когда другой части кода понадобится обратиться к элементу, мы видим код следующего вида:

```
Sensor s = (Sensor)sensors.get(sensorId );
```

Причем видим его не только в этом месте, но снова и снова по всему коду. Клиент кода несет ответственность за получение Object из Map и его приведение к правильному типу. Такое решение работает, но «чистым» его не назовешь. Кроме того, этот код не излагает свою историю, как ему положено. Удобочитаемость кода можно было бы заметно улучшить при помощи шаблонов (параметризованных контейнеров):

```
Map<Sensor> sensors = new HashMap<Sensor>();
...
Sensor s = sensors.get(sensorId );
```

Но и такая реализация не решает проблемы: Map<Sensor> предоставляет намного больше возможностей, чем нам хотелось бы.

Свободная передача Map<Sensor> по системе означает, что в случае изменения интерфейса Map исправления придется вносить во множестве мест. Казалось бы, такие изменения маловероятны, но вспомните, что интерфейс изменился при добавлении поддержки шаблонов в Java 5. В самом деле, мы видели системы, разработчики которых воздерживались от использования шаблонов из-за большого количества потенциальных изменений, связанных с частым использованием Map.

Ниже представлен другой, более чистый вариант использования Map. С точки зрения пользователя Sensors совершенно не важно, используются шаблоны или нет. Это решение стало (и всегда должно быть) подробностью реализации.

```
public class Sensors {
  private Map sensors = new HashMap();

  public Sensor getById(String id) {
    return (Sensor) sensors.get(id);
  }

  //...
}
```

Граничный интерфейс (Map) скрыт от пользователя. Он может развиваться независимо, практически не оказывая никакого влияния на остальные части приложения. Применение шаблонов уже не создает проблем, потому что все преобразования типов выполняются в классе Sensors.

Этот интерфейс также приспособлен и ограничен в соответствии с потребностями приложения. Код становится более понятным, а возможности злоупотреблений со стороны пользователя сокращаются. Класс Sensors может обеспечивать выполнение архитектурных требований и требований бизнес-логики.

Поймите правильно: мы не предлагаем инкапсулировать каждое применение Map в этой форме. Скорее, мы рекомендуем ограничить передачу Map (или любого

другого граничного интерфейса) по системе. Если вы используете граничный интерфейс вроде Map, держите его внутри класса (или тесно связанного семейства классов), в которых он используется. Избегайте его возвращения или передачи в аргументах при вызовах методов общедоступных API.

Исследование и анализ границ

Сторонний код помогает нам реализовать больше функциональности за меньшее время. С чего начинать, если мы хотим использовать сторонний пакет? Тестирование чужого кода не входит в наши обязанности, но, возможно, написание тестов для стороннего кода, используемого в наших продуктах, в наших же интересах.

Допустим, вам не ясно, как использовать стороннюю библиотеку. Можно потратить день-два (или более) на чтение документации и принятие решений о том, как работать с библиотекой. Затем вы пишете код, использующий стороннюю библиотеку, и смотрите, делает ли он то, что ожидалось. Далее вы, скорее всего, погрязнете в долгих сеансах отладки, пытаясь разобраться, в чьем коде возникают ошибки – в стороннем или в вашем собственном.

Изучение чужого кода – непростая задача. Интеграция чужого кода тоже сложна. Одновременное решение обоих задач создает двойные сложности. А что, если пойти по другому пути? Вместо того чтобы экспериментировать и опробовать новую библиотеку в коде продукта, можно написать тесты, проверяющие наше понимание стороннего кода. Джим Ньюкирк (Jim Newkirk) называет такие тесты «учебными тестами» [BeckTDD, pp. 136–137].

В учебных тестах мы вызываем методы стороннего API в том виде, в котором намереваемся использовать их в своем приложении. Фактически выполняется контролируемый эксперимент, проверяющий наше понимание стороннего API. Основное внимание в тестах направлено на то, чего мы хотим добиться при помощи API.

Изучение log4j

Допустим, вместо того чтобы писать специализированный журнальный модуль, мы хотим использовать пакет apache log4j. Мы загружаем пакет и открываем страницу вводной документации. Не особенно вчитываясь в нее, мы пишем свой первый тестовый сценарий, который, как предполагается, будет выводить на консоль строку «hello».

```
@Test
public void testLogCreate() {
  Logger logger = Logger.getLogger("MyLogger");
  logger.info("hello");
}
```

При запуске журнальный модуль выдает ошибку. В описании ошибки говорится, что нам понадобится нечто под названием Appender. После непродолжительных поисков в документации обнаруживается класс ConsoleAppender. Соответственно, мы создаем объект ConsoleAppender и проверяем, удалось ли нам раскрыть секреты вывода журнала на консоль:

```
@Test
public void testLogAddAppender() {
  Logger logger = Logger.getLogger("MyLogger");
  ConsoleAppender appender = new ConsoleAppender();
  logger.addAppender(appender);
  logger.info("hello");
}
```

На этот раз выясняется, что у объекта Appender нет выходного потока. Странно – логика подсказывает, что он должен быть. После небольшой помощи от Google опробуется следующее решение:

```
@Test
public void testLogAddAppender() {
  Logger logger = Logger.getLogger("MyLogger");
  logger.removeAllAppenders();
  logger.addAppender(new ConsoleAppender(
    new PatternLayout("%p %t %m%n"),
    ConsoleAppender.SYSTEM_OUT));
  logger.info("hello");
}
```

Заработало; на консоли выводится сообщение со словом «hello»! На первый взгляд происходящее выглядит немного странно: мы должны указывать ConsoleAppender, что данные выводятся на консоль.

Еще интереснее, что при удалении аргумента ConsoleAppender.SystemOut сообщение «hello» все равно выводится. Но если убрать аргумент PatternLayout, снова начинаются жалобы на отсутствие выходного потока. Все это выглядит очень странно.

После более внимательного чтения документации мы видим, что конструктор ConsoleAppender по умолчанию «не имеет конфигурации» – весьма неочевидное и бесполезное решение. Похоже, это ошибка (или по крайней мере нелогичность) в log4j.

После некоторых поисков, чтения документации и тестирования мы приходим к листингу 8.1. Попутно мы получили много полезной информации о том, как работает log4j, и закодировали ее в наборе простых модульных тестов.

Листинг 8.1. LogTest.java

```
public class LogTest {
    private Logger logger;

    @Before
    public void initialize() {
        logger = Logger.getLogger("logger");
        logger.removeAllAppenders();
```

```
        Logger.getRootLogger().removeAllAppenders();
    }
    @Test
    public void basicLogger() {
        BasicConfigurator.configure();
        logger.info("basicLogger");
    }

    @Test
    public void addAppenderWithStream() {
      logger.addAppender(new ConsoleAppender(
          new PatternLayout("%p %t %m%n"),
          ConsoleAppender.SYSTEM_OUT));
      logger.info("addAppenderWithStream");
    }

    @Test
    public void addAppenderWithoutStream() {
      logger.addAppender(new ConsoleAppender(
          new PatternLayout("%p %t %m%n")));
      logger.info("addAppenderWithoutStream");
    }
}
```

Теперь мы знаем, как инициализировать простейший консольный вывод и можем воплотить эти знания в специализированном журнальном классе, чтобы изолировать остальной код приложения от граничного интерфейса `log4j`.

Учебные тесты: выгоднее, чем бесплатно

Учебные тесты не стоят ничего. API все равно приходится изучать, а написание тестов является простым способом получения необходимой информации, в изоляции от рабочего кода. Учебные тесты были точно поставленными экспериментами, которые помогли нам расширить границы своего понимания.

Учебные тесты не просто бесплатны – они приносят дополнительную прибыль. При выходе новых версий сторонних пакетов вы сможете провести учебные тесты и выяснить, не изменилось ли поведение пакета.

Учебные тесты позволяют убедиться в том, что сторонние пакеты, используемые в коде, работают именно так, как мы ожидаем. Нет никаких гарантий, что сторонний код, интегрированный в наши приложения, всегда будет сохранять совместимость. Например, авторы могут изменить код в соответствии с какими-то новыми потребностями. Изменения также могут происходить из-за исправления ошибок и добавления новых возможностей. Выход каждой новой версии сопряжен с новым риском. Если в стороннем пакете появятся изменения, несовместимые с нашими тестами, мы сразу узнаем об этом.

Впрочем, независимо от того, нужна ли вам учебная информация, получаемая в ходе тестирования, в системе должна существовать четкая граница, которая под-

держивается группой исходящих тестов, использующих интерфейс по аналогии с кодом продукта. Без *граничных тестов*, упрощающих процесс миграции, у нас появляются причины задержаться на старой версии дольше необходимого.

Использование несуществующего кода

Также существует еще одна разновидность границ, отделяющая известное от неизвестного. В коде часто встречаются места, в которых мы не располагаем полной информацией. Иногда то, что находится на другой стороне границы, остается неизвестным (по крайней мере в данный момент). Иногда мы намеренно не желаем заглядывать дальше границы.

Несколько лет назад я работал в группе, занимавшейся разработкой программного обеспечения для системы радиосвязи. В нашем продукте была подсистема «Передатчик», о которой мы почти ничего не знали, а люди, ответственные за разработку этой подсистемы, еще не дошли до определения своего интерфейса. Мы не хотели простаивать и поэтому начали работать подальше от неизвестной части кода.

Мы неплохо представляли себе, где заканчивалась наша зона ответственности и начиналась чужая территория. В ходе работы мы иногда наталкивались на границу. Хотя туманы и облака незнания скрывали пейзаж за границей, в ходе работы мы начали понимать, каким должен быть граничный интерфейс. Передатчику должны были отдаваться распоряжения следующего вида:

Настроить передатчик на заданную частоту и отправить аналоговое представление данных, поступающих из следующего потока.

Мы тогда понятия не имели, как это будет делаться, потому что API еще не был спроектирован. Поэтому подробности было решено отложить на будущее.

Чтобы не останавливать работу, мы определили собственный интерфейс с броским именем `Transmitter`. Интерфейс содержал метод `transmit`, которому при вызове передавались частота и поток данных. Это был тот интерфейс, который нам хотелось бы иметь.

У этого интерфейса было одно важное достоинство: он находился под нашим контролем. В результате клиентский код лучше читался, а мы в своей работе могли сосредоточиться на том, чего стремились добиться.

На рис. 8.2 мы видим, что классы `CommunicationsController` отделены от API передатчика (который находился вне нашего контроля и оставался неопределенным). Использование конкретного интерфейса нашего приложения позволило сохранить чистоту и выразительность кода `CommunicationsController`. После того как другая группа определила API передатчика, мы написали класс `TransmitterAdapter` для «наведения мостов». АДАПТЕР[1] инкапсулировал взаимодействие с API и создавал единое место для внесения изменений в случае развития API.

Такая архитектура также создает в коде очень удобный «стык» для тестирования. Используя подходящий `FakeTransmitter`, мы можем тестировать классы `Communi-`

[1] Gamma et al., ADAPTER, [GOF].

cationsController. Кроме того, сразу же после появления TransmitterAPI можно создать граничные тесты для проверки правильности использования API.

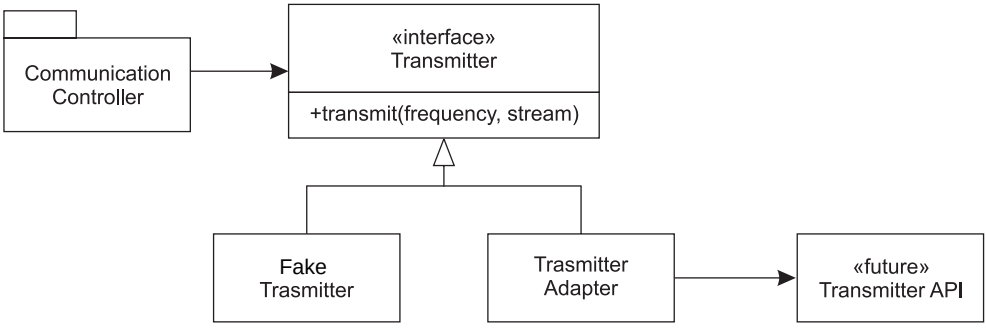


Рис. 8.2. Прогнозирование интерфейса передатчика

Чистые границы

На границах происходит много интересного. В частности, стоит уделить особое внимание изменениям. В хорошей программной архитектуре внесение изменений обходится без значительных затрат и усилий по переработке. Если в продукте используется код, находящийся вне нашего контроля, примите особые меры по защите капиталовложений и позаботьтесь о том, чтобы будущие изменения обходились не слишком дорого.

Для граничного кода необходимо четкое разделение сторон и тесты, определяющие ожидания пользователя. Постарайтесь, чтобы ваш код поменьше знал о специфических подробностях реализации стороннего кода. Лучше зависеть от того, что находится под вашим контролем, чем от тех факторов, которые вы не контролируете (а то, чего доброго, они начнут контролировать вас).

Чтобы границы со сторонним кодом не создавали проблем в наших проектах, мы сводим к минимуму количество обращений к ним. Для этого можно воспользоваться обертками, как в примере с Map, или реализовать паттерн АДАПТЕР для согласования нашего идеального интерфейса с реальным, полученным от разработчиков. В обоих вариантах код становится более выразительным, обеспечивается внутренняя согласованность обращений через границы, а изменение стороннего кода требует меньших затрат на сопровождение.

Литература

[BeckTDD]: Test Driven Development, Kent Beck, Addison-Wesley, 2003.

[GOF]: Design Patterns: Elements of Reusable Object Oriented Software, Gamma et al., Addison-Wesley, 1996.

[WELC]: Working Effectively with Legacy Code, Addison-Wesley, 2004.

9 Модульные тесты

За последние десять лет наша профессия прошла долгий путь. В 1997 году никто не слыхал о методологии TDD (Test Driven Development, то есть «разработка через тестирование»). Для подавляющего большинства разработчиков модульные тесты представляли собой короткие фрагменты временного кода, при помощи которого мы убеждались в том, что наши программы «работают». Мы тщательно выписывали свои классы и методы, а потом подмешивали специализированный код для их тестирования. Как правило, при этом использовалась какая-нибудь несложная управляющая программа, которая позволяла вручную взаимодействовать с тестируемым кодом.

Помню, в середине 90-х я написал программу на C++ для встроенной системы реального времени. Программа представляла собой простой таймер со следующей сигнатурой:

```
void Timer::ScheduleCommand(Command* theCommand, int milliseconds)
```

Идея была проста; метод `Execute` класса `Command` выполнялся в новом программном потоке с заданной задержкой в миллисекундах. Оставалось понять, как его тестировать.

Я соорудил простую управляющую программу, которая прослушивала события клавиатуры. Каждый раз, когда на клавиатуре вводился символ, программа планировала выполнение команды, повторяющей этот же символ пять секунд спустя. Затем я настучал на клавиатуре ритмичную мелодию и подождал, пока эта мелодия «появится» на экране спустя пять секунд.

«Мне... нужна такая девушка... как та... которую нашел мой старый добрый папа...»

Я напевал эту мелодию, нажимая клавишу «.», а потом пропел ее снова, когда точки начали появляться на экране.

И это был весь тест! Я убедился в том, что программа работает, показал ее своим коллегам и выкинул тестовый код.

Как я уже говорил, наша профессия прошла долгий путь. Сейчас я бы написал комплексный тест, проверяющий, что все углы и закоулки моего кода работают именно так, как положено. Я бы изолировал свой код от операционной системы, не полагаясь на стандартное выполнение по таймеру. Я бы самостоятельно реализовал хронометраж, чтобы тестирование проходило под моим полным контролем. Запланированные команды устанавливали бы логические флаги, а потом тестовый код выполнял бы мою программу в пошаговом режиме, наблюдая за состоянием флагов и их переходами из ложного состояния в истинное по прохождении нужного времени.

Когда у меня накопился бы пакет тестов, я бы позаботился о том, чтобы эти тесты были удобными для любого другого программиста, которому потребуется работать с моим кодом. Я бы проследил за тем, чтобы тесты и код поставлялись вместе, в одном исходном пакете.

Да, мы прошли долгий путь; но дорога еще не пройдена до конца. Движения гибких методологий и TDD поощряют многих программистов писать автоматизированные модульные тесты, а их ряды ежедневно пополняются новыми сторонниками. Однако в лихорадочном стремлении интегрировать тестирование в свою работу многие программисты упускают более тонкие и важные аспекты написания хороших тестов.

Три закона TTD

В наши дни каждому известно, что по требованиям методологии TDD модульные тесты должны писаться заранее, еще до написания кода продукта. Но это правило — всего лишь верхушка айсберга. Рассмотрим следующие три закона[1]:

[1] Professionalism and Test-Driven Development, Robert C. Martin, Object Mentor, IEEE Software, May/June 2007 (Vol. 24, No. 3), pp. 32–36; http://doi.ieeecomputersociety.org/10.1109/MS.2007.85

Первый закон. Не пишите код продукта, пока не напишете отказной модульный тест.

Второй закон. Не пишите модульный тест в объеме большем, чем необходимо для отказа. Невозможность компиляции является отказом.

Третий закон. Не пишите код продукта в объем большем, чем необходимо для прохождения текущего отказного теста.

Эти три закона устанавливают рамки рабочего цикла, длительность которого составляет, вероятно, около 30 секунд. Тесты и код продукта пишутся вместе, а тесты на несколько секунд опережают код продукта.

При такой организации работы мы пишем десятки тестов ежедневно, сотни тестов ежемесячно, тысячи тестов ежегодно. При такой организации работы тесты охватывают практически все аспекты кода продукта. Громадный объем тестов, сравнимый с объемом самого кода продукта, может создать немало организационных проблем.

О чистоте тестов

Несколько лет назад мне предложили заняться обучением группы, которая решила, что тестовый код не должен соответствовать тем же стандартам качества, что и код продукта. Участники группы сознательно разрешили друг другу нарушать правила в модульных тестах. «На скорую руку» — вот каким девизом они руководствовались. Разумно выбирать имена переменных не обязательно, короткие и содержательные тестовые функции не обязательны. Качественно проектировать тестовый код, организовать его продуманное логическое деление не обязательно. Тестовый код работает, охватывает код продукта — и этого вполне достаточно.

Пожалуй, некоторые читатели сочувственно отнесутся к этому решению. Возможно, кто-то в прошлом писал тесты наподобие тех, которые я написал для своего класса `Timer`. Примитивные «временные» тесты отделены огромным расстоянием от пакетов автоматизированного модульного тестирования. Многие программисты (как и та группа, в которой я преподавал) полагают, что тесты «на скорую руку» — лучше, чем полное отсутствие тестов.

Но на самом деле тесты «на скорую руку» равносильны полному отсутствию тестов, если не хуже. Дело в том, что тесты должны изменяться по мере развития кода продукта. Чем примитивнее тесты, тем труднее их изменять. Если тестовый код сильно запутан, то может оказаться, что написание нового кода продукта займет меньше времени, чем попытки втиснуть новые тесты в обновленный пакет. При изменении кода продукта старые тесты перестают проходить, а неразбериха в тестовом коде не позволяет быстро разобраться с возникшими проблемами. Таким образом, тесты начинают рассматриваться как постоянно растущий балласт.

От версии к версии затраты на сопровождение тестового пакета непрерывно росли. В конечном итоге тесты стали главной причиной для жалоб разработчи-

ков. Когда руководство спрашивало, почему работа занимает столько времени, разработчики винили во всем тесты. Кончилось все тем, что они полностью отказались от тестового пакета.

Однако без тестов программисты лишились возможности убедиться в том, что изменения в кодовой базе работают так, как ожидалось. Без тестов они уже не могли удостовериться в том, что изменения в одной части системы не нарушают работу других частей. Количество ошибок стало возрастать. А с ростом количества непредвиденных дефектов программисты начали опасаться изменений. Они перестали чистить код продукта, потому что боялись: не будет ли от изменений больше вреда, чем пользы? Код продукта стал загнивать. В итоге группа осталась без тестов, с запутанной и кишащей ошибками кодовой базой, с недовольными клиентами и с чувством, что все усилия по тестированию не принесли никакой пользы.

И в определенном смысле они были правы. Их усилия по тестированию *действительно* оказались бесполезными. Но виной всему было их решение — небрежно написанные тесты привели к катастрофе. Если бы группа ответственно подошла к написанию тестов, то затраченное время не пропало бы даром. Я говорю об этом вполне уверенно, потому что работал (и преподавал) во многих группах, добившихся успеха с аккуратно написанными модульными тестами.

Мораль проста: *тестовый код не менее важен, чем код продукта*. Не считайте его «кодом второго сорта». К написанию тестового кода следует относиться вдумчиво, внимательно и ответственно. Тестовый код должен быть таким же чистым, как и код продукта.

Тесты как средство обеспечения изменений

Если не поддерживать чистоту своих тестов, то вы их лишитесь. А без тестов утрачивается все то, что обеспечивает гибкость кода продукта. Да, вы не ошиблись. Именно модульные тесты обеспечивают гибкость, удобство сопровождения и возможность повторного использования нашего кода. Это объясняется просто: если у вас есть тесты, вы не боитесь вносить изменения в код! Без тестов любое изменение становится потенциальной ошибкой. Какой бы гибкой ни была ваша архитектура, каким бы качественным ни было логическое деление вашей архитектуры, без тестов вы будете сопротивляться изменениям из опасений, что они приведут к появлению скрытых ошибок.

С тестами эти опасения практически полностью исчезают. Чем шире охват тестирования, тем меньше вам приходится опасаться. Вы можете практически свободно вносить изменения даже в имеющий далеко не идеальную архитектуру, запутанный и малопонятный код. Таким образом, вы можете спокойно улучшать архитектуру и строение кода!

Итак, наличие автоматизированного пакета модульных тестов, охватывающих код продукта, имеет важнейшее значение для чистоты и ясности архитектуры.

А причина заключается в том, что тесты обеспечивают возможность внесения изменения.

Таким образом, если ваши тесты недостаточно чисты и проработаны, ваши возможности по изменению кода сокращаются и вы лишаетесь возможности улучшения структуры кода. Некачественные тесты приводит к некачественному коду продукта. В конечном итоге тестирование вообще становятся невозможным, и код продукта начинает загнивать.

Чистые тесты

Какими отличительными признаками характеризуется чистый тест? Тремя: удобочитаемостью, удобочитаемостью и удобочитаемостью. Вероятно, удобочитаемость в модульных тестах играет еще более важную роль, чем в коде продукта. Что делает тестовый код удобочитаемым? То же, что делает удобочитаемым любой другой код: ясность, простота и выразительность. В тестовом коде необходимо передать максимум информации минимумом выразительных средств.

В листинге 9.1 приведен фрагмент кода из проекта FitNesse. Эти три теста трудны для понимания; несомненно, их можно усовершенствовать. Прежде всего, повторные вызовы `addPage` и `assertSubString` содержат огромное количество повторяющегося кода [G5]. Что еще важнее, код просто забит второстепенными подробностями, снижающими выразительность теста.

Листинг 9.1. SerializedPageResponderTest.java

```
public void testGetPageHieratchyAsXml() throws Exception
{
  crawler.addPage(root, PathParser.parse("PageOne"));
  crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.parse("PageTwo"));

  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
      (SimpleResponse) responder.makeResponse(
         new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
}

public void testGetPageHieratchyAsXmlDoesntContainSymbolicLinks()
  throws Exception
```

```
{
  WikiPage pageOne = crawler.addPage(root, PathParser.parse("PageOne"));
  crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.parse("PageTwo"));

  PageData data = pageOne.getData();
  WikiPageProperties properties = data.getProperties();
  WikiPageProperty symLinks = properties.set(SymbolicPage.PROPERTY_NAME);
  symLinks.set("SymPage", "PageTwo");
  pageOne.commit(data);

  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
      (SimpleResponse) responder.makeResponse(
         new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
  assertNotSubString("SymPage", xml);
}

public void testGetDataAsHtml() throws Exception
{
  crawler.addPage(root, PathParser.parse("TestPageOne"), "test page");

  request.setResource("TestPageOne");
  request.addInput("type", "data");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
       (SimpleResponse) responder.makeResponse(
          new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("test page", xml);
  assertSubString("<Test", xml);
}
```

Например, присмотритесь к вызовам `PathParser`, преобразующим строки в экземпляры `PagePath`, используемые обходчиками (crawlers). Это преобразование абсолютно несущественно для целей тестирования и только затемняет намерения автора. Второстепенные подробности, окружающие создание ответчика, а также сбор и преобразование ответа тоже представляют собой обычный шум. Также обратите внимание на неуклюжий способ построения URL-адреса запроса из ресурса и аргумента. (Я участвовал в написании этого кода, поэтому считаю, что вправе критиковать его.)

В общем, этот код не предназначался для чтения. На несчастного читателя обрушивается целый водопад мелочей, в которых необходимо разобраться, чтобы уловить в тестах хоть какой-то смысл.

Теперь рассмотрим усовершенствованные тесты в листинге 9.2. Они делают абсолютно то же самое, но код был переработан в более ясную и выразительную форму.

Листинг 9.2. SerializedPageResponderTest.java (переработанная версия)

```
public void testGetPageHierarchyAsXml() throws Exception {
  makePages("PageOne", "PageOne.ChildOne", "PageTwo");

  submitRequest("root", "type:pages");

  assertResponseIsXML();
  assertResponseContains(
    "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
  );
}

public void testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception {
  WikiPage page = makePage("PageOne");
  makePages("PageOne.ChildOne", "PageTwo");

  addLinkTo(page, "PageTwo", "SymPage");

  submitRequest("root", "type:pages");

  assertResponseIsXML();
  assertResponseContains(
    "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
  );
  assertResponseDoesNotContain("SymPage");
}

public void testGetDataAsXml() throws Exception {
  makePageWithContent("TestPageOne", "test page");

  submitRequest("TestPageOne", "type:data");

  assertResponseIsXML();
  assertResponseContains("test page", "<Test");
}
```

В структуре тестов очевидно воплощен паттерн ПОСТРОЕНИЕ-ОПЕРАЦИИ-ПРОВЕРКА[1]. Каждый тест четко делится на три части. Первая часть строит тестовые данные, вторая часть выполняет операции с тестовыми данными, а третья часть проверяет, что операция привела к ожидаемым результатам.

[1] http://tnesse.org/FitNesse.AcceptanceTestPatterns.

Обратите внимание: большая часть раздражающих мелочей исчезла. Тесты не делают ничего лишнего, и в них используются только действительно необходимые типы данных и функции.

Любой программист, читающий эти тесты, очень быстро разберется в том, что они делают, не сбиваясь с пути и не увязнув в лишних подробностях.

Предметно-ориентированный язык тестирования

Тесты в листинге 9.2 демонстрируют методику построения предметно-ориентированного языка для программирования тестов. Вместо вызова функций API, используемых программистами для манипуляций с системой, мы строим набор функций и служебных программ, использующих API; это упрощает написание и чтение тестов. Наши функции и служебные программы образуют специализированный API, то есть по сути — язык тестирования, который программисты используют для упрощения работы над тестами, а также чтобы помочь другим программистам, которые будут читать эти тесты позднее.

Тестовый API не проектируется заранее; он развивается на базе многократной переработки тестового кода, перегруженного ненужными подробностями. По аналогии с тем, как я переработал листинг 9.1 в листинг 9.2, дисциплинированные разработчики перерабатывают свой тестовый код в более лаконичные и выразительные формы.

Двойной стандарт

Группа, о которой я упоминал в начале этой главы, в определенном смысле была права. Код тестового API подчиняется несколько иным техническим стандартам, чем код продукта. Он также должен быть простым, лаконичным и выразительным, но от него не требуется такая эффективность. В конце концов, тестовый код работает в тестовой среде, а не в среде реальной эксплуатации продукта, а эти среды весьма заметно различаются по своим потребностям.

Рассмотрим тест из листинга 9.3. Я написал его в ходе работы над прототипом системы контроля окружающей среды. Не вдаваясь в подробности, скажу, что тест этот проверяет, что при слишком низкой температуре включается механизм оповещения о низкой температуре, обогреватель и система подачи нагретого воздуха.

Листинг 9.3. EnvironmentControllerTest.java

```
@Test
  public void turnOnLoTempAlarmAtThreashold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
```

продолжение ⇩

Листинг 9.3 (продолжение)

```
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
  }
```

Конечно, этот листинг содержит множество ненужных подробностей. Например, что делает функция `tic`? Я бы предпочел, чтобы читатель не задумывался об этом в ходе чтения теста. Читатель должен думать о другом: соответствует ли конечное состояние системы его представлениям о «слишком низкой» температуре.

Обратите внимание: в ходе чтения теста вам постоянно приходится переключаться между названием проверяемого состояния и условием проверки. Вы смотрите на `heaterState` (состояние обогревателя), а затем ваш взгляд скользит налево к `assertTrue`. Вы смотрите на `coolerState` (состояние охладителя), а ваш взгляд отступает к `assertFalse`. Все эти перемещения утомительны и ненадежны. Они усложняют чтение теста.

В листинге 9.4 представлена новая форма теста, которая читается гораздо проще.

Листинг 9.4. EnvironmentControllerTest.java (переработанная версия)

```
@Test
  public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
  }
```

Конечно, я скрыл функцию `tic`, создав более понятную функцию `wayTooCold`. Но особого внимания заслуживает странная строка в вызове `assertEquals`. Верхний регистр означает включенное состояние, нижний регистр — выключенное состояние, а буквы всегда следуют в определенном порядке: *{обогреватель, подача воздуха, охладитель, сигнал о высокой температуре, сигнал о низкой температуре}*.

Хотя такая форма близка к нарушению правила о мысленных преобразованиях[1], в данном случае она выглядит уместной. Если вам известен смысл этих обозначений, ваш взгляд скользит по строке в одном направлении и вы можете быстро интерпретировать результаты. Чтение таких тестов почти что доставляет удовольствие. Взгляните на листинг 9.5 и убедитесь, как легко понять их смысл.

Листинг 9.5. EnvironmentControllerTest.java (расширенный набор)

```
@Test
  public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
    tooHot();
    assertEquals("hBChl", hw.getState());
  }

  @Test
  public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
    tooCold();
```

[1] См. «Избегайте мысленных преобразований», с. 47.

```
    assertEquals("HBchl", hw.getState());
  }

  @Test
  public void turnOnHiTempAlarmAtThreshold() throws Exception {
    wayTooHot();
    assertEquals("hBCHl", hw.getState());
  }

  @Test
  public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
  }
```

Функция `getState` приведена в листинге 9.6. Обратите внимание: эффективность этого кода оставляет желать лучшего. Чтобы сделать его более эффективным, вероятно, мне стоило использовать класс `StringBuffer`.

Листинг 9.6. MockControlHardware.java

```
public String getState() {
    String state = "";
    state += heater ? "H" : "h";
    state += blower ? "B" : "b";
    state += cooler ? "C" : "c";
    state += hiTempAlarm ? "H" : "h";
    state += loTempAlarm ? "L" : "l";
    return state;
  }
```

Класс `StringBuffer` некрасив и неудобен. Даже в коде продукта я стараюсь избегать его, если это не приводит к большим потерям; конечно, в коде из листинга 9.6 потери невелики. Однако следует учитывать, что приложение пишется для встроенной системы реального времени, в которой вычислительные ресурсы и память сильно ограничены. С другой стороны, в среде тестирования такие ограничения отсутствуют.

В этом проявляется природа двойного стандарта. Многое из того, что вы никогда не станете делать в среде эксплуатации продукта, абсолютно нормально выглядит в среде тестирования. Обычно речь идет о затратах памяти или эффективности работы процессора — но *никогда* о проблемах чистоты кода.

Одна проверка на тест

Существует точка зрения[1], согласно которой каждая тестовая функция в тесте JUnit должна содержать одну — и только одну — директиву `assert`. Такое правило

[1] См. запись в блоге Дейва Астела (Dave Astel): http://www.artima.com/weblogs/viewpost.jsp?thread=35578.

может показаться излишне жестким, но его преимущества наглядно видны в листинге 9.5. Тесты приводят к одному выводу, который можно быстро и легко понять при чтении.

Но что вы скажете о листинге 9.2? В нем объединена проверка двух условий: что выходные данные представлены в формате XML и они содержат некоторые подстроки. На первый взгляд такое решение выглядит сомнительно. Впрочем, тест можно разбить на два отдельных теста, каждый из которых имеет собственную директиву `assert`, как показано в листинге 9.7.

Листинг 9.7. SerializedPageResponderTest.java (одна директива assert)

```
public void testGetPageHierarchyAsXml() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

    whenRequestIsIssued("root", "type:pages");

    thenResponseShouldBeXML();
  }

  public void testGetPageHierarchyHasRightTags() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

    whenRequestIsIssued("root", "type:pages");

    thenResponseShouldContain(
      "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
    );
  }
```

Обратите внимание: я переименовал функции в соответствии со стандартной схемой `given-when-then` [RSpec]. Это еще сильнее упрощает чтение тестов. К сожалению, такое разбиение приводит к появлению большого количества дублирующегося кода.

Чтобы избежать дублирования, можно воспользоваться паттерном ШАБЛОННЫЙ МЕТОД [GOF], включить части `given/when` в базовый класс, а части `then` — в различные производные классы. А можно создать отдельный тестовый класс, поместить части `given` и `when` в функцию `@Before`, а части `then` — в каждую функцию `@Test`. Но похоже, такой механизм слишком сложен для столь незначительной проблемы. В конечном итоге я предпочел решение с множественными директивами `assert` из листинга 9.2.

Я думаю, что правило «одного `assert`» является хорошей рекомендацией. Обычно я стараюсь создать предметно-ориентированный язык тестирования, который это правило поддерживает, как в листинге 9.5. Но при этом я не боюсь включать в свои тесты более одной директивы `assert`. Вероятно, лучше всего сказать, что количество директив `assert` в тесте должно быть сведено к минимуму.

Одна концепция на тест

Пожалуй, более полезное правило гласит, что в каждой тестовой функции должна тестироваться одна концепция. Мы не хотим, чтобы длинные тестовые функции выполняли несколько разнородных проверок одну за другой. Листинг 9.8 содержит типичный пример такого рода. Этот тест следовало бы разбить на три независимых теста, потому что в нем выполняются три независимых проверки. Объединение их в одной функции заставляет читателя гадать, почему в функцию включается каждая секция, и какое условие проверяется в этой секции.

Листинг 9.8

```
/**
 * Тесты для метода addMonths().
 */
public void testAddMonths() {
    SerialDate d1 = SerialDate.createInstance(31, 5, 2004);

    SerialDate d2 = SerialDate.addMonths(1, d1);
    assertEquals(30, d2.getDayOfMonth());
    assertEquals(6, d2.getMonth());
    assertEquals(2004, d2.getYYYY());

    SerialDate d3 = SerialDate.addMonths(2, d1);
    assertEquals(31, d3.getDayOfMonth());
    assertEquals(7, d3.getMonth());
    assertEquals(2004, d3.getYYYY());

    SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
    assertEquals(30, d4.getDayOfMonth());
    assertEquals(7, d4.getMonth());
    assertEquals(2004, d4.getYYYY());
}
```

Вероятно, три тестовые функции должны выглядеть так:

- Given: последний день месяца, состоящего из 31 дня (например, май).
  1) When: при добавлении одного месяца, последним днем которого является 30-е число (например, июнь), датой должно быть 30-е число этого месяца, а не 31-е.
  2) When: при добавлении двух месяцев, когда последним днем второго месяца является 31-е число, датой должно быть 31-е число.
- Given: последний день месяца, состоящего из 30 дней (например, июнь).
  1) When: при добавлении одного месяца, последним днем которого является 31-е число, датой должно быть 30-е число этого месяца, а не 31-е.

В такой формулировке видно, что среди разнородных тестов скрывается одно общее правило. При увеличении месяца дата не может превысить последнее число нового месяца. Следовательно, в результате увеличение месяца для 28 февраля

должна быть получена дата 28 марта. В текущей версии это условие не проверяется, хотя такой тест был бы полезен.

Таким образом, проблема не в множественных директивах `assert` в разных секциях листинга 9.8. Проблема в том, что тест проверяет более одной концепции. Так что, вероятно, лучше всего сформулировать это правило так: количество директив `assert` на концепцию должно быть минимальным, и в тестовой функции должна проверяться только одна концепция.

F.I.R.S.T.[1]

Чистые тесты должны обладать еще пятью характеристиками, названия которых образуют приведенное сокращение.

Быстрота (Fast). Тесты должны выполняться быстро. Если тесты выполняются медленно, вам не захочется часто запускать их. Без частого запуска тестов проблемы не будут выявляться на достаточно ранней стадии, когда они особенно легко исправляются. В итоге вы уже не так спокойно относитесь к чистке своего кода, и со временем код начинает загнивать.

Независимость (Independent). Тесты не должны зависеть друг от друга. Один тест не должен создавать условия для выполнения следующего теста. Все тесты должны выполняться независимо и в любом порядке на ваше усмотрение. Если тесты зависят друг от друга, то при первом отказе возникает целый каскад сбоев, который усложняет диагностику и скрывает дефекты в зависимых тестах.

Повторяемость (Repeatable). Тесты должны давать повторяемые результаты в любой среде. Вы должны иметь возможность выполнить тесты в среде реальной эксплуатации, в среде тестирования или на вашем ноутбуке во время возвращения домой с работы. Если ваши тесты не будут давать однозначных результатов в любых условиях, вы всегда сможете найти отговорку для объяснения неудач. Также вы лишитесь возможности проводить тестирование, если нужная среда недоступна.

Очевидность (Self-Validating). Результатом выполнения теста должен быть логический признак. Тест либо прошел, либо не прошел. Чтобы узнать результат, пользователь не должен читать журнальный файл. Не заставляйте его вручную сравнивать два разных текстовых файла. Если результат теста не очевиден, то отказы приобретают субъективный характер, а выполнение тестов может потребовать долгой ручной обработки данных.

Своевременность (Timely). Тесты должны создаваться своевременно. Модульные тесты пишутся *непосредственно перед* кодом продукта, обеспечивающим их прохождение. Если вы пишете тесты после кода продукта, вы можете решить, что тестирование кода продукта создает слишком много трудностей, а все из-за того, что удобство тестирования не учитывалось при проектировании кода продукта.

[1] Учебные материалы Object Mentor.

Заключение

В этой главе мы едва затронули тему тестирования. Я думаю, что на тему *чистых тестов* можно было бы написать целую книгу. Для «здоровья» проекта тесты не менее важны, чем код продукта. А может быть, они еще важнее, потому что тесты сохраняют и улучшают гибкость, удобство сопровождения и возможности повторного использования кода продукта. Постоянно следите за чистотой своих тестов. Постарайтесь сделать их выразительными и лаконичными. Изобретайте тестовые API, которым отводится роль предметно-ориентированного языка тестирования, упрощающего написание тестов.

Если вы будете пренебрежительно относиться к тестам, то и ваш код начнет загнивать. Поддерживайте чистоту в своих тестах.

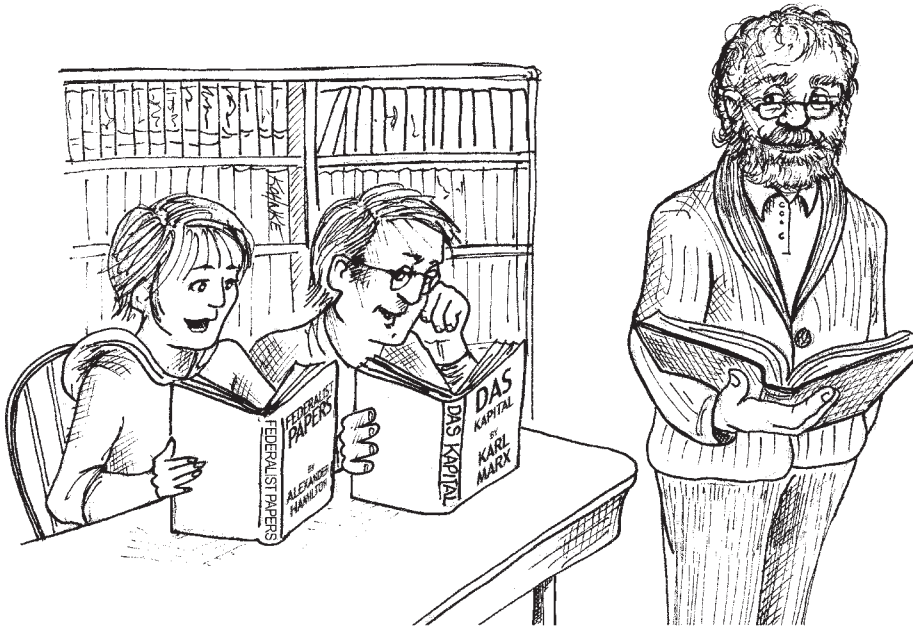
Литература

[RSpec]: RSpec: Behavior Driven Development for Ruby Programmers, Aslak Hellesy, David Chelimsky, Pragmatic Bookshelf, 2008.

[GOF]: Design Patterns: Elements of Reusable Object Oriented Software, Gamma et al., Addison-Wesley, 1996.

10 Классы

Совместно с Джеффом Лангром



До настоящего момента наше внимание было сосредоточено исключительно на том, как качественно написать строки и блоки кода. Мы разобрались с правильной композицией функций и их взаимосвязями. Но какими бы выразительными ни были функции и содержащиеся в них команды, мы не добьемся чистоты кода до тех пор, пока не обратим внимание на более высокие уровни организации кода. В этой главе речь пойдет о чистоте классов.

Строение класса

По стандартным правилам Java класс должен начинаться со списка переменных. Сначала перечисляются открытые статические константы. Далее следуют приватные статические переменные, а за ними идут приватные переменные экзем-

пляров. Открытых переменных обычно нет, трудно найти веские причины для их использования.

За списком переменных обычно следуют открытые функции. Мы предпочитаем размещать приватные вспомогательные функции, вызываемые открытыми функциями, непосредственно за самой открытой функцией. Такое размещение соответствует правилу понижения, в результате чего программа читается как газетная статья.

Инкапсуляция

Мы предпочитаем объявлять переменные и вспомогательные функции приватными, но относимся к ним без фанатизма. Иногда переменную или вспомогательную функцию приходится объявлять защищенной, чтобы иметь возможность обратиться к ней из теста. С нашей точки зрения тесты исключительно важны. Если тест из того же пакета должен вызвать функцию или обратиться к переменной, мы используем защищенный или пакетный уровень доступа. Тем не менее начинать следует с поиска способа, сохраняющего приватность. Ослабление инкапсуляции всегда должно быть последней мерой.

Классы должны быть компактными!

Первое правило: классы должны быть компактными. Второе правило: классы должны быть еще компактнее. Нет, мы не собираемся повторять текст из главы 3. Но как и в случае с функциями, компактность должна стать основным правилом проектирования классов. И для классов начинать следует с вопроса: «А насколько компактными?»

Размер функций определяется количеством физических строк. В классах используется другая метрика; мы подсчитываем *ответственности* [RDD].

В листинге 10.1 представлен класс `SuperDashboard`, предоставляющий около 70 открытых методов. Большинство разработчиков согласится с тем, что это перебор.

Листинг 10.1. Слишком много ответственностей

```
public class SuperDashboard extends JFrame implements MetaDataUser
   public String getCustomizerLanguagePath()
   public void setSystemConfigPath(String systemConfigPath)
   public String getSystemConfigDocument()
   public void setSystemConfigDocument(String systemConfigDocument)
   public boolean getGuruState()
   public boolean getNoviceState()
   public boolean getOpenSourceState()
   public void showObject(MetaObject object)
   public void showProgress(String s)
   public boolean isMetadataDirty()
   public void setIsMetadataDirty(boolean isMetadataDirty)
   public Component getLastFocusedComponent()
   public void setLastFocused(Component lastFocused)
```

продолжение ⇩

Листинг 10.1 (продолжение)

```
public void setMouseSelectState(boolean isMouseSelected)
public boolean isMouseSelected()
public LanguageManager getLanguageManager()
public Project getProject()
public Project getFirstProject()
public Project getLastProject()
public String getNewProjectName()
public void setComponentSizes(Dimension dim)
public String getCurrentDir()
public void setCurrentDir(String newDir)
public void updateStatus(int dotPos, int markPos)
public Class[] getDataBaseClasses()
public MetadataFeeder getMetadataFeeder()
public void addProject(Project project)
public boolean setCurrentProject(Project project)
public boolean removeProject(Project project)
public MetaProjectHeader getProgramMetadata()
public void resetDashboard()
public Project loadProject(String fileName, String projectName)
public void setCanSaveMetadata(boolean canSave)
public MetaObject getSelectedObject()
public void deselectObjects()
public void setProject(Project project)
public void editorAction(String actionName, ActionEvent event)
public void setMode(int mode)
public FileManager getFileManager()
public void setFileManager(FileManager fileManager)
public ConfigManager getConfigManager()
public void setConfigManager(ConfigManager configManager)
public ClassLoader getClassLoader()
public void setClassLoader(ClassLoader classLoader)
public Properties getProps()
public String getUserHome()
public String getBaseDir()
public int getMajorVersionNumber()
public int getMinorVersionNumber()
public int getBuildNumber()
public MetaObject pasting(
MetaObject target, MetaObject pasted, MetaProject project)
public void processMenuItems(MetaObject metaObject)
public void processMenuSeparators(MetaObject metaObject)
public void processTabPages(MetaObject metaObject)
public void processPlacement(MetaObject object)
public void processCreateLayout(MetaObject object)
public void updateDisplayLayer(MetaObject object, int layerIndex)
public void propertyEditedRepaint(MetaObject object)
public void processDeleteObject(MetaObject object)
public boolean getAttachedToDesigner()
public void processProjectChangedState(boolean hasProjectChanged)
public void processObjectNameChanged(MetaObject object)
public void runProject()
```

```
    public void setAçowDragging(boolean allowDragging)
    public boolean allowDragging()
    public boolean isCustomizing()
    public void setTitle(String title)
    public IdeMenuBar getIdeMenuBar()
    public void showHelper(MetaObject metaObject, String propertyName)
    // ... и еще много других, не-открытых методов...
}
```

А если бы класс SuperDashboard содержал только методы, приведенные в листинге 10.2?

Листинг 10.2. Достаточно компактно?

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

Пять методов — не слишком много, не так ли? В нашем случае слишком, потому что несмотря на малое количество методов, класс SuperDashboard по-прежнему имеет слишком много ответственностей.

Имя класса должно описывать его ответственности. В сущности, имя должно стать первым фактором, способствующим определению размера класса. Если для класса не удается подобрать четкое, короткое имя, вероятно, он слишком велик. Чем туманнее имя класса, тем больше вероятность, что он имеет слишком много ответственностей. В частности, присутствие в именах классов словпроныр «Processor», «Manager» и «Super» часто свидетельствует о нежелательном объединении ответственностей.

Краткое описание класса должно укладываться примерно в 25 слов, без выражений «если», «и», «или» и «но». Как бы вы описали класс SuperDashboard? «Класс предоставляет доступ к компоненту, который последним имел фокус ввода, и позволяет отслеживать номера версии и сборки». Первое «и» указывает на то, что SuperDashboard имеет слишком много ответственностей.

Принцип единой ответственности (SRP)

Принцип единой ответственности (SRP[1]) утверждает, что класс или модуль должен иметь одну — и только одну — причину для изменения. Этот принцип дает нам как определение ответственности, так и критерий для оценки размера класса. Классы должны иметь одну ответственность, то есть одну причину для изменений.

[1] За более подробной информацией об этом принципе обращайтесь к [PPP].

Небольшой, казалось бы, класс SuperDashboard в листинге 10.2 имеет две причины для изменений. Во-первых, он отслеживает версию, которая, вероятно, будет изменяться при каждом обновлении продукта. Во-вторых, он управляет компонентами Java Swing (потомки класса JFrame, представляющего графическое окно верхнего уровня в Swing). Несомненно, номер версии должен обновляться при любых изменениях кода Swing, но обратное не всегда верно: номер версии также может изменяться вследствие изменений в другом коде системы.

Попытки идентификации ответственностей (причин для изменения) часто помогают выявить и создать более качественные абстракции для нашего кода. Все три метода SuperDashboard, относящиеся к версии, легко выделяются в отдельный класс с именем Version (листинг 10.3). Класс Version обладает хорошим потенциалом для повторного использования в других приложениях!

Листинг 10.3. Класс с единой ответственностью

```
public class Version {
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

Принцип единой ответственности — одна из самых важных концепций в объектно-ориентированном проектировании. Кроме того, его относительно несложно понять и соблюдать. Но как ни странно, принцип единой ответственности часто оказывается самым нарушаемым принципом проектирования классов. Мы постоянно встречаем классы, которые делают слишком много всего. Почему?

Заставить программу работать и написать чистый код — совершенно разные вещи. Обычно мы думаем прежде всего о том, чтобы наш код заработал, а не о его структуре и чистоте. И это абсолютно законно. Разделение ответственности в работе программиста играет не менее важную роль, чем в наших программах.

К сожалению, слишком многие из нас полагают, что после того, как программа заработает, их работа закончена. Мы не переключаемся на усовершенствование ее структуры и чистоты. Мы переходим к следующей задаче вместо того, чтобы сделать шаг назад и разделить разбухшие классы на отдельные блоки с единой ответственностью.

В то же время многие разработчики опасаются, что множество небольших узкоспециализированных классов затруднит понимание общей картины. Их беспокоит то, что им придется переходить от класса к классу, чтобы разобраться в том, как решается более крупная задача.

Однако система с множеством малых классов имеет не больше «подвижных частей», чем система с несколькими большими классами. В последней тоже придется разбираться, и это будет ничуть не проще. Так что вопрос заключается в следующем: хотите ли вы, чтобы ваши инструменты были разложены по ящикам с множеством небольших отделений, содержащих четко определенные

и подписанные компоненты? Или вы предпочитаете несколько больших ящиков, в которые можно сваливать все подряд?

Каждая крупная система содержит большой объем рабочей логики и обладает высокой сложностью. Первоочередной целью управления этой сложностью является *формирование структуры*, при которой разработчик знает, где искать то, что ему требуется, и в любой момент времени может досконально знать только ту часть системы, которая непосредственно относится к его работе. Напротив, в системе с большими, многоцелевыми классами нам неизбежно приходится разбираться с множеством аспектов, которые в данный момент нас не интересуют.

Еще раз выделю основные моменты: система должна состоять из множества мелких классов, а не из небольшого числа больших. Каждый класс инкапсулирует одну ответственность, имеет одну причину для изменения и взаимодействует с другими классами для реализации желаемого поведения системы.

Связность

Классы должны иметь небольшое количество переменных экземпляров. Каждый метод класса должен оперировать с одной или несколькими из этих переменных. В общем случае, чем с большим количеством переменных работает метод, тем выше связность этого метода со своим классом. Класс, в котором каждая переменная используется каждым методом, обладает максимальной связностью.

Как правило, создавать классы с максимальной связностью не рекомендуется... а скорее всего, это нереально. С другой стороны, связность класса должна быть высокой. Высокая связность означает, что методы и переменные класса взаимозависимы и существуют как единое целое.

Рассмотрим реализацию стека из листинга 10.4. Этот класс обладает очень высокой связностью. Из трех его методов только `size()` не использует обе переменные.

Листинг 10.4. Stack.java — класс с высокой связностью

```
public class Stack {
  private int topOfStack = 0;
  List<Integer> elements = new LinkedList<Integer>();

  public int size() {
    return topOfStack;
  }

  public void push(int element) {
    topOfStack++;
    elements.add(element);
  }

  public int pop() throws PoppedWhenEmpty {
    if (topOfStack == 0)
      throw new PoppedWhenEmpty();
```

продолжение ⇩

Листинг 10.4 (продолжение)

```
    int element = elements.get(--topOfStack);
    elements.remove(topOfStack);
    return element;
  }
}
```

Стратегия компактных функций и коротких списков параметров иногда приводит к росту переменных экземпляров, используемых подмножеством методов. Это почти всегда свидетельствует о том, что по крайней мере один класс пытается выделиться из более крупного класса. Постарайтесь разделить переменные и методы на два и более класса, чтобы новые классы обладали более высокой связностью.

Поддержание связности приводит к уменьшению классов

Сам акт разбиения больших функций на меньшие приводит к росту количества классов. Допустим, имеется большая функция, в которой объявлено много переменных. Вы хотите выделить один небольшой фрагмент этой функции в отдельную функцию. Однако выделяемый код использует четыре переменные, объявленные в исходной функции. Может, передать все четыре переменные новой функции в виде аргументов?

Ни в коем случае! Преобразовав эти четыре переменные в переменные экземпляров класса, мы сможем выделить код без передачи переменных. Таким образом, разбиение функции на меньшие фрагменты упрощается.

К сожалению, это также означает, что наши классы теряют связность, потому что в них накапливается все больше переменных экземпляров, созданных исключительно для того, чтобы они могли совместно использоваться небольшим подмножеством функций. Но постойте! Если группа функций должна работать с некоторыми переменными, не образуют ли они класс сами по себе? Конечно, образуют. Если классы утрачивают связность, разбейте их!

Таким образом, разбиение большой функции на много мелких функций также часто открывает возможность для выделения нескольких меньших классов. В результате строение программы улучшается, а ее структура становится более прозрачной.

Для демонстрации мы воспользуемся проверенным временем примером из замечательной книги Кнута «Literate Programming» [Knuth92]. В листинге 10.5 представлена программа Кнута `PrintPrimes`, переведенная на Java. Справедливости ради стоит отметить, что это не та программа, которую написал Кнут, а та, которую выводит его утилита WEB. Я воспользуюсь ей, потому что она является отличной отправной точкой для разбиения большой функции на несколько меньших функций и классов.

Листинг 10.5. PrintPrimes.java

```
package literatePrimes;

public class PrintPrimes {
  public static void main(String[] args) {
    final int M = 1000;
    final int RR = 50;
    final int CC = 4;
    final int WW = 10;
    final int ORDMAX = 30;
    int P[] = new int[M + 1];
    int PAGENUMBER;
    int PAGEOFFSET;
    int ROWOFFSET;
    int C;
    int J;
    int K;
    boolean JPRIME;
    int ORD;
    int SQUARE;
    int N;
    int MULT[] = new int[ORDMAX + 1];

    J = 1;
    K = 1;
    P[1] = 2;
    ORD = 2;
    SQUARE = 9;

    while (K < M) {
      do {
        J = J + 2;
        if (J == SQUARE) {
          ORD = ORD + 1;
          SQUARE = P[ORD] * P[ORD];
          MULT[ORD - 1] = J;
        }
        N = 2;
        JPRIME = true;
        while (N < ORD && JPRIME) {
          while (MULT[N] < J)
            MULT[N] = MULT[N] + P[N] + P[N];
          if (MULT[N] == J)
            JPRIME = false;
          N = N + 1;
        }
      } while (!JPRIME);
      K = K + 1;
      P[K] = J;
    }
    {
```

продолжение ⇩

Листинг 10.5 (продолжение)

```
      PAGENUMBER = 1;
      PAGEOFFSET = 1;
      while (PAGEOFFSET <= M) {
        System.out.println("The First " + M +
                           " Prime Numbers --- Page " + PAGENUMBER);
        System.out.println("");
        for (ROWOFFSET = PAGEOFFSET; ROWOFFSET < PAGEOFFSET + RR; ROWOFFSET++){
          for (C = 0; C < CC;C++)
            if (ROWOFFSET + C * RR <= M)
              System.out.format("%10d", P[ROWOFFSET + C * RR]);
          System.out.println("");
        }
        System.out.println("\f");
        PAGENUMBER = PAGENUMBER + 1;
        PAGEOFFSET = PAGEOFFSET + RR * CC;
      }
    }
  }
}
```

Записанная в виде одной функции, эта программа представляет собой полную неразбериху. Многоуровневая вложенность, множество странных переменных, структура с жесткой привязкой... По крайней мере, одну большую функцию следует разбить на несколько меньших функций.

В листингах 10.6–10.8 показано, что получается после разбиения кода из листинга 10.5 на меньшие классы и функции, с выбором осмысленных имен для классов, функций и переменных.

Листинг 10.6. PrimePrinter.java (переработанная версия)

```
package literatePrimes;

public class PrimePrinter {
  public static void main(String[] args) {
    final int NUMBER_OF_PRIMES = 1000;
    int[] primes = PrimeGenerator.generate(NUMBER_OF_PRIMES);

    final int ROWS_PER_PAGE = 50;
    final int COLUMNS_PER_PAGE = 4;
    RowColumnPagePrinter tablePrinter =
      new RowColumnPagePrinter(ROWS_PER_PAGE,
                               COLUMNS_PER_PAGE,
                               "The First " + NUMBER_OF_PRIMES +
                               " Prime Numbers");

    tablePrinter.print(primes);
  }

}
```

Листинг 10.7. RowColumnPagePrinter.java

```
package literatePrimes;

import java.io.PrintStream;

public class RowColumnPagePrinter {
  private int rowsPerPage;
  private int columnsPerPage;
  private int numbersPerPage;
  private String pageHeader;
  private PrintStream printStream;

  public RowColumnPagePrinter(int rowsPerPage,
                              int columnsPerPage,
                              String pageHeader) {
    this.rowsPerPage = rowsPerPage;
    this.columnsPerPage = columnsPerPage;
    this.pageHeader = pageHeader;
    numbersPerPage = rowsPerPage * columnsPerPage;
    printStream = System.out;
  }

  public void print(int data[]) {
    int pageNumber = 1;
    for (int firstIndexOnPage = 0;
         firstIndexOnPage < data.length;
         firstIndexOnPage += numbersPerPage) {
      int lastIndexOnPage =
        Math.min(firstIndexOnPage + numbersPerPage - 1,
                 data.length - 1);
      printPageHeader(pageHeader, pageNumber);
      printPage(firstIndexOnPage, lastIndexOnPage, data);
      printStream.println("\f");
      pageNumber++;
    }
  }

  private void printPage(int firstIndexOnPage,
                         int lastIndexOnPage,
                         int[] data) {
    int firstIndexOfLastRowOnPage =
      firstIndexOnPage + rowsPerPage - 1;
    for (int firstIndexInRow = firstIndexOnPage;
         firstIndexInRow <= firstIndexOfLastRowOnPage;
         firstIndexInRow++) {
      printRow(firstIndexInRow, lastIndexOnPage, data);
      printStream.println("");
    }
  }
```

продолжение ⇩

Листинг 10.7 (продолжение)

```
  private void printRow(int firstIndexInRow,
                        int lastIndexOnPage,
                        int[] data) {
    for (int column = 0; column < columnsPerPage; column++) {
      int index = firstIndexInRow + column * rowsPerPage;
      if (index <= lastIndexOnPage)
        printStream.format("%10d", data[index]);
    }
  }

  private void printPageHeader(String pageHeader,
                               int pageNumber) {
    printStream.println(pageHeader + " --- Page " + pageNumber);
    printStream.println("");
  }

  public void setOutput(PrintStream printStream) {
    this.printStream = printStream;
  }
}
```

Листинг 10.8. PrimeGenerator.java

```
package literatePrimes;

import java.util.ArrayList;

public class PrimeGenerator {
  private static int[] primes;
  private static ArrayList<Integer> multiplesOfPrimeFactors;

  protected static int[] generate(int n) {
    primes = new int[n];
    multiplesOfPrimeFactors = new ArrayList<Integer>();
    set2AsFirstPrime();
    checkOddNumbersForSubsequentPrimes();
    return primes;
  }

  private static void set2AsFirstPrime() {
    primes[0] = 2;
    multiplesOfPrimeFactors.add(2);
  }

  private static void checkOddNumbersForSubsequentPrimes() {
    int primeIndex = 1;
    for (int candidate = 3;
         primeIndex < primes.length;
         candidate += 2) {
      if (isPrime(candidate))
        primes[primeIndex++] = candidate;
    }
  }
```

```
  private static boolean isPrime(int candidate) {
    if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate)) {
      multiplesOfPrimeFactors.add(candidate);
      return false;
    }
    return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
  }

  private static boolean
  isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate) {
    int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.size()];
    int leastRelevantMultiple = nextLargerPrimeFactor * nextLargerPrimeFactor;
    return candidate == leastRelevantMultiple;
  }

  private static boolean
  isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
    for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
      if (isMultipleOfNthPrimeFactor(candidate, n))
        return false;
    }
    return true;
  }

  private static boolean
  isMultipleOfNthPrimeFactor(int candidate, int n) {
    return
      candidate == smallestOddNthMultipleNotLessThanCandidate(candidate, n);
  }

  private static int
  smallestOddNthMultipleNotLessThanCandidate(int candidate, int n) {
    int multiple = multiplesOfPrimeFactors.get(n);
    while (multiple < candidate)
      multiple += 2 * primes[n];
    multiplesOfPrimeFactors.set(n, multiple);
    return multiple;
  }
}
```

Прежде всего бросается в глаза, что программа стала значительно длиннее. От одной с небольшим страницы она разрослась почти до трех страниц. Это объясняется несколькими причинами. Во-первых, в переработанной программе используются более длинные, более содержательные имена переменных. Во-вторых, объявления функций и классов в переработанной версии используются для комментирования кода. В третьих, пробелы и дополнительное форматирование обеспечивают удобочитаемость программы.

Обратите внимание на логическое разбиение программы в соответствии с тремя основными видами ответственности. Основной код программы содержится в классе `PrimePrinter`; он отвечает за управлении средой выполнения. Именно этот код изменится в случае смены механизма вызова. Например, если в будущем

программа будет преобразована в службу SOAP, то изменения будут внесены в код PrimePrinter.

Класс RowColumnPagePrinter специализируется на форматировании списка чисел в страницы с определенным количеством строк и столбцов. Если потребуется изменить формат вывода, то изменения затронут только этот класс.

Класс PrimeGenerator специализируется на построении списка простых чисел. Создание экземпляров этого класса не предполагается. Класс всего лишь определяет удобную область видимости, в которой можно объявлять и скрывать переменные. Он изменится при изменении алгоритма вычисления простых чисел.

При этом программа не была переписана! Мы не начинали работу «с нуля» и не писали код заново. В самом деле, внимательно присмотревшись к двум программам, вы увидите, что они используют одинаковые алгоритмы и одинаковую механику для решения своих задач.

Модификация началась с написания тестового пакета, досконально проверявшего поведение первой программы. Далее в код последовательно вносились многочисленные мелкие изменения. После каждого изменения проводились тесты, которые подтверждали, что поведение программы не изменилось. Так, шаг за шагом, первая программа очищалась и трансформировалась во вторую.

Структурирование с учетом изменений

Большинство систем находится в процессе непрерывных изменений. Каждое изменение создает риск того, что остальные части системы будут работать не так, как мы ожидаем. В чистой системе классы организованы таким образом, чтобы риск от изменений был сведен к минимуму.

Класс Sql в листинге 10.9 используется для построения правильно сформированных строк SQL по соответствующим метаданным. Работа еще не завершена, поэтому класс не поддерживает многие функции SQL (например, команды update). Когда придет время включения в класс Sql поддержки update, придется «открыть» этот класс для внесения изменений. Но как уже говорилось, открытие класса создает риск. Любые изменения в этом классе создают потенциальную возможность для нарушения работы остального кода класса, поэтому весь код приходится полностью тестировать заново.

Листинг 10.9. Класс, который необходимо открыть для внесения изменений

```
public class Sql {
   public Sql(String table, Column[] columns)
   public String create()
   public String insert(Object[] fields)
   public String selectAll()
   public String findByKey(String keyColumn, String keyValue)
   public String select(Column column, String pattern)
```

```
    public String select(Criteria criteria)
    public String preparedInsert()
    private String columnList(Column[] columns)
    private String valuesList(Object[] fields, final Column[] columns)
    private String selectWithCriteria(String criteria)
    private String placeholderList(Column[] columns)
}
```

Класс `Sql` изменяется при добавлении нового типа команды. Кроме того, он будет изменяться при изменении подробностей реализации уже существующего типа команды — скажем, если нам понадобится изменить функциональность `select` для поддержки подчиненной выборки. Две причины для изменения означают, что класс `Sql` нарушает принцип единой ответственности.

Нарушение принципа единой ответственности проявляется и в структуре кода. Из набора методов `Sql` видно, что класс содержит приватные методы (например, `selectWithCriteria`), относящиеся только к командам `select`.

Приватные методы, действие которых распространяется только на небольшое подмножество класса, — хороший признак для поиска потенциальных усовершенствований. Тем не менее основные усилия следует направить на изменение самой системы. Если бы класс `Sql` выглядел логически завершенным, то нам не пришлось бы беспокоиться о разделении ответственности. Если бы в обозримом будущем функциональность `update` не понадобилась, `Sql` можно было бы оставить в покое. Но как только выясняется, что класс необходимо открыть, нужно рассмотреть возможность усовершенствования его структуры.

Почему бы не воспользоваться решением, представленным в листинге 10.10? Для каждого метода открытого интерфейса, определенного в предыдущей версии `Sql` из листинга 10.9, создается соответствующий класс, производный от `Sql`. При этом приватные методы (такие, как `valuesList`) перемещаются непосредственно туда, где они понадобятся. Общее приватное поведение изолируется в паре вспомогательных классов, `Where` и `ColumnList`.

Листинг 10.10. Набор закрытых классов

```
abstract public class Sql {
    public Sql(String table, Column[] columns)
    abstract public String generate();
}

public class CreateSql extends Sql {
    public CreateSql(String table, Column[] columns)
    @Override public String generate()
}

public class SelectSql extends Sql {
    public SelectSql(String table, Column[] columns)
    @Override public String generate()
}
```

продолжение ⇩

Листинг 10.10 (продолжение)

```
public class InsertSql extends Sql {
   public InsertSql(String table, Column[] columns, Object[] fields)
   @Override public String generate()
   private String valuesList(Object[] fields, final Column[] columns)
}

public class SelectWithCriteriaSql extends Sql {
   public SelectWithCriteriaSql(
      String table, Column[] columns, Criteria criteria)
   @Override public String generate()
}

public class SelectWithMatchSql extends Sql {
   public SelectWithMatchSql(
      String table, Column[] columns, Column column, String pattern)
   @Override public String generate()
}

public class FindByKeySql extends Sql
   public FindByKeySql(
      String table, Column[] columns, String keyColumn, String keyValue)
   @Override public String generate()
}

public class PreparedInsertSql extends Sql {
   public PreparedInsertSql(String table, Column[] columns)
   @Override public String generate() {
   private String placeholderList(Column[] columns)
}

public class Where {
   public Where(String criteria)
   public String generate()
}
```

Код каждого класса становится до смешного простым. Время, необходимое для понимания класса, падает почти до нуля. Вероятность того, что одна из функций нарушит работу другой, ничтожно мала. С точки зрения тестирования проверка всех фрагментов логики в этом решении упрощается, поскольку все классы изолированы друг от друга.

Что не менее важно, когда придет время добавления `update`, вам не придется изменять ни один из существующих классов! Логика построения команды `update` реализуется в новом субклассе `Sql` с именем `UpdateSql`. Это изменение не нарушит работу другого кода в системе.

Переработанная логика `Sql` положительна во всех отношениях. Она поддерживает принцип единой ответственности. Она также поддерживает другой ключевой принцип проектирования классов в ООП, называемый принципом открытости/закрытости [PPP]: классы должны быть открыты для расширений, но закрыты

для модификации. Наш переработанный класс Sql открыт для добавления новой функциональности посредством создания производных классов, но при внесении этого изменения все остальные классы остаются закрытыми. Новый класс UpdateSql просто размещается в положенном месте.

Структура системы должна быть такой, чтобы обновление системы (с добавлением новых или изменением существующих аспектов) создавало как можно меньше проблем. В идеале новая функциональность должна реализовываться расширением системы, а не внесением изменений в существующий код.

Изоляция изменений

Потребности меняются со временем; следовательно, меняется и код. В начальном курсе объектно-ориентированного программирования мы узнали, что классы делятся на конкретные, содержащие подробности реализации (код), и абстрактные, представляющие только концепции. Если клиентский класс зависит от конкретных подробностей, то изменение этих подробностей может нарушить его работоспособность. Чтобы изолировать воздействие этих подробностей на класс, в систему вводятся интерфейсы и абстрактные классы.

Зависимости от конкретики создает проблемы при тестировании системы. Если мы строим класс Portfolio, зависящий от внешнего API TokyoStockExchange для вычисления текущей стоимости портфеля ценных бумаг, наши тестовые сценарии начинают зависеть от ненадежного внешнего фактора. Трудно написать тест, если вы получаете разные ответы каждые пять минут!

Вместо того чтобы проектировать Portfolio с прямой зависимостью от TokyoStockExchange, мы создаем интерфейс StockExchange, в котором объявляется один метод:

```
public interface StockExchange {
   Money currentPrice(String symbol);
}
```

Класс TokyoStockExchange проектируется с расчетом на реализацию этого интерфейса. При ссылке на StockExchange передается в аргументе конструктора Portfolio:

```
public Portfolio {
   private StockExchange exchange;
   public Portfolio(StockExchange exchange) {
      this.exchange = exchange;
   }
   // ...
}
```

Теперь наш тест может создать пригодную для тестирования реализацию интерфейса StockExchange, эмулирующую реальный API TokyoStockExchange. Тестовая реализация задает текущую стоимость каждого вида акций, используемых при тестировании. Если тест демонстрирует приобретение пяти акций Microsoft, мы

кодируем тестовую реализацию так, чтобы для Microsoft всегда возвращалась стоимость $100 за акцию. Тестовая реализация интерфейса StockExchange сводится к простому поиску по таблице. После этого пишется тест, который должен вернуть общую стоимость портфеля в $500:

```
public class PortfolioTest {
  private FixedStockExchangeStub exchange;
  private Portfolio portfolio;

  @Before
  protected void setUp() throws Exception {
    exchange = new FixedStockExchangeStub();
    exchange.fix("MSFT", 100);
    portfolio = new Portfolio(exchange);
  }

  @Test
  public void GivenFiveMSFTTotalShouldBe500() throws Exception {
    portfolio.add(5, "MSFT");
    Assert.assertEquals(500, portfolio.value());
  }
}
```

Если система обладает достаточной логической изоляцией для подобного тестирования, она также становится более гибкой и более подходящей для повторного использования. Отсутствие жестких привязок означает, что элементы системы лучше изолируются друг от друга и от изменений. Изоляция упрощает понимание каждого элемента системы.

Сведение к минимуму логических привязок соответствует другому принципу проектирования классов, известному как *принцип обращения зависимостей* (DIP, Dependency Inversion Principle). По сути DIP гласит, что классы системы должны зависеть от абстракций, а не от конкретных подробностей.

Вместо того чтобы зависеть от подробностей реализации класса TokyoStock-Exchange, наш класс Portfolio теперь зависит от интерфейса StockExchange. Интерфейс StockExchange представляет абстрактную концепцию запроса текущей стоимости акций. Эта абстракция изолирует класс от конкретных подробностей получения такой цены — в том числе и от источника, из которого берется реальная информация.

Литература

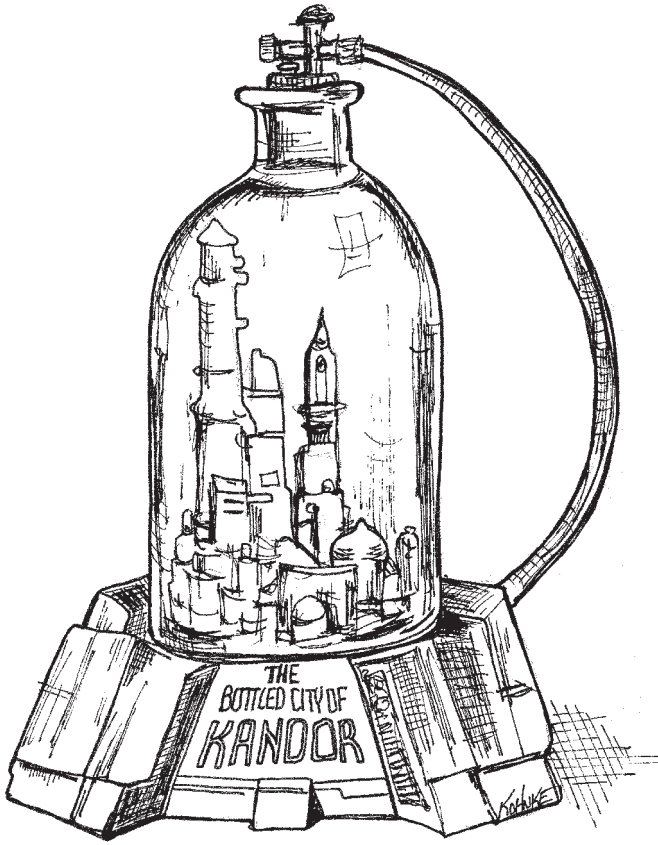
[RDD]: Object Design: Roles, Responsibilities, and Collaborations, Rebecca Wirfs-Brock et al., Addison-Wesley, 2002.

[PPP]: Agile Software Development: Principles, Patterns, and Practices, Robert C. Martin, Prentice Hall, 2002.

[Knuth92]: Literate Programming, Donald E. Knuth, Center for the Study of language and Information, Leland Stanford Junior University, 1992.

11 Системы

Кевин Дин Уомплер



Сложность убивает. Она вытягивает жизненные силы из разработчиков, затрудняя планирование, построение и тестирование продуктов.

Рэй Оззи, технический директор
Microsoft Corporation

Как бы вы строили город?

Смогли бы вы лично разработать план до последней мелочи? Вероятно, нет. Даже управление существующим городом не под силу одному человеку. Да, города работают (в основном). Они работают, потому что в городах есть группы людей, управляющие определенными аспектами городской жизни: водопроводом, электричеством, транспортом, соблюдением законности, правилами застройки и т. д. Одни отвечают за общую картину, другие занимаются мелочами.

Города работают еще и потому, что в них развились правильные уровни абстракции и модульности, которые обеспечивают эффективную работу людей и «компонентов», находящихся под их управлением, — даже без понимания полной картины.

Группы разработки программного обеспечения тоже организуются по аналогичным принципам, но системы, над которыми они работают, часто не имеют аналогичного разделения обязанностей и уровней абстракции. Чистый код помогает достичь этой цели на нижних уровнях абстракции. В этой главе мы поговорим о том, как сохранить чистоту на более высоких уровнях, то есть на уровне *системы*.

Отделение конструирования системы от ее использования

Прежде всего необходимо понять, что конструирование и использование системы — два совершенно разных процесса. Когда я пишу эти строки, из моего окна в Чикаго виден новый строящийся отель. Сейчас это голая бетонная коробка со строительным краном и лифтом, закрепленным на наружной стене. Все рабочие носят каски и спецовки. Через год-другой строительство будет завершено. Кран и служебный лифт исчезнут. Здание очистится, заблестит стеклянными окнами и новой краской. Люди, работающие и останавливающиеся в нем, тоже будут выглядеть совершенно иначе.

В программных системах фаза инициализации, в которой конструируются объекты приложения и «склеиваются» основные зависимости, тоже должна отделяться от логики времени выполнения, получающей управление после ее завершения.

Фаза *инициализации* присутствует в каждом приложении. Это первая из *областей ответственности* (concerns), которую мы рассмотрим в этой главе, а сама концепция *разделения ответственности* относится к числу самых старых и важных приемов нашего ремесла.

К сожалению, во многих приложениях такое разделение отсутствует. Код инициализации пишется бессистемно и смешивается с логикой времени выполнения.

Типичный пример:

```
public Service getService() {
  if (service == null)
    service = new MyServiceImpl(...);  // Инициализация по умолчанию,
                                       // подходящая для большинства случаев?
  return service;
}
```

Идиома ОТЛОЖЕННОЙ ИНИЦИАЛИЗАЦИИ обладает определенными достоинствами. Приложение не тратит времени на конструирование объекта до момента его фактического использования, а это может ускорить процесс инициализации. Кроме того, мы следим за тем, чтобы функция никогда не возвращала `null`.

Однако в программе появляется жестко закодированная зависимость от класса `MyServiceImpl` и всего, что необходимо для его конструктора (который я не привел). Программа не компилируется без разрешения этих зависимостей, даже если объект этого типа ни разу не используется во время выполнения!

Проблемы могут возникнуть и при тестировании. Если `MyServiceImpl` представляет собой тяжеловесный объект, нам придется позаботиться о том, чтобы перед вызовом метода в ходе модульного тестирования в поле `service` был сохранен соответствующий ТЕСТОВЫЙ ДУБЛЕР [Mezzaros07] или ФИКТИВНЫЙ ОБЪЕКТ. А поскольку логика конструирования смешана с логикой нормальной обработки, мы должны протестировать все пути выполнения (в частности, проверку `null` и ее блок). Наличие обеих обязанностей означает, что метод выполняет более одной операции, а это указывает на некоторое нарушение принципа единой ответственности.

Но хуже всего другое — мы не знаем, является ли `MyServiceImpl` правильным объектом во всех случаях. Я намекнул на это в комментарии. Почему класс с этим методом должен знать глобальный контекст? Можем ли мы вообще определить, какой объект должен здесь использоваться? И вообще, может ли один тип быть подходящим для всех возможных контекстов?

Конечно, одно вхождение ОТЛОЖЕННОЙ ИНИЦИАЛИЗАЦИИ не создает серьезных проблем. Однако в приложениях идиомы инициализации обычно встречаются во множество экземпляров. Таким образом, глобальная стратегия инициализации (если она здесь вообще присутствует) распределяется по всему приложению, с минимальной модульностью и значительным дублированием кода.

Если вы действительно стремитесь к созданию хорошо структурированных, надежных систем, никогда не допускайте, чтобы удобные идиомы вели к нарушению модульности. Процесс конструирования объектов и установления связей не является исключением. Этот процесс должен быть отделен от нормальной логики времени выполнения, а вы должны позаботиться о выработке глобальной, последовательной стратегии разрешения основных зависимостей.

Отделение main

Один из способов отделения конструирования от использования заключается в простом перемещении всех аспектов конструирования в `main` (или модули, вызываемые из `main`). Далее весь остальной код системы пишется в предположении, что все объекты были успешно сконструированы и правильно связаны друг с другом (рис. 11.1).

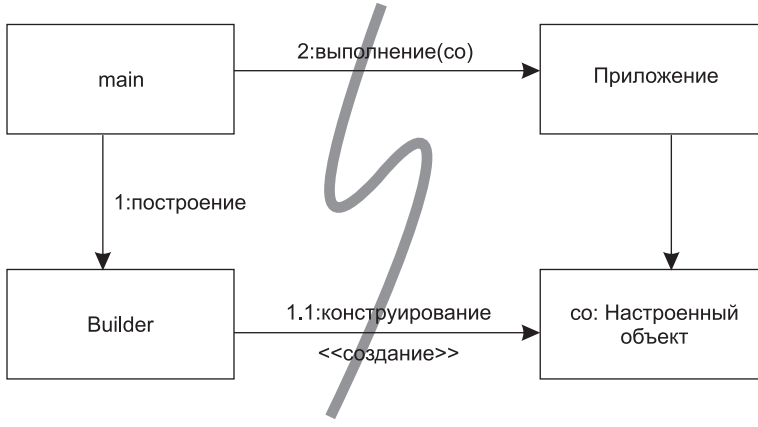


Рис. 11.1. Изоляция конструирования в main

На рисунке хорошо видна последовательность передачи управления. Функция `main` строит объекты, необходимые для системы, а затем передает их приложению, которое их просто использует. Обратите внимание на направление стрелок зависимостей, пересекающих границу между `main` и приложением. Все стрелки указывают в одном направлении — от `main`. Это означает, что приложение ничего не знает о `main` или о процессе конструирования. Оно просто ожидает, что все объекты были построены правильно.

Фабрики

Конечно, в некоторых ситуациях момент создания объекта должен определяться приложением. Например, в системе обработки заказов приложение должно создать экземпляры товаров `LineItem` для включения их в объект заказа `Order`. В этом случае можно воспользоваться паттерном АБСТРАКТНАЯ ФАБРИКА [GOF], чтобы приложение могло само выбрать момент для создания `LineItem`, но при этом подробности конструирования были отделены от кода приложения (рис. 11.2).

И снова обратите внимание на то, что все стрелки зависимостей ведут от `main` к приложению `OrderProcessing`. Это означает, что приложение изолировано от подробностей построения `LineItem`. Вся информация хранится в реализации

LineItemFactoryImplementation, находящейся на стороне main. Тем не менее приложение полностью управляет моментом создания экземпляров LineItem и даже может передать аргументы конструктора, специфические для конкретного приложения.

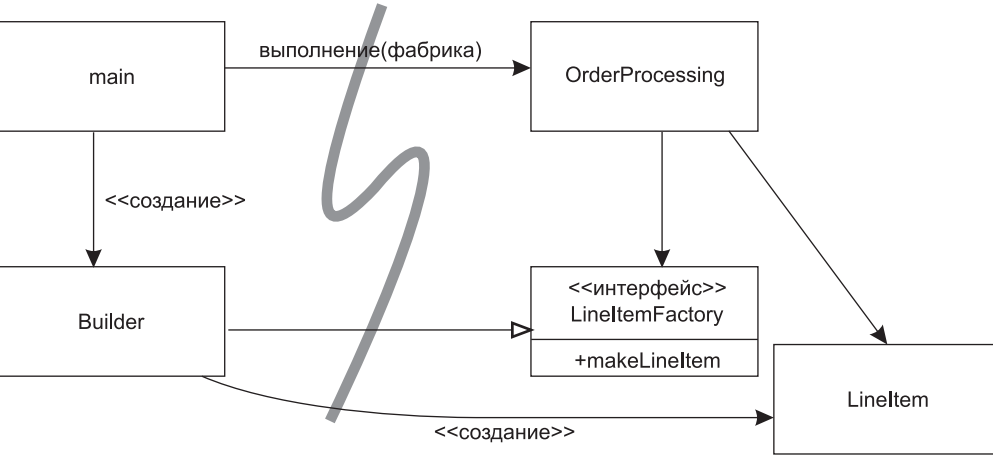


Рис. 11.2. Отделение конструирования с применением фабрики

Внедрение зависимостей

Внедрение зависимостей (DI, Dependency Injection) — мощный механизм отделения конструирования от использования, практическое применение обращения контроля (IoC, Inversion of Control) в области управления зависимостями[1]. Обращение контроля перемещает вторичные обязанности объекта в другие объекты, созданные специально для этой цели, тем самым способствуя соблюдению принципа единой ответственности. В контексте управления зависимостями объект не должен брать на себя ответственность за создание экземпляров зависимостей. Вместо этого он передает эту обязанность другому «уполномоченному» механизму. Так как инициализация является глобальной областью ответственности, этим уполномоченным механизмом обычно является либо функция main, либо специализированный *контейнер*.

Примером «частичной» реализации внедрения зависимостей является запрос JNDI, когда объект обращается к серверу каталоговой информации с запросом на предоставление «сервиса» с заданным именем:

```
MyService myService = (MyService)(jndiContext.lookup("NameOfMyService"));
```

Вызывающий объект не управляет тем, какой именно объект будет возвращен (конечно, при условии, что этот объект реализует положенный интерфейс), но при этом происходит активное разрешение зависимости.

[1] Например, см. [Fowler].

Истинное внедрение зависимостей идет еще на один шаг вперед. Класс не предпринимает непосредственных действий по разрешению своих зависимостей; он остается абсолютно пассивным. Вместо этого он предоставляет set-методы и/или аргументы конструктора, используемые для внедрения зависимостей. В процессе конструирования контейнер DI создает экземпляры необходимых объектов (обычно по требованию) и использует аргументы конструктора или set-методы для скрепления зависимостей. Фактически используемые зависимые объекты задаются в конфигурационном файле или на программном уровне в специализированном конструирующем модуле.

Самый известный DI-контейнер для Java присутствует в Spring Framework[1]. Подключаемые объекты перечисляются в конфигурационном файле XML, после чего конкретный объект запрашивается по имени в коде Java. Пример будет рассмотрен ниже.

Но как же преимущества ОТЛОЖЕННОЙ ИНИЦИАЛИЗАЦИИ? Эта идиома иногда бывает полезной и при внедрении зависимостей. Во-первых, большинство DI-контейнеров не конструирует объекты до того момента, когда это станет необходимо. Во-вторых, многие из этих контейнеров предоставляют механизмы использования фабрик или конструирования посредников (proxies), которые могут использоваться для ОТЛОЖЕННОЙ ИНИЦИАЛИЗАЦИИ и других аналогичных оптимизаций[2].

Масштабирование

Города вырастают из городков, которые, в свою очередь, появляются на месте деревень. Дороги сначала узки и едва заметны, но со временем они расширяются и покрываются камнем. Мелкие строения и пустые места заполняются более крупными зданиями, часть из которых в конечном итоге будет заменена небоскребами.

На первых порах в городе полностью отсутствует инфраструктура: водопровод, электричество, канализация и (о ужас!) Интернет. Все эти возможности добавляются позднее, с ростом населения и плотности застройки.

Рост не обходится без проблем. Сколько раз вам приходилось едва ползти в потоке машин вдоль проекта по «расширению дороги», когда вы спрашивали себя: «Почему нельзя было сразу построить дорогу достаточной ширины?!»

Но иначе и быть не могло. Кто сможет объяснить затраты на строительство шестиполосной магистрали в середине маленького городка, которому предрекают расширение? Да и кто бы захотел иметь такую дорогу в своем городе?

---

[1] См. [Spring] и описание Spring.NET.

[2] Не забывайте, что отложенная инициализация — всего лишь разновидность оптимизации... и возможно, преждевременная!

Возможность построить «правильную систему с первого раза» — миф. Вместо этого мы сегодня реализуем текущие потребности, а завтра перерабатываем и расширяем систему для реализации новых потребностей. В этом заключается суть итеративной, пошаговой гибкой разработки. Разработка через тестирование, рефакторинг и полученный в результате их применения чистый код обеспечивают работу этой схемы на уровне кода.

А как же системный уровень? Разве архитектура системы не требует предварительного планирования? Не может же она последовательно расти от простого к сложному?

> В этом проявляется важнейшее отличие программных систем от физических. Архитектура программных систем **может** развиваться последовательно, если обеспечить правильное разделение ответственности.

Как вы вскоре убедитесь, нематериальная природа программных систем делает это возможным. Но давайте начнем с контрпримера архитектуры, в которой нормальное разделение ответственности отсутствует.

Исходные архитектуры EJB1 и EJB2 не обеспечивали должного разделения областей ответственности и поэтому создавали лишние барьеры для естественного роста. Возьмем хотя бы *компонент-сущность* (Entity Bean) для постоянного (persistent) класса. Компонентом-сущностью называется представление реляционных данных (иначе говоря, записи таблицы) в памяти.

Для начала необходимо определить локальный (внутрипроцессный) или удаленный (на отдельной JVM) интерфейс, который будет использоваться клиентами. Возможный локальный интерфейс представлен в листинге 11.1.

Листинг 11.1. Локальный интерфейс EJB2 для EJB Bank

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public interface BankLocal extends java.ejb.EJBLocalObject {
  String getStreetAddr1() throws EJBException;
  String getStreetAddr2() throws EJBException;
  String getCity() throws EJBException;
  String getState() throws EJBException;
  String getZipCode() throws EJBException;
  void setStreetAddr1(String street1) throws EJBException;
  void setStreetAddr2(String street2) throws EJBException;
  void setCity(String city) throws EJBException;
  void setState(String state) throws EJBException;
  void setZipCode(String zip) throws EJBException;
  Collection getAccounts() throws EJBException;
  void setAccounts(Collection accounts) throws EJBException;
  void addAccount(AccountDTO accountDTO) throws EJBException;
}
```

В интерфейс включены некоторые атрибуты адреса Bank, а также коллекция счетов, принадлежащих банку; данные каждого счета представляются отдельным EJB Account. В листинге 11.2 приведен соответствующий класс реализации компонента Bank.

Листинг 11.2. Соответствующая реализация компонента-сущности EJB2

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public abstract class Bank implements javax.ejb.EntityBean {
  // Бизнес-логика...
  public abstract String getStreetAddr1();
  public abstract String getStreetAddr2();
  public abstract String getCity();
  public abstract String getState();
  public abstract String getZipCode();
  public abstract void setStreetAddr1(String street1);
  public abstract void setStreetAddr2(String street2);
  public abstract void setCity(String city);
  public abstract void setState(String state);
  public abstract void setZipCode(String zip);
  public abstract Collection getAccounts();
  public abstract void setAccounts(Collection accounts);
  public void addAccount(AccountDTO accountDTO) {
    InitialContext context = new InitialContext();
    AccountHomeLocal accountHome = context.lookup("AccountHomeLocal");
    AccountLocal account = accountHome.create(accountDTO);
    Collection accounts = getAccounts();
    accounts.add(account);
  }
  // Логика контейнера EJB
  public abstract void setId(Integer id);
  public abstract Integer getId();
  public Integer ejbCreate(Integer id) { ... }
  public void ejbPostCreate(Integer id) { ... }
  // Остальные методы должны быть реализованы, но обычно остаются пустыми:
  public void setEntityContext(EntityContext ctx) {}
  public void unsetEntityContext() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void ejbLoad() {}
  public void ejbStore() {}
  public void ejbRemove() {}
}
```

В листинге не приведен ни соответствующий интерфейс LocalHome (по сути — фабрика, используемая для создания объектов), ни один из возможных методов поиска Bank, которые вы можете добавить.

Наконец, вы должны написать один или несколько дескрипторов в формате XML, которые определяют подробности соответствия между объектом

и реляционными данными, желаемое транзакционное поведение, ограничения безопасности и т. д.

Бизнес-логика тесно привязана к «контейнеру» приложения EJB2. Вы должны субклассировать контейнерные типы, а также предоставить многие методы жизненного цикла, необходимые для контейнера.

Привязка к тяжеловесному контейнеру затрудняет изолированное модульное тестирование. Приходится либо имитировать контейнер, что непросто, либо тратить много времени на развертывание EJB и тестов на реальном сервере. Повторное использование за пределами архитектуры EJB2 практически невозможно из-за жесткой привязки.

Наконец, такое решение противоречит принципам объектно-ориентированного программирования. Один компонент не может наследовать от другого компонента. Обратите внимание на логику добавления нового счета. В EJB2 компоненты часто определяют «объекты передачи данных» (DTO), которые фактически представляют собой «структуры без поведения». Обычно это приводит к появлению избыточных типов, содержащих по сути одинаковые данные, и необходимости использования стереотипного кода для копирования данных между объектами.

Поперечные области ответственности

В некоторых областях архитектура EJB2 приближается к полноценному разделению ответственности. Например, желательное поведение в области транзакционности, безопасности и сохранения объектов объявляется в дескрипторах независимо от исходного кода.

Такие области, как сохранение объектов, выходят за рамки естественных границ объектов предметной области. Например, все объекты обычно сохраняются по одной стратегии, с использованием определенной СУБД[1] вместо неструктурированных файлов, с определенной схемой выбора имен таблиц и столбцов, единой транзакционной семантикой и т. д.

Теоретически возможен модульный, инкапсулированный подход к определению стратегии сохранения объектов. Однако на практике вам приходится повторять по сути одинаковый код, реализующий стратегию сохранения, во многих объектах. Для подобных областей используется термин «*поперечные области ответственности*». При этом инфраструктура сохранения может быть модульной, и логика предметной области, рассматриваемая в изоляции, тоже может быть модульной. Проблемы возникают в точках пересечения этих областей. Можно сказать, что подход, использованный в архитектуре EJB по отношению к сохранению объектов, безопасности и транзакциям, предвосхитил *аспектно-ориентированное программирование* (АОП[2]), которое представляет собой универсальный подход к восстановлению модульности для поперечных областей ответственности.

---

[1] Система управления базами данных.

[2] За общей информацией об аспектах обращайтесь к [AOSD], а за конкретной информацией об AspectJ — к [AspectJ] и [Colyer].

В АОП специальные модульные конструкции, называемые *аспектами*, определяют, в каких точках системы поведение должно меняться некоторым последовательным образом в соответствии с потребностями определенной области ответственности. Определение осуществляется на уровне декларативного или программного механизма.

В примере с сохранением объектов вы объявляете, какие объекты, атрибуты и т. д. должны сохраняться, а затем делегируете задачи сохранения своей инфраструктуре сохранения. Изменения в поведении вносятся инфраструктурой АОП без вмешательства в целевой код[1]. Рассмотрим три аспекта (или «аспекто-подобных» механизма) в Java.

Посредники

Посредники (proxies) хорошо подходят для простых ситуаций — например, для создания «оберток» для вызова методов отдельных объектов или классов. Тем не менее динамические посредники, содержащиеся в JDK, работают только с интерфейсами. Чтобы создать посредника для класса, приходится использовать библиотеки для выполнения манипуляций с байт-кодом — такие, как CGLIB, ASM или Javassist[2].

В листинге 11.3 приведена заготовка посредника JDK, обеспечивающего поддержку сохранения объектов в нашем приложении `Bank` (представлены только методы чтения/записи списка счетов).

Листинг 11.3. Пример посредника JDK

```
// Bank.java (подавление имен пакетов...)
import java.utils.*;

// Абстрактное представление банка.
public interface Bank {
  Collection<Account> getAccounts();
  void setAccounts(Collection<Account> accounts);
}

// BankImpl.java
import java.utils.*;

// POJO-объект ("Plain Old Java Object"), реализующий абстракцию.
public class BankImpl implements Bank {
  private List<Account> accounts;
  public Collection<Account> getAccounts() {
    return accounts;
  }
  public void setAccounts(Collection<Account> accounts) {
```

[1] То есть без необходимости ручного редактирования целевого кода.

[2] См. [CGLIB], [ASM] и [Javassist].

```
    this.accounts = new ArrayList<Account>();
    for (Account account: accounts) {
      this.accounts.add(account);
    }
  }
}

// BankProxyHandler.java
import java.lang.reflect.*;
import java.util.*;

// Реализация InvocationHandler, необходимая для API посредника.
public class BankProxyHandler implements InvocationHandler {
  private Bank bank;

  public BankHandler (Bank bank) {
    this.bank = bank;
  }

  // Метод, определенный в InvocationHandler
  public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable {
    String methodName = method.getName();
    if (methodName.equals("getAccounts")) {
      bank.setAccounts(getAccountsFromDatabase());
      return bank.getAccounts();
    } else if (methodName.equals("setAccounts")) {
      bank.setAccounts((Collection<Account>) args[0]);
      setAccountsToDatabase(bank.getAccounts());
      return null;
    } else {
      ...
    }
  }

  // Подробности:
  protected Collection<Account> getAccountsFromDatabase() { ... }
  protected void setAccountsToDatabase(Collection<Account> accounts) { ... }
}

// В другом месте...
Bank bank = (Bank) Proxy.newProxyInstance(
  Bank.class.getClassLoader(),
  new Class[] { Bank.class },
  new BankProxyHandler(new BankImpl()));
```

Мы определили интерфейс `Bank`, который будет инкапсулироваться посредником, и POJO-объект («Plain Old Java Object», то есть «обычный Java-объект») `BankImpl`, реализующий бизнес-логику. (Вскоре мы вернемся к теме POJO-объектов).

Для работы посредника необходим объект `InvocationHandler`, который вызывается для реализации всех вызовов методов `Bank`, обращенных к посреднику. Наша реа-

лизация `BankProxyHandler` использует механизм рефлексии Java для отображения вызовов обобщенных методов на соответствующие методы `BankImpl`.

Код получается *весьма* объемистым и относительно сложным, даже в этом простом случае[1]. Не меньше проблем создает и использование библиотек для манипуляций с байт-кодом. Объем и сложность кода — два основных недостатка посредников. Эти два фактора усложняют создание чистого кода! Кроме того, у посредников не существует механизма определения «точек интереса» общесистемного уровня, необходимых для полноценного АОП-решения[2].

АОП-инфраструктуры на «чистом» Java

К счастью, большая часть шаблонного кода посредников может автоматически обрабатываться вспомогательными средствами. Посредники используются во внутренней реализации нескольких инфраструктур Java — например, Spring AOP и JBoss AOP — для реализации аспектов непосредственно на уровне Java[3].

В Spring бизнес-логика записывается в форме POJO-объектов. Такие объекты полностью сосредоточены на своей предметной области. Они не имеют зависимостей во внешних инфраструктурах (или любых других областях); соответственно им присуща большая концептуальная простота и удобство тестирования. Благодаря относительной простоте вам будет проще обеспечить правильную реализацию соответствующих пожеланий пользователей, а также сопровождение и эволюцию кода при появлении новых пожеланий.

Вся необходимая инфраструктура приложения, включая поперечные области ответственности (сохранение объектов, транзакции, безопасность, кэширование, преодоление отказов и т. д.), определяется при помощи декларативных конфигурационных файлов или API. Во многих случаях вы фактически определяете аспекты библиотек Spring или JBoss, а инфраструктура берет на себя всю механику использования посредников Java или библиотек байт-кода в режиме, прозрачном для пользователя. Объявления управляют контейнером внедрения зависимостей (DI), который создает экземпляры основных объектов и связывает их по мере необходимости.

В листинге 11.4 приведен типичный фрагмент конфигурационного файла Spring V2.5 *app.xml*[4].

---

[1] Более подробные примеры API посредников и его использования можно найти, например, в [Goetz].

[2] Методологию АОП иногда путают с приемами, используемыми для ее реализации например перехватом методов и «инкапсуляцией» посредников. Подлинная ценность АОП-системы заключается в способности модульного, компактного определения системного поведения.

[3] См. [Spring] и [JBoss]. «Непосредственно на уровне Java» в данном случае означает «без применения AspectJ».

[4] По материалам http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25.

Листинг 11.4. Конфигурационный файл Spring 2.X

```
<beans>
  ...
  <bean id="appDataSource"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="com.mysql.jdbc.Driver"
    p:url="jdbc:mysql://localhost:3306/mydb"
    p:username="me"/>

  <bean id="bankDataAccessObject"
    class="com.example.banking.persistence.BankDataAccessObject"
    p:dataSource-ref="appDataSource"/>

  <bean id="bank"
    class="com.example.banking.model.Bank"
    p:dataAccessObject-ref="bankDataAccessObject"/>
  ...
</beans>
```

Каждый компонент напоминает одну из частей русской «матрешки»: объект предметной области `Bank` «упаковывается» в объект доступа к данным DAO (Data Accessor Object), который, в свою очередь, упаковывается в объект источника данных JDBC (рис. 11.3).

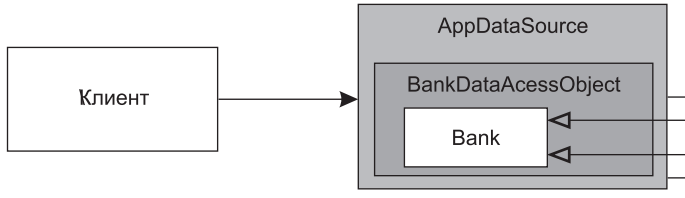


Рис. 11.3. «Матрешка» из декораторов

Клиент полагает, что он вызывает метод `getAccounts()` объекта `Bank`, но в действительности он взаимодействует с внешним объектом из набора вложенных ДЕКОРАТОРОВ [GOF], расширяющих базовое поведение POJO-объекта `Bank`. Мы могли бы добавить другие декораторы для транзакций, кэширования и т. д.

Чтобы запросить у DI-контейнера объекты верхнего уровня, заданные в файле XML, достаточно включить в приложение несколько строк:

```
XmlBeanFactory bf =
  new XmlBeanFactory(new ClassPathResource("app.xml", getClass()));
Bank bank = (Bank) bf.getBean("bank");
```

Так как объем кода, специфического для Spring, минимален, приложение почти полностью изолировано от Spring. Тем самым устраняются все проблемы жесткой привязки, характерные для таких систем, как EJB2.

Хотя код XML занимает много места и плохо читается[1], определяемая в этих конфигурационных файлах «политика» все же проще сложной логики посредников и аспектов, скрытой от наших глаз и создаваемой автоматически. Архитектура выглядит настолько заманчиво, что инфраструктуры вроде Spring привели к полной переработке стандарта EJB для версии 3. EJB3 в значительной мере следует характерной для Spring модели декларативной поддержки поперечных областей ответственности с использованием конфигурационных файлов XML и/или аннотаций Java 5.

В листинге 11.5 приведен объект Bank, переписанный для EJB3[2].

Листинг 11.5. Компонент Bank для EBJ3

```
package com.example.banking.model;
import javax.persistence.*;
import java.util.ArrayList;
import java.util.Collection;

@Entity
@Table(name = "BANKS")
public class Bank implements java.io.Serializable {
   @Id @GeneratedValue(strategy=GenerationType.AUTO)
   private int id;

   @Embeddable // Объект "встраивается" в запись базы данных Bank
   public class Address {
      protected String streetAddr1;
      protected String streetAddr2;
      protected String city;
      protected String state;
      protected String zipCode;
   }

   @Embedded
   private Address address;

   @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,
              mappedBy="bank")
   private Collection<Account> accounts = new ArrayList<Account>();

   public int getId() {
      return id;
   }
```

[1] Приведенный пример можно упростить — существуют специальные механизмы, использующие правила конфигурации и аннотации Java 5 для сокращения объема явно определяемой «связующей» логики.

[2] По материалам http://www.onjava.com/pub/a/onjava/2006/05/17/standardizing-with-ejb3-java-persistence-api.html.

```
    public void setId(int id) {
        this.id = id;
    }

    public void addAccount(Account account) {
        account.setBank(this);
        accounts.add(account);
    }

    public Collection<Account> getAccounts() {
        return accounts;
    }

    public void setAccounts(Collection<Account> accounts) {
        this.accounts = accounts;
    }
}
```

Этот вариант кода намного чище исходного кода EJB2. Некоторые подробности о сущностях все еще присутствуют в аннотациях. Тем не менее, поскольку эта информация не выходит за пределы аннотаций, код остается чистым, понятным, а следовательно, простым в тестировании, сопровождении и т. д.

Часть информации о сохранении объектов, содержащейся в аннотациях, можно при желании переместить в дескрипторы XML, оставив действительно чистый POJO-объект. Если детали сохранения объектов изменяются относительно редко, многие группы отдадут предпочтение аннотациям, но с гораздо меньшими отрицательными последствиями по сравнению с EJB2.

Аспекты AspectJ

Наконец, самым полнофункциональным инструментом для разделения областей ответственности посредством использования аспектов является язык AspectJ[1] — расширение Java, предоставляющее «полноценную» поддержку аспектов как модульных конструкций. Чистых Java-решений на базе Spring и JBoss достаточно для 80–90% ситуаций, в которых применяются аспекты. Тем не менее AspectJ предоставляет очень мощный и разносторонний инструментарий для реализации разделения ответственности. Недостатком AspectJ является необходимость освоения нескольких новых инструментов, а также изучения новых языковых конструкций и идиом.

Эти проблемы отчасти компенсируются появившейся недавно «аннотационной» формой AspectJ, в которой аннотации Java 5 используются для определения аспектов в «чистом» коде Java. Кроме того, Spring Framework также содержит ряд функций, существенно упрощающих внедрение аспектов на базе аннотаций в рабочих группах с ограниченным опытом применения AspectJ.

[1] См. [AspectJ] и [Colyer].

Полное описание AspectJ выходит за рамки книги. За дополнительной информацией обращайтесь к [AspectJ], [Colyer] и [Spring].

Испытание системной архитектуры

Трудно переоценить потенциал разделения ответственности посредством аспектных решений. Если вы можете написать логику предметной области своего приложения в виде POJO-объектов, отделенных от любых архитектурных областей ответственности на кодовом уровне, то перед вами открывается возможность проведения полноценных испытаний вашей архитектуры. Вы сможете развивать ее от простого к сложному, как потребует ситуация, подбирая новые технологии по мере надобности. Не обязательно создавать Большой Изначальный Проект (BDUF, Big Design Up Front)[1]. Более того, это даже вредно, потому что BDUF снижает возможность адаптации к изменениям из-за нашего психологического нежелания расставаться с результатами уже затраченных усилий; кроме того, изначально принятые решения влияют на наши последующие представления об архитектуре.

Архитекторы, занимающиеся строительством зданий, вынуждены работать по принципу BDUF, потому что они не могут вносить радикальные архитектурные изменения в наполовину возведенное физическое строение[2]. Программные продукты тоже обладают собственной физикой[3], но радикальные изменения в них могут оказаться экономически оправданными — при условии, что в программном проекте эффективно реализовано разделение ответственности.

Это означает, что мы можем начать программный проект с «простой до наивности», но лишенной жестких привязок архитектуры, быстро реализовать пожелания пользователей, а затем добавлять новую инфраструктуру по мере масштабирования. Некоторые из крупнейших мировых сайтов достигли высочайших показателей доступности и производительности, с применением сложного кэширования данных, безопасности, виртуализации и т. д., и все это делается эффективно и гибко — и только потому, что на каждом уровне абстракции их архитектура оставалась простой и обладала минимальными привязками.

Конечно, это не означает, что за проект нужно браться по принципу «как-нибудь по ходу разберемся». Вы уже в определенной степени представляете себе общий масштаб, цели и график проекта, а также общую структуру итоговой системы. Однако при этом необходимо сохранить возможность «смены курса» в соответствии с изменяющимися обстоятельствами.

---

[1] Не путайте с полезной практикой упреждающего проектирования. BDUF — привычка проектировать заранее *все без исключения*, до написания какого-либо кода реализации.

[2] Впрочем, даже после начала строительства идут серьезные итеративные исследования и обсуждения подробностей.

[3] Выражение «физика программного продукта» впервые было использовано в [Kolence].

Ранняя архитектура EJB была всего лишь одним из многих API, которые отличались излишней сложностью, нарушавшей принцип разделения ответственности. Впрочем, даже хорошо спроектированный API может оказаться «перебором» в конкретной ситуации, если его применение не объясняется реальной необходимостью. Хороший API должен исчезать из вида большую часть времени, чтобы большая часть творческих усилий группы расходовалась на реализацию пожеланий пользователей. В противном случае архитектурные ограничения помешают оптимальной реализации интересов клиента.

Подведем итог.

> Оптимальная архитектура системы состоит из модульных областей ответственности, каждая из которых реализуется на базе POJO-объектов. Области интегрируются между собой при помощи аспектов или аналогичных средств, минимальным образом вмешивающихся в их работу. Такая архитектура может строиться на базе методологии разработки через тестирование, как и программный код.

Оптимизация принятия решений

Модульность и разделение ответственности позволяют децентрализовать управление и принятие решений. В достаточно крупной системе, будь то город или программный проект, один человек не может принять все необходимые решения.

Как известно, ответственные решения лучше всего поручить самому квалифицированному. Однако мы часто забываем, что принятие решений лучше всего откладывать до последнего момента. Дело не в лени или безответственности; просто это позволяет принять информированное решение с максимумом возможной информации. Преждевременное решение принимается на базе неполной информации. Принимая решение слишком рано, мы лишаемся всего полезного, что происходит на более поздних стадиях: обратной связи от клиентов, возможности поразмышлять над текущим состоянием проекта и опыта применения решений из области реализации.

> Гибкость POJO-системы с модульными областями ответственности позволяет принимать оптимальные, своевременные решения на базе новейшей информации. Кроме того, она способствует снижению сложности таких решений.

Применяйте стандарты разумно, когда они приносят очевидную пользу

Строительство кажется настоящим чудом из-за темпов, которым возводятся новые здания (даже в разгар зимы), и из-за необычных архитектурных дизайнов, ставших возможными благодаря современным технологиям. Строительство стало

развитой областью промышленности с высокой оптимизацией частей, методов и стандартов, сформированных под давлением времени.

Многие группы использовали архитектуру EJB2 только потому, что она считалась стандартом, даже если в их проектах хватило бы более легких и прямолинейных решений. Я видел группы, которые теряли голову от *разрекламированных* стандартов и забывали о своей главной задаче: реализовывать интересы клиента.

> Стандарты упрощают повторное использование идей и компонентов, привлечение людей с необходимым опытом, воплощение удачных идей и связывание компонентов. Тем не менее, процесс создания стандарта иногда занимает слишком много времени (а отрасль не стоит на месте), в результате чего стандарты теряют связь с реальными потребностями тех людей, которым они должны служить.

Системам необходимы предметно-ориентированные языки

В области строительства, как и в большинстве технических областей, сформировался богатый язык со своим словарем, идиомами и паттернами[1], позволяющими четко и лаконично передать важную информацию. В области разработки программного обеспечения в последнее время снова возобновился интерес к предметно-ориентированным языкам[2] (DSL, Domain-Specic Languages) — отдельным маленьким сценарным языкам или API стандартных языков, код которых читается как структурированная форма текста, написанного экспертом в данной предметной области.

Хороший предметно-ориентированный язык сводит к минимуму «коммуникационный разрыв» между концепцией предметной области и кодом, реализующим эту концепцию — по аналогии с тем, как гибкие методологии оптимизируют обмен информацией между группой и ключевыми участниками проекта. Реализация логики предметной области на языке, используемом экспертом в этой области, снижает риск неверного представления предметной области в коде.

Предметно-ориентированные языки, когда они используются эффективно, поднимают уровень абстракции над программными идиомами и паттернами проектирования. Они позволяют разработчику выразить свои намерения на соответствующем уровне абстракции.

> Предметно-ориентированные языки позволяют выразить в форме POJO-объектов все уровни абстракции и все предметные области приложения, от высокоуровневых политик до низкоуровневых технических подробностей.

---

[1] Работа [Alexander] оказала особенно заметное влияние на сообщество разработчиков ПО.

[2] Например, см. [DSL]. [JMock] — хороший пример Java API, создавшего свой предметно-ориентированный язык.

Заключение

Чистым должен быть не только код, но и архитектура системы. Агрессивная, «всепроникающая» архитектура скрывает логику предметной области и снижает гибкость. Первое приводит к снижению качества: ошибкам проще спрятаться в коде, а разработчику труднее реализовать пожелания пользователей. Второе оборачивается снижением производительности, а также потерей всех преимуществ TDD.

Намерения разработчика должны быть четко выражены на всех уровнях абстракции. Это произойдет только в том случае, если он создает POJO-объекты, и использует аспекты (или другие аналогичные механизмы) для неагрессивного воплощения других сторон реализации.

Независимо от того, проектируете ли вы целую систему или ее отдельные модули, помните: *используйте самое простое решение из всех возможных*.

Литература

[Alexander]: Christopher Alexander, A Timeless Way of Building, Oxford University Press, New York, 1979.

[AOSD]: Aspect-Oriented Software Development port, http://aosd.net

[ASM]: ASM Home Page, http://asm.objectweb.org

[AspectJ]: http://eclipse.org/aspectj

[CGLIB]: Code Generation Library, http://cglib.sourceforge.net

[Colyer]: Adrian Colyer, Andy Clement, George Hurley, Mathew Webster, Eclipse AspectJ, Person Education, Inc., Upper Saddle River, NJ, 2005.

[DSL]: Domain-specific programming language, http://en.wikipedia.org/wiki/Domain-specific_programming_language

[Fowler]: Inversion of Control Containers and the Dependency Injection pattern, http://martinfowler.com/articles/injection.html

[Goetz]: Brian Goetz, Java Theory and Practice: Decorating with Dynamic Proxies, http://www.ibm.com/developerworks/java/library/j-jtp08305.html

[Javassist]: Javassist Home Page, http://www.csg.is.titech.ac.jp/~chiba/javassist

[JBoss]: JBoss Home Page, http://jboss.org

[JMock]: JMock — A Lightweight Mock Object Library for Java, http://jmock.org

[Kolence]: Kenneth W. Kolence, Software physics and computer performance measurements, Proceedings of the ACM annual conference—Volume 2, Boston, Massachusetts, pp. 1024–1040, 1972.

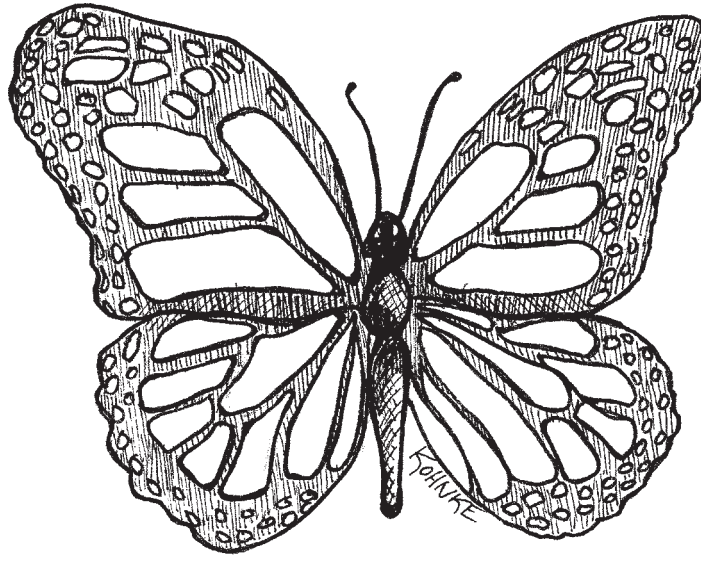
[Spring]: The Spring Framework, http://www.springframework.org

[Mezzaros07]: XUnit Patterns, Gerard Mezzaros, Addison-Wesley, 2007.

[GOF]: Design Patterns: Elements of Reusable Object Oriented Software, Gamma et al., Addison-Wesley, 1996.

12 Формирование архитектуры

Джефф Лангр



Четыре правила

Разве не хотелось бы вам знать четыре простых правила, выполнение которых помогало бы повысить качество проектирования? Четыре правила, помогающих составить представление о важнейших особенностях структуры и архитектуры кода, упрощающих применение таких принципов, как SRP (принцип единой ответственности) и DIP (принцип обращения зависимостей)? Четыре правила, способствующих формированию хороших архитектур?

Многие полагают, что четыре правила *простой архитектуры* [XPE] Кента Бека оказывают значительную помощь в проектировании программных продуктов.

Согласно Кенту, архитектура может считаться «простой», если она:

- обеспечивает прохождение всех тестов,
- не содержит дублирующегося кода,
- выражает намерения программиста,
- использует минимальное количество классов и методов.

Правила приведены в порядке их важности.

Правило № 1: выполнение всех тестов

Прежде всего система должна делать то, что задумано ее проектировщиком. Система может быть отлично спланирована «на бумаге», но если не существует простого способа убедиться в том, что она действительно решает свои задачи, то результат выглядит сомнительно.

Система, тщательно протестированная и прошедшая все тесты, *контролируема*. На первый взгляд утверждение кажется очевидным, но это весьма важно. Невозможно проверить работу системы, которая не является контролируемой, а непроверенные системы не должны запускаться в эксплуатацию.

К счастью, стремление к контролируемости системы ведет к архитектуре с компактными узкоспециализированными классами. Все просто: классы, соответствующие принципу SRP, проще тестировать. Чем больше тестов мы напишем, тем дальше продвинемся к простоте тестирования. Таким образом, обеспечение полной контролируемости системы помогает повысить качество проектирования.

Жесткая привязка усложняет написание тестов. Таким образом, чем больше тестов мы пишем, тем интенсивнее используем такие принципы, как DIP, и такие инструменты, как внедрение зависимостей, интерфейсы и абстракции, для минимизации привязок.

Как ни удивительно, выполнение простого и очевидного правила, гласящего, что для системы необходимо написать тесты и постоянно выполнять их, влияет на соответствие системы важнейшим критериям объектно-ориентированного программирования: устранению жестких привязок и повышению связности. Написание тестов улучшает архитектуру системы.

Правила № 2–4: переработка кода

Когда у вас появился полный набор тестов, можно заняться чисткой кода и классов. Для этого код подвергается последовательной переработке (рефакторингу). Мы добавляем несколько строк кода, делаем паузу и анализируем новую архитектуру. Не ухудшилась ли она по сравнению с предыдущим вариантом? Если ухудшилась, то мы чистим код и тестируем его, чтобы убедиться, что в нем ничего не испорчено. Наличие тестов избавляет от опасений, что чистка кода нарушит его работу!

В фазе переработки применяется абсолютно все, что вы знаете о качественном проектировании программных продуктов. В ход идут любые приемы: повышение связности, устранение жестких привязок, разделение ответственности, изоляция системных областей ответственности, сокращение объема функций и классов, выбор более содержательных имен и т. д. Также применяются три критерия простой архитектуры: устранение дубликатов, обеспечение выразительности и минимизация количества классов и методов.

Отсутствие дублирования

Дублирование — главный враг хорошо спроектированной системы. Его последствия — лишняя работа, лишний риск и лишняя избыточная сложность. Дублирование проявляется во многих формах. Конечно, точное совпадение строк кода свидетельствует о дублировании. Похожие строки часто удается «причесать» так, чтобы сходство стало еще более очевидным; это упростит рефакторинг. Кроме того, дублирование может существовать и в других формах — таких, как дублирование реализации. Например, класс коллекции может содержать следующие методы:

```
int size() {}
boolean isEmpty() {}
```

Методы могут иметь разные реализации. Допустим, метод `isEmpty` может использовать логический флаг, а `size` — счетчик элементов. Однако мы можем устранить дублирование, связав `isEmpty` с определением `size`:

```
boolean isEmpty() {
   return 0 == size();
}
```

Чтобы создать чистую систему, необходимо сознательно стремиться к устранению дубликатов, пусть даже всего в нескольких строках кода. Для примера рассмотрим следующий код:

```
  public void scaleToOneDimension(
       float desiredDimension, float imageDimension) {
    if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
       return;
    float scalingFactor = desiredDimension / imageDimension;
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);

    RenderedOp newImage = ImageUtilities.getScaledImage(
       image, scalingFactor, scalingFactor);
    image.dispose();
    System.gc();
    image = newImage;
 }
 public synchronized void rotate(int degrees) {
    RenderedOp newImage = ImageUtilities.getRotatedImage(
       image, degrees);
```

```
    image.dispose();
    System.gc();
    image = newImage;
  }
```

Чтобы обеспечить чистоту системы, следует устранить незначительное дублирование между методами `scaleToOneDimension` и `rotate`:

```
  public void scaleToOneDimension(
      float desiredDimension, float imageDimension) {
    if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
      return;
    float scalingFactor = desiredDimension / imageDimension;
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);
    replaceImage(ImageUtilities.getScaledImage(
      image, scalingFactor, scalingFactor));
  }

  public synchronized void rotate(int degrees) {
    replaceImage(ImageUtilities.getRotatedImage(image, degrees));
  }

  private void replaceImage(RenderedOp newImage) {
    image.dispose();
    System.gc();
    image = newImage;
  }
```

В ходе выделения общности конструкций на этом микроскопическом уровне начинают проявляться нарушения принципа SRP. Таким образом, только что сформированный метод можно переместить в другой класс. Это расширяет видимость метода. Другой участник группы может найти возможность дальнейшего абстрагирования нового метода и его использования в другом контексте. Таким образом, принцип «повторного использования даже в мелочах» может привести к значительному сокращению сложности системы. Понимание того, как обеспечить повторное использование в мелочах, абсолютно необходимо для его обеспечения в большом масштабе.

Паттерн ШАБЛОННЫЙ МЕТОД [GOF] относится к числу стандартных приемов устранения высокоуровневого дублирования. Пример:

```
public class VacationPolicy {
   public void accrueUSDivisionVacation() {
      // Код вычисления продолжительности отпуска
      // по количеству отработанных часов
      // ...
      // Код проверки минимальной продолжительности отпуска
      // по стандартам США
      // ...
      // Код внесения отпуска в платежную ведомость
      // ...
   }
```

```
   public void accrueEUDivisionVacation() {
      // Код вычисления продолжительности отпуска
      // по количеству отработанных часов
      // ...
      // Код проверки минимальной продолжительности отпуска
      // по европейским стандартам
      // ...
      // Код внесения отпуска в платежную ведомость
      // ...
   }
}
```

Код `accrueUSDivisionVacation` и `accrueEuropeanDivisionVacation` в основном совпадает, если не считать проверки минимальной продолжительности. Этот фрагмент алгоритма изменяется в зависимости от типа работника.

Для устранения этого очевидного дублирования можно воспользоваться паттерном ШАБЛОННЫЙ МЕТОД:

```
abstract public class VacationPolicy {
   public void accrueVacation() {
      calculateBaseVacationHours();
      alterForLegalMinimums();
      applyToPayroll();
   }

   private void calculateBaseVacationHours() { /* ... */ };
   abstract protected void alterForLegalMinimums();
   private void applyToPayroll() { /* ... */ };
}

public class USVacationPolicy extends VacationPolicy {
   @Override protected void alterForLegalMinimums() {
      // Логика для США
   }
}

public class EUVacationPolicy extends VacationPolicy {
   @Override protected void alterForLegalMinimums() {
      // Логика для Европы
   }
}
```

Субклассы «заполняют пробел» в обобщенном алгоритме `accrueVacation`; они предоставляют только ту информацию, которая различается в специализированных версиях алгоритма.

Выразительность

Большинству читателей доводилось работать с запутанным кодом. Многие из них создавали запутанный код сами. Легко написать код, понятный для нас самих,

потому что в момент его написания мы глубоко понимаем решаемую проблему. У других программистов, которые будут заниматься сопровождением этого кода, такого понимания не будет.

Основные затраты программного проекта связаны с его долгосрочным сопровождением. Чтобы свести к минимуму риск появления дефектов в ходе внесения изменений, очень важно понимать, как работает система. С ростом сложности системы разработчику приходится разбираться все дольше и дольше, а вероятность того, что он поймет что-то неправильно, только возрастает. Следовательно, код должен четко выражать намерения своего автора. Чем понятнее будет код, тем меньше времени понадобится другим программистам, чтобы разобраться в нем. Это способствует уменьшению количества дефектов и снижению затрат на сопровождение.

Хороший выбор имен помогает выразить ваши намерения. Имя класса или функции должно восприниматься «на слух», а когда читатель разбирается в том, что делает класс, это не должно вызывать у него удивления.

Относительно небольшой размер функций и классов также помогает выразить ваши намерения. Компактным классам и функциям проще присваивать имена; они легко пишутся и в них легко разобраться.

Стандартная номенклатура также способствует выражению намерений автора. В частности, передача информация и выразительность являются важнейшими целями для применения паттернов проектирования. Включение стандартных названий паттернов (например, КОМАНДА или ПОСЕТИТЕЛЬ) в имена классов, реализующих эти паттерны, помогает кратко описать вашу архитектуру для других разработчиков.

Хорошо написанные модульные тесты тоже выразительны. Они могут рассматриваться как разновидность документации, построенная на конкретных примерах. Читая код тестов, разработчик должен составить хотя бы общее представление о том, что делает класс.

И все же самое важное, что можно сделать для создания выразительного кода — это *постараться* сделать его выразительным. Как только наш код заработает, мы обычно переходим к следующей задаче, не прикладывая особых усилий к тому, чтобы код легко читался другими людьми. Но помните: следующим человеком, которому придется разбираться в вашем коде, с большой вероятностью окажетесь вы сами.

Так что уделите немного внимания качеству исполнения своего продукта. Немного поразмыслите над каждой функцией и классом. Попробуйте улучшить имена, разбейте большие функции на меньшие и вообще проявите заботу о том, что вы создали. Неравнодушие — воистину драгоценный ресурс.

Минимум классов и методов

Даже такие фундаментальные концепции, как устранение дубликатов, выразительность кода и принцип единой ответственности, могут зайти слишком далеко. Стремясь уменьшить объем кода наших классов и методов, мы можем наплодить слишком много крошечных классов и методов. Это правило рекомендует ограничиться небольшим количеством функций и классов.

Многочисленность классов и методов иногда является результатом бессмысленного догматизма. В качестве примера можно привести стандарт кодирования, который требует создания интерфейса для каждого без исключения класса. Или разработчиков, настаивающих, что поля данных и поведение всегда должны быть разделены на классы данных и классы поведения. Избегайте подобных догм, а в своей работе руководствуйтесь более прагматичным подходом.

Наша цель — сделать так, чтобы система была компактной, но при этом одновременно сохранить компактность функций и классов. Однако следует помнить, что из четырех правил простой архитектуры это правило обладает наименьшим приоритетом. Свести к минимуму количество функций и классов важно, однако прохождение тестов, устранение дубликатов и выразительность кода все же важнее.

Заключение

Может ли набор простых правил заменить практический опыт? Нет, конечно. С другой стороны, правила, описанные в этой главе и в книге, представляют собой кристаллизованную форму многих десятилетий практического опыта авторов. Принципы простой архитектуры помогают разработчикам следовать по тому пути, который им пришлось бы самостоятельно прокладывать в течение многих лет.

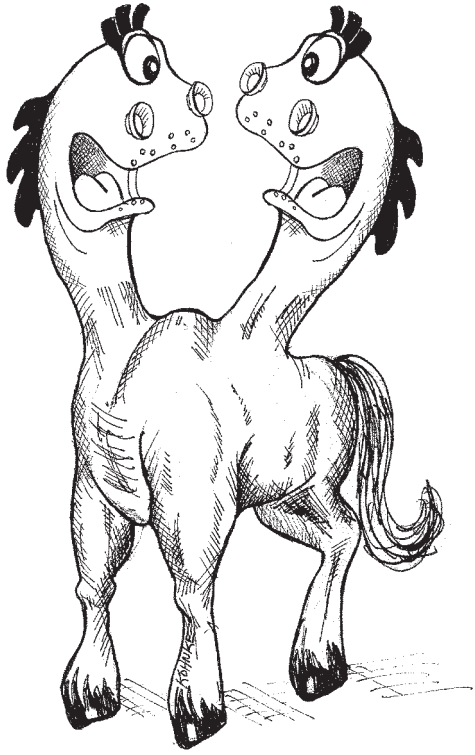
Литература

[XPE]: Extreme Programming Explained: Embrace Change, Kent Beck, Addison-Wesley, 1999.

[GOF]: Design Patterns: Elements of Reusable Object Oriented Software, Gamma et al., Addison-Wesley, 1996.

13 Многопоточность

Бретт Л. Шухерт



Объекты — абстракции для обработки данных. Программные потоки — абстракции для планирования.

Джеймс О. Коплиен

Написать чистую многопоточную программу трудно — очень трудно. Гораздо проще писать код, выполняемый в одном программном потоке. Многопоточный

код часто выглядит нормально на первый взгляд, но содержит дефекты на более глубоком уровне. Такой код работает нормально до тех пор, пока система не заработает с повышенной нагрузкой.

В этой главе мы поговорим о том, почему необходимо многопоточное программирование и какие трудности оно создает. Далее будут представлены рекомендации относительно того, как справиться с этими трудностями и как написать чистый многопоточный код. В завершение главы рассматриваются проблемы тестирования многопоточного кода.

Чистый многопоточный код — сложная тема, по которой вполне можно было бы написать отдельную книгу. В этой главе приводится обзор, а более подробный учебный материал содержится в приложении «Многопоточность II» на с. 357. Если вы хотите получить общее представление о многопоточности, этой главы будет достаточно. Чтобы разобраться в теме на более глубоком уровне, читайте вторую главу.

Зачем нужна многопоточность?

Многопоточное программирование может рассматриваться как стратегия устранения привязок. Оно помогает отделить выполняемую операцию от момента ее выполнения. В однопоточных приложениях «что» и «когда» связаны так сильно, что просмотр содержимого стека часто позволяет определить состояние всего приложения. Программист, отлаживающий такую систему, устанавливает точку прерывания (или серию точек прерывания) и узнает состояние системы на момент остановки.

Отделение «что» от «когда» способно кардинально улучшить как производительность, так и структуру приложения. Со структурной точки зрения многопоточное приложение выглядит как взаимодействие нескольких компьютеров, а не как один большой управляющий цикл. Такая архитектура упрощает понимание системы и предоставляет мощные средства для разделения ответственности.

Для примера возьмем «сервлет», одну из стандартных моделей веб-приложений. Такие системы работают под управлением веб-контейнера или контейнера EJB, который частично управляет многопоточностью за разработчика. Сервлеты выполняются асинхронно при поступлении веб-запросов. Разработчику сервера не нужно управлять входящими запросами. В принципе каждый выполняемый экземпляр сервлета существует в своем замкнутом мире, отделенном от всех остальных экземпляров сервлетов.

Конечно, если бы все было так просто, эта глава стала бы ненужной. Изоляция, обеспечиваемая веб-контейнерами, далеко не идеальна. Чтобы многопоточный код работал корректно, разработчики сервлетов должны действовать очень внимательно и осторожно. И все же структурные преимущества модели сервлетов весьма значительны.

Но структура — не единственный аргумент для многопоточного программирования. В некоторых системах действуют ограничения по времени отклика и пропускной способности, требующие ручного кодирования многопоточных решений. Для примера возьмем однопоточный агрегатор, который получает информацию с многих сайтов и объединяет ее в ежедневную сводку. Так как система работает в однопоточном режиме, она последовательно обращается к каждому сайту, всегда завершая получение информации до перехода к следующему сайту. Ежедневный сбор информации должен занимать менее 24 часов. Но по мере добавления новых сайтов время непрерывно растет, пока в какой-то момент на сбор всех данных не потребуется более 24 часов. Однопоточной реализации приходится подолгу ожидать завершения операций ввода/вывода в сокетах. Для повышения производительности такого приложения можно было бы воспользоваться многопоточным алгоритмом, параллельно работающим с несколькими сайтами.

Или другой пример: допустим, система в любой момент времени работает только с одним пользователем, обслуживание которого у нее занимает всего одну секунду. При малом количестве пользователей система оперативно реагирует на все запросы, но с увеличением количества пользователей растет и время отклика. Никто не захочет стоять в очереди после 150 других пользователей! Время отклика такой системы можно было бы улучшить за счет параллельного обслуживания многих пользователей.

Или возьмем систему, которая анализирует большие объемы данных, но выдает окончательный результат только после их полной обработки. Наборы данных могут обрабатываться параллельно на разных компьютерах.

Мифы и неверные представления

Итак, существуют весьма веские причины для использования многопоточности. Но как говорилось ранее, написать многопоточную программу трудно. Необходимо действовать очень осторожно, иначе в программе могут возникнуть крайне неприятные ситуации. С многопоточностью связан целый ряд распространенных мифов и неверных представлений.

- *Многопоточность всегда повышает быстродействие.*

  Действительно, многопоточность иногда повышает быстродействие, но только при относительно большом времени ожидания, которое могло бы эффективно использоваться другими потоками или процессорами.

- *Написание многопоточного кода не изменяет архитектуру программы.*

  На самом деле архитектура многопоточного алгоритма может заметно отличаться от архитектуры однопоточной системы. Отделение «что» от «когда» обычно оказывает огромное влияние на структуру системы.

- *При работе с контейнером (например, веб-контейнером или EJB-контейнером) разбираться в проблемах многопоточного программирования не обязательно.*

В действительности желательно знать, как работает контейнер и как защититься от проблем одновременного обновления и взаимных блокировок, описанных позднее в этой главе.

Несколько более объективных утверждений, относящихся к написанию многопоточного кода:

- Многопоточность сопряжена с определенными дополнительными затратами — в отношении как производительности, так и написания дополнительного кода.
- Правильная реализация многопоточности сложна даже для простых задач.
- Ошибки в многопоточном коде обычно не воспроизводятся, поэтому они часто игнорируются как случайные отклонения[1] (а не как систематические дефекты, которыми они на самом деле являются).
- Многопоточность часто требует фундаментальных изменений в стратегии проектирования.

Трудности

Что же делает многопоточное программирование таким сложным? Рассмотрим тривиальный класс:

```
public class X {
   private int lastIdUsed;
   public int getNextId() {
        return ++lastIdUsed;
    }
}
```

Допустим, мы создаем экземпляр `X`, присваиваем полю `lastIdUsed` значение 42, а затем используем созданный экземпляр в двух программных потоках. В обоих потоках вызывается метод `getNextId()`; возможны три исхода:

- Первый поток получает значение 43, второй получает значение 44, в поле `lastIdUsed` сохраняется 44.
- Первый поток получает значение 44, второй получает значение 43, в поле `lastIdUsed` сохраняется 44.
- Первый поток получает значение 43, второй получает значение 43, поле `lastIdUsed` содержит 43.

Удивительный третий результат[2] встречается тогда, когда два потока «перебивают» друг друга. Это происходит из-за того, что выполнение одной строки кода Java в двух потоках может пойти по разным путям, и некоторые из этих путей порождают неверные результаты. Сколько существует разных путей? Чтобы ответить на этот вопрос, необходимо понимать, как JIT-компилятор обрабатывает

[1] Фазы Луны, космические лучи и т. д.

[2] См. раздел «Копаем глубже» на с. 364.

сгенерированный байт-код, и разбираться в том, какие операции рассматриваются моделью памяти Java как атомарные.

В двух словах скажу, что в сгенерированном байт-коде приведенного фрагмента существует 12 870 разных путей выполнения[1] метода `getNextId` в двух программных потоках. Если изменить тип `lastIdUsed` с `int` на `long`, то количество возможных путей возрастет до 2 704 156. Конечно, на большинстве путей выполнения вычисляются правильные результаты. Проблема в том, что на *некоторых* путях результаты будут неправильными.

Защита от ошибок многопоточности

Далее перечислены некоторые принципы и приемы, которые помогают защитить вашу систему от проблем многопоточности.

Принцип единой ответственности

Принцип единой ответственности (SRP) [PPP] гласит, что метод/класс/компонент должен иметь только одну причину для изменения. Многопоточные архитектуры достаточно сложны, чтобы их можно было рассматривать как причину изменения сами по себе, а следовательно, они должны отделяться от основного кода. К сожалению, подробности многопоточной реализации нередко встраиваются в другой код. Однако разработчик должен учитывать ряд факторов:

- Код реализации многопоточности имеет собственный цикл разработки, модификации и настройки.
- При написании кода реализации многопоточности возникают специфические сложности, принципиально отличающиеся от сложностей однопоточного кода (и часто превосходящие их).
- Количество потенциальных сбоев в неверно написанном многопоточном коде достаточно велико и без дополнительного бремени в виде окружающего кода приложения.

Рекомендация: *отделяйте код, относящийся к реализации многопоточности, от остального кода*[2].

Следствие: ограничивайте область видимости данных

Как было показано ранее, два программных потока, изменяющих одно поле общего объекта, могут мешать друг другу, что приводит к непредвиденному поведению. Одно из возможных решений — защита критической секции кода, в которой про-

[1] См. раздел «Пути выполнения» на с. 262.

[2] См. раздел «Пример архитектуры «клиент/сервер»» на с. 357.

исходят обращения к общему объекту, ключевым словом `synchronized`. Количество критических секций в коде должно быть сведено к минимуму. Чем больше в программе мест, в которых обновляются общие данные, тем с большей вероятностью:

- вы забудете защитить одно или несколько из этих мест, что приведет к нарушению работы всего кода, изменяющего общие данные.
- попытки уследить за тем, чтобы все было надежно защищено, приведут к дублированию усилий (нарушение принципа DRY [PRAG]).

Вам будет труднее определить источник многопоточных сбоев, который и так достаточно сложно найти.

Рекомендация: *серьезно относитесь к инкапсуляции данных; жестко ограничьте доступ ко всем общим данным.*

Следствие: используйте копии данных

Как избежать нежелательных последствий одновременного доступа к данным? Например, просто не использовать его. Существуют разные стратегии: например, в одних ситуациях можно скопировать общий объект и ограничить доступ к копии (доступ только для чтения). В других ситуациях объекты копируются, результаты работы нескольких программных потоков накапливаются в копиях, а затем объединяются в одном потоке.

Если существует простой способ избежать одновременного доступа к объектам, то вероятность возникновения проблем в полученном коде значительно снижается. Вас беспокоят затраты на создание лишних объектов? Поэкспериментируйте и выясните, действительно ли она так высока. Как правило, если копирование объектов позволяет избежать синхронизации в коде, экономия на защитных блокировках быстро окупит дополнительные затраты на создание объектов и уборку мусора.

Следствие: потоки должны быть как можно более независимы

Постарайтесь писать многопоточный код так, чтобы каждый поток существовал в собственном замкнутом пространстве и не использовал данные совместно с другими процессами. Каждый поток обрабатывает один клиентский запрос, все его данные берутся из отдельного источника и хранятся в локальных переменных. В этом случае каждый поток работает так, словно других потоков не существует, а следовательно, нет и требований к синхронизации.

Например, классы, производные от `HttpServlet`, получают всю информацию в параметрах, передаваемых методам `doGet` и `doPost`. В результате каждый сервлет действует так, словно в его распоряжении находится отдельный компьютер. Если код сервлета ограничивается одними локальными переменными, он ни при каких условиях не вызовет проблем синхронизации. Конечно, большинство приложе-

ний, использующих сервлеты, рано или поздно сталкиваются с использованием общих ресурсов — например, подключений к базам данных.

Рекомендация: постарайтесь разбить данные не независимые подмножества, с которыми могут работать независимые потоки (возможно, на разных процессорах).

Знайте свою библиотеку

В Java 5 возможности многопоточной разработки были значительно расширены по сравнению с предыдущими версиями. При написании многопоточного кода в Java 5 следует руководствоваться следующими правилами:

- Используйте потоково-безопасные коллекции.
- Используйте механизм Executor Framework для выполнения несвязанных задач.
- По возможности используйте неблокирующие решения.
- Некоторые библиотечные классы не являются потоково-безопасными.

Потоково-безопасные коллекции

Когда язык Java был еще молод, Даг Ли написал основополагающую книгу «Concurrent Programming in Java» [Lea99]. В ходе работы над книгой он разработал несколько потоково-безопасных коллекций, которые позднее были включены в JDK в пакете `java.util.concurrent`. Коллекции этого пакета безопасны в условиях многопоточного выполнения, к тому же они достаточно эффективно работают. Более того, реализация `ConcurrentHashMap` почти всегда работает лучше `HashMap`. К тому же она поддерживает возможность выполнения параллельных операций чтения и записи и содержит методы для выполнения стандартных составных операций, которые в общем случае не являются потоково-безопасными. Если ваша программа будет работать в среде Java 5, используйте `ConcurrentHashMap` в разработке.

Также в Java 5 были добавлены другие классы для поддержки расширенной многопоточности. Несколько примеров.

`ReentrantLock`	Блокировка, которая может устанавливаться и освобождаться в разных методах
`Semaphore`	Реализация классического семафора (блокировка со счетчиком)
`CountDownLatch`	Блокировка, которая ожидает заданного количества событий до освобождения всех ожидающих потоков. Позволяет организовать более или менее одновременный запуск нескольких потоков

Рекомендация: *изучайте доступные классы. Если вы работаете на Java, уделите особое внимание пакетам* `java.util.concurrent`, `java.util.concurrent.atomic` *и* `java.util.concurrent.locks`.

Знайте модели выполнения

В многопоточных приложениях возможно несколько моделей логического разбиения поведения программы. Но чтобы понять их, необходимо сначала познакомиться с некоторыми базовыми определениями.

Связанные ресурсы	Ресурсы с фиксированным размером или количеством, существующие в многопоточной среде, например подключения к базе данных или буферы чтения/записи
Взаимное исключение	В любой момент времени с общими данными или с общим ресурсом может работать только один поток
Зависание	Работа одного или нескольких потоков приостанавливается на слишком долгое время (или навсегда). Например, если высокоприоритетным потокам всегда предоставляется возможность отработать первыми, то низкоприоритетные потоки зависнут (при условии, что в системе постоянно появляются новые высокоприоритетные потоки)
Взаимная блокировка (deadlock)	Два и более потока бесконечно ожидают завершения друг друга. Каждый поток захватил ресурс, необходимый для продолжения работы другого потока, и ни один поток не может завершиться без получения захваченного другим потоком ресурса
Обратимая блокировка[1] (livelock)	Потоки не могут «разойтись» — каждый из потоков пытается выполнять свою работу, но обнаруживает, что другой поток стоит у него на пути. Потоки постоянно пытаются продолжить выполнение, но им это не удается в течение слишком долгого времени (или вообще не удается)

Вооружившись этими определениями, можно переходить к обсуждению различных моделей выполнения, встречающихся в многопоточном программировании.

Модель «производители-потребители»[1]

Один или несколько потоков-производителей создают *задания* и помещают их в буфер или очередь. Один или несколько потоков-потребителей извлекают задания из очереди и выполняют их. Очередь между производителями и потребителями является связанным ресурсом. Это означает, что производители перед записью должны дожидаться появления свободного места в очереди, а потребители должны дожидаться появления заданий в очереди для обработки. Координация производителей и потребителей основана на передаче сигналов. Производитель записывает задание и сигнализирует о том, что очередь не пуста. Потребитель читает задание и сигнализирует о том, что очередь не заполнена. Обе стороны должны быть готовы ожидать оповещения о возможности продолжения работы.

---

[1] Также встречается термин «активная блокировка». — *Примеч. перев.*

[2] http://en.wikipedia.org/wiki/Producer-consumer

Модель «читатели-писатели»[1]

Если в системе имеется общий ресурс, который в основном служит источником информации для потоков-«читателей», но время от времени обновляется потоками-«писателями», на первый план выходит проблема оперативности обновления. Если обновление будет происходить недостаточно часто, это может привести к зависанию и накоплению устаревших данных. С другой стороны, слишком частые обновления влияют на производительность. Координация работы читателей так, чтобы они не пытались читать данные, обновляемые писателями, и наоборот, — весьма непростая задача. Писатели обычно блокируют работу многих читателей в течение долгого периода времени, а это отражается на производительности.

Проектировщик должен найти баланс между потребностями читателей и писателей, чтобы обеспечить правильный режим работы, нормальную производительность системы и избежать зависания. В одной из простых стратегий писатели дожидаются, пока в системе не будет ни одного читателя, и только после этого выполняют обновление. Однако при постоянном потоке читателей такая стратегия приведет к зависанию писателей. С другой стороны, при большом количестве высокоприоритетных писателей пострадает производительность. Поиск баланса и предотвращение ошибок многопоточного обновления — основные проблемы этой модели выполнения.

Модель «обедающих философов»[2]

Представьте нескольких философов, сидящих за круглым столом. Слева у каждого философа лежит вилка, а в центре стола стоит большая тарелка спагетти. Философы проводят время в размышлениях, пока не проголодаются. Проголодавшись, философ берет вилки, лежащие по обе стороны, и приступает к еде. Для еды необходимы две вилки. Если сосед справа или слева уже использует одну из необходимых вилок, философу приходится ждать, пока сосед закончит есть и положит вилки на стол. Когда философ поест, он кладет свои вилки на стол и снова погружается в размышления.

Заменив философов программными потоками, а вилки — ресурсами, мы получаем задачу, типичную для многих корпоративных систем, в которых приложения конкурируют за ресурсы из ограниченного набора. Если небрежно отнестись к проектированию такой системы, то конкуренция между потоками может привести к возникновению взаимных блокировок, обратимых блокировок, падению производительности и эффективности работы.

Большинство проблем многопоточности, встречающихся на практике, обычно представляют собой те или иные разновидности этих трех моделей. Изучайте

[1] http://en.wikipedia.org/wiki/Readers-writers_problem
[2] http://en.wikipedia.org/wiki/Dining_philosophers_problem

алгоритмы, самостоятельно создавайте их реализации, чтобы столкнувшись с этими проблемами, вы были готовы к их решению.

Рекомендация: *изучайте базовые алгоритмы, разбирайтесь в решениях.*

Остерегайтесь зависимостей между синхронизированными методами

Зависимости между синхронизированными методами приводят к появлению коварных ошибок в многопоточном коде. В языке Java существует ключевое слово `synchronized` для защиты отдельных методов. Но если общий класс содержит более одного синхронизированного метода, возможно, ваша система спроектирована неверно[1].

Рекомендация: *избегайте использования нескольких методов одного совместно используемого объекта.*

Впрочем, иногда без использования разных методов одного общего объекта обойтись все же не удается. Для обеспечения правильности работы кода в подобных ситуациях существуют три стандартных решения:

- *Блокировка на стороне клиента* — клиент устанавливает блокировку для сервера перед вызовом первого метода и следит за тем, чтобы блокировка распространялась на код, вызывающий последний метод.
- *Блокировка на стороне сервера* — на стороне сервера создается метод, который блокирует сервер, вызывает все методы, после чего снимает блокировку. Этот новый метод вызывается клиентом.
- *Адаптирующий сервер* — в системе создается посредник, который реализует блокировку. Ситуация может рассматриваться как пример блокировки на стороне сервера, в которой исходный сервер не может быть изменен.

Синхронизированные секции должны иметь минимальный размер

Ключевое слово `synchronized` устанавливает блокировку. Все секции кода, защищенные одной блокировкой, в любой момент времени гарантированно выполняются только в одном программном потоке. Блокировки обходятся дорого, так как они создают задержки и увеличивают затраты ресурсов. Следовательно, код не должен перегружаться лишними конструкциями `synchronized`. С другой

[1] См. раздел «Зависимости между методами могут нарушить работу многопоточного кода», с. 370.

стороны, все критические секции[1] должны быть защищены. Следовательно, код должен содержать как можно меньше критических секций.

Для достижения этой цели некоторые наивные программисты делают свои критические секции очень большими. Однако синхронизация за пределами минимальных критических секций увеличивает конкуренцию между потоками и снижает производительность[2].

Рекомендация: *синхронизированные секции в ваших программах должны иметь минимальные размеры.*

О трудности корректного завершения

Написание системы, которая должна работать бесконечно, заметно отличается от написания системы, которая работает в течение некоторого времени, а затем корректно завершается.

Реализовать корректное завершение порой бывает весьма непросто. Одна из типичных проблем — взаимная блокировка[3] программных потоков, бесконечно долго ожидающих сигнала на продолжение работы.

Представьте систему с родительским потоком, который порождает несколько дочерних потоков, а затем дожидается их завершения, чтобы освободить свои ресурсы и завершиться. Что произойдет, если один из дочерних потоков попадет во взаимную блокировку? Родитель будет ожидать вечно, и система не сможет корректно завершиться.

Или возьмем аналогичную систему, получившую сигнал о завершении. Родитель приказывает всем своим потомкам прервать свои операции и завершить работу. Но что если два потомка составляют пару «производитель/потребитель»? Допустим, производитель получает сигнал от родителя, и прерывает свою работу. Потребитель, в этот момент ожидавший сообщения от производителя, блокируется в состоянии, в котором он не может получить сигнал завершения. В результате он переходит в бесконечное ожидание — а значит, родитель тоже не сможет завершиться.

Подобные ситуации вовсе не являются нетипичными. Если вы пишете многопоточный код, который должен корректно завершаться, не жалейте времени на обеспечение нормального завершения работы.

Рекомендация: *начинайте думать о корректном завершении на ранней стадии разработки. На это может уйти больше времени, чем вы предполагаете. Проанализируйте существующие алгоритмы, потому что эта задача сложнее, чем кажется.*

---

[1] «Критической секцией» называется любой фрагмент кода, который должен быть защищен от одновременного использования несколькими программными потоками.

[2] См. раздел «Увеличение производительности», с. 375.

[3] См. раздел «Взаимная блокировка», с. 377.

Тестирование многопоточного кода

Тестирование не гарантирует правильности работы кода. Тем не менее качественное тестирование сводит риск к минимуму. Для однопоточных решений эти утверждения безусловно верны. Но как только в системе появляются два и более потока, использующие общий код и работающих с общими данными, ситуация значительно усложняется.

Рекомендация: *пишите тесты, направленные на выявление существующих проблем. Часто выполняйте их для разных вариантов программных/системных конфигураций и уровней нагрузки. Если при выполнении теста происходит ошибка, обязательно найдите причину. Не игнорируйте ошибку только потому, что при следующем запуске тест был выполнен успешно.*

Несколько более конкретных рекомендаций:

- Рассматривайте непериодические сбои как признаки возможных проблем многопоточности.
- Начните с отладки основного кода, не связанного с многопоточностью.
- Реализуйте логическую изоляцию конфигураций многопоточного кода.
- Обеспечьте возможность настройки многопоточного кода.
- Протестируйте программу с количеством потоков, превышающим количество процессоров.
- Протестируйте программу на разных платформах.
- Применяйте инструментовку кода для повышения вероятности сбоев.

Рассматривайте непериодические сбои как признаки возможных проблем многопоточности

В многопоточном коде сбои происходят даже там, где их вроде бы и быть не может. Многие разработчики (в том числе и автор) не обладают интуитивным представлением о том, как многопоточный код взаимодействует с другим кодом. Ошибки в многопоточном коде могут проявляться один раз за тысячу или даже миллион запусков. Воспроизвести такие ошибки в системе бывает очень трудно, поэтому разработчики часто склонны объяснять их «фазами Луны», случайными сбоями оборудования или другими несистематическими причинами. Однако игнорируя существование этих «разовых» сбоев, вы строите свой код на потенциально ненадежном фундаменте.

Рекомендация: *не игнорируйте системные ошибки, считая их случайными, разовыми сбоями.*

Начните с отладки основного кода, не связанного с многопоточностью

На первый взгляд совет выглядит тривиально, но еще раз подчеркнуть его значимость не лишне. Убедитесь в том, что сам код работает вне многопоточного контекста. В общем случае это означает создание POJO-объектов, вызываемых из потоков. POJO-объекты не обладают поддержкой многопоточности, а следовательно, могут тестироваться вне многопоточной среды. Чем больше системного кода можно разместить в таких POJO-объектах, тем лучше.

Рекомендация: не пытайтесь одновременно отлавливать ошибки в обычном и многопоточном коде. Убедитесь в том, что ваш код работает за пределами многопоточной среды выполнения.

Реализуйте переключение конфигураций многопоточного кода

Напишите вспомогательный код поддержки многопоточности, который может работать в разных конфигурациях.

- Один поток; несколько потоков; количество потоков изменяется по ходу выполнения.
- Многопоточный код взаимодействует с реальным кодом или тестовыми заменителями.
- Код выполняется с тестовыми заменителями, которые работают быстро; медленно; с переменной скоростью.
- Настройте тесты таким образом, чтобы они могли выполняться заданное количество раз.

Рекомендация: *реализуйте свой многопоточный код так, чтобы он мог выполняться в различных конфигурациях.*

Обеспечьте логическую изоляцию конфигураций многопоточного кода

Правильный баланс программных потоков обычно определяется методом проб и ошибок. Прежде всего найдите средства измерения производительности системы в разных конфигурациях. Реализуйте систему так, чтобы количество программных потоков могло легко изменяться. Подумайте, нельзя ли разрешить его изменение во время работы системы. Рассмотрите возможность автоматической настройки в зависимости от текущей производительности и загрузки системы.

Протестируйте программу с количеством потоков, превышающим количество процессоров

При переключении контекста системы между задачами могут происходить всякие неожиданности. Чтобы форсировать переключение задач, выполняйте свой код с количеством потоков, превышающим количество физических процессоров или ядер. Чем чаще происходит переключение задач, тем больше вероятность выявления пропущенной критической секции или возникновения взаимной блокировки.

Протестируйте программу на разных платформах

В середине 2007-го года мы разрабатывали учебный курс по многопоточному программированию. Разработка курса велась в OS X. Материал курса излагался в системе Windows XP, запущенной на виртуальной машине. Однако сбои в тестах, написанных для демонстрации ошибок, происходили в среде XP заметно реже, чем при запуске в OS X.

Тестируемый код всегда был заведомо некорректным. Эта история лишний раз доказывает, что в разных операционных системах используются разные политики многопоточности, влияющие на выполнение кода. Многопоточный код поразному работает в разных средах[1].

Протестируйте систему во всех средах, которые могут использоваться для ее развертывания.

Рекомендация: *многопоточный код необходимо тестировать на всех целевых платформах — часто и начиная с ранней стадии.*

Применяйте инструментовку кода для повышения вероятности сбоев

Ошибки в многопоточном коде обычно хорошо скрыты от наших глаз. Простыми тестами они не выявляются. Такие ошибки могут проявляться с периодичностью в несколько часов, дней или недель!

Почему же многопоточные ошибки возникают так редко и непредсказуемо, почему их так трудно воспроизвести? Потому что лишь несколько из тысяч возможных путей выполнения кода плохо написанной секции приводят к фактическому отказу. Таким образом, вероятность выбора сбойного пути ничтожно мала. Это обстоятельство серьезно усложняет выявление ошибок и отладку.

---

[1] А вы знаете, что потоковая модель Java не гарантирует вытесняющей многопоточности? В большинстве современных ОС поддерживается вытесняющая многопоточность, которую вы фактически получаете автоматически. И все же JVM ее не гарантирует.

Как повысить вероятность выявления таких редких ошибок? Внесите в свой код соответствующие изменения и заставьте его выполняться по разным путям — включите в него вызовы таких методов, как `Object.wait()`, `Object.sleep()`, `Object.yield()` и `Object.priority()`.

Каждый из этих методов влияет на порядок выполнения программы, повышая шансы на выявление сбоя. Сбои в дефектном коде должны выявляться как можно раньше и как можно чаще.

Существует два способа инструментовки кода:

- Ручная.
- Автоматическая.

Ручная инструментовка

Разработчик вставляет вызовы `wait()`, `sleep()`, `yield()` и `priority()` в свой код вручную. Такой вариант отлично подходит для тестирования особенно коварных фрагментов кода.

Пример:

```
public synchronized String nextUrlOrNull() {
    if(hasNext()) {
        String url = urlGenerator.next();
        Thread.yield(); // Вставлено для тестирования
        updateHasNext();
        return url;
    }
    return null;
}
```

Добавленный вызов `yield()` изменяет путь выполнения кода. В результате в программе может произойти сбой там, где раньше его не было. Если работа программы действительно нарушается, то это произошло не из-за того, что вы добавили вызов `yield()`[1]. Просто ваш код содержал скрытые ошибки, а в результате вызова `yield()` они стали очевидными.

Ручная инструментовка имеет много недостатков:

- Разработчик должен каким-то образом найти подходящие места для вставки вызовов.
- Как узнать, где и какой именно вызов следует вставить?
- Если вставленные вызовы останутся в окончательной версии кода, это приведет к замедлению его работы.
- Вам приходится действовать «наобум»: вы либо находите скрытые дефекты, либо не находите их. Вообще говоря, шансы не в вашу пользу.

[1] Строго говоря, это не совсем так. Поскольку JVM не гарантирует вытесняющей многопоточности, конкретный алгоритм может всегда работать в ОС, не поддерживающей вытеснения. Обратное тоже возможно, но по другим причинам.

Отладочные вызовы должны присутствовать только на стадии тестирования, но не в окончательной версии кода. Кроме того, вам понадобятся средства для простого переключения конфигураций между запусками, повышающего вероятность обнаружения ошибок в общей кодовой базе.

Конечно, разделение системы на POJO-объекты, ничего не знающие о многопоточности, и классы, управляющие многопоточностью, упрощает поиск подходящих мест для инструментовки кода. Кроме того, такое разделение позволит нам создать целый набор «испытательных пакетов», активизирующих POJO-объекты с разными режимами вызова `sleep`, `yield` и т. д.

Автоматизированная инструментовка

Также возможна программная инструментовка кода с применением таких инструментов, как Aspect-Oriented Framework, CGLIB или ASM. Допустим, в программу включается класс с единственным методом:

```
public class ThreadJigglePoint {
    public static void jiggle() {
    }
}
```

Вызовы этого метода размещаются в разных позициях кода:

```
public synchronized String nextUrlOrNull() {
    if(hasNext()) {
        ThreadJiglePoint.jiggle();
        String url = urlGenerator.next();
        ThreadJiglePoint.jiggle();
        updateHasNext();
        ThreadJiglePoint.jiggle();
        return url;
    }
    return null;
}
```

Теперь в вашем распоряжении появился простой аспект, случайным образом выбирающий между обычным продолжением работы, приостановкой и передачей управления.

Или представьте, что класс `ThreadJigglePoint` имеет две реализации. В первой реализации `jiggle` не делает ничего; эта реализация используется в окончательной версии кода. Вторая реализация генерирует случайное число для выбора между приостановкой, передачей управления и обычным выполнением. Если теперь повторить тестирование тысячу раз со случайным выбором, возможно, вам удастся выявить некоторые дефекты. Даже если тестирование пройдет успешно, по крайней мере вы сможете сказать, что приложили должные усилия для выявления недостатков. Такой подход выглядит несколько упрощенно, но и он может оказаться разумной альтернативой для применения более сложных инструментов.

Программа ConTest[1], разработанная фирмой IBM, работает по аналогичному принципу, но предоставляет расширенные возможности.

Впрочем, суть тестирования остается неизменной: вы ломаете предсказуемость пути выполнения, чтобы при разных запусках код проходил по разным путям. Комбинация хорошо написанных тестов и случайного выбора пути может радикально повысить вероятность поиска ошибок.

Рекомендация: *используйте стратегию случайного выбора пути выполнения для выявления ошибок.*

Заключение

Правильно написать многопоточный код непросто. Даже очевидный, хорошо понятный код превращается в сущий кошмар, когда в игру вступают множественные потоки и одновременный доступ к данным. Если вы столкнулись с задачей из области многопоточного программирования, вам придется приложить все усилия к написанию чистого кода или столкнуться с коварными, непредсказуемыми сбоями.

Прежде всего следуйте принципу единой ответственности. Разбейте систему на POJO-объекты, отделяющие многопоточный код от кода, с потоками никак не связанного. Проследите за тем, чтобы при тестировании многопоточного кода тестировался только этот код, и ничего лишнего. Из этого следует, что многопоточный код должен быть компактным и сконцентрированным в одном месте.

Знайте типичные источники многопоточных ошибок: работа с общими данными из нескольких программных потоков, использование пула общих ресурсов. Особенно непростыми оказываются пограничные случаи: корректное завершение работы, завершение итераций циклов и т. д.

Изучайте свои библиотеки и знайте фундаментальные алгоритмы. Разберитесь в том, как некоторые функции библиотек используются для решения проблем, сходных с проблемами фундаментальных алгоритмов.

Научитесь находить секции кода, которые должны защищаться блокировками, и защищайте их. Не устанавливайте блокировки для тех секций, которые защищать не нужно. Избегайте вызовов одной заблокированной секции из другой заблокированной секции — для них необходимо глубокое понимание того, какие ресурсы находятся в общем или монопольном доступе. Сведите к минимуму количество совместно используемых объектов и масштаб общего доступа. Измените архитектуру объектов с общими данными так, чтобы они поддерживали одновременные обращения со стороны клиентов, вместо того чтобы заставлять самих клиентов заниматься управлением состоянием общего доступа.

[1] http://www.alphaworks.ibm.com/tech/contest

В ходе программирования неизбежно возникнут проблемы. Те из них, которые не проявляются на самой ранней стадии, часто списываются на случайности. Эти так называемые «несистематические» ошибки часто встречаются только при высокой нагрузке или вообще в случайные (на первый взгляд) моменты. Следовательно, вы должны позаботиться о том, чтобы ваш многопоточный код мог многократно запускаться в разных конфигурациях на многих платформах. Тестируемость, естественным образом проистекающая из трех законов TDD, подразумевает определенный уровень модульности, которая обеспечивает возможность выполнения кода в более широком диапазоне конфигураций.

Потратив немного времени на инструментовку кода, вы значительно повысите шансы обнаружения некорректного кода. Инструментовка может производиться как вручную, так и с применением технологий автоматизации. Начинайте с ранних стадий работы над продуктом. Многопоточный код должен отработать в течение как можно большего времени, прежде чем он будет включен в окончательную версию продукта.

Если вы будете стремиться к чистоте своего кода, вероятность того, что вам удастся правильно реализовать его, значительно возрастет.

Литература

[Lea99]: Concurrent Programming in Java: Design Principles and Patterns, 2d. ed., Doug Lea, Prentice Hall, 1999.

[PPP]: Agile Software Development: Principles, Patterns, and Practices, Robert C. Martin, Prentice Hall, 2002.

[PRAG]: The Pragmatic Programmer, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

14 Последовательное очищение

Дело о разборе аргументов командной строки

В этой главе представлен вполне реальный сценарий последовательного очищения кода. Мы рассмотрим модуль, который внешне смотрелся вполне достойно, но плохо масштабировался. Вы увидите, как происходила переработка и очистка этого модуля.

Многим из нас время от времени приходится заниматься разбором аргументов командной строки. Если под рукой не окажется удобного инструмента, мы просто перебираем элементы массива строк, переданного функции `main`. Я знал немало хороших инструментов из разных источников, однако ни один из них не делал

именно того, что мне было нужно. Разумеется, я решил написать собственную реализацию — назовем ее Args.

Класс Args очень прост в использовании. Вы конструируете экземпляр класса Args с входными аргументами и форматной строкой, а затем обращаетесь к нему за значениями аргументов. Рассмотрим простой пример.

Листинг 14.1. Простое использование Args

```
public static void main(String[] args) {
  try {
    Args arg = new Args("l,p#,d*", args);
    boolean logging = arg.getBoolean('l');
    int port = arg.getInt('p');
    String directory = arg.getString('d');
    executeApplication(logging, port, directory);
  } catch (ArgsException e) {
    System.out.printf("Argument error: %s\n", e.errorMessage());
  }
}
```

Вы и сами видите, что все действительно просто. Мы создаем экземпляр класса Args с двумя параметрами. Первый параметр задает форматную строку: "l,p#,d*.". Эта строка определяет три аргумента командной строки. Первый аргумент, -l, относится к логическому (булевскому) типу. Второй аргумент, -p, относится к целочисленному типу. Третий аргумент, -d, является строковым. Во втором параметре конструктора Args содержится массив аргументов командной строки, полученный main.

Если конструктор возвращает управление без выдачи исключения ArgsException, значит, разбор входной командной строки прошел успешно, и экземпляр Args готов к приему запросов. Методы getBoolean, getInteger, getString и т. д. используются для получения значений аргументов по именам.

При возникновении проблем (в форматной строке или в самих аргументах командной строки) инициируется исключение ArgsException. Для получения текстового описания проблемы следует вызвать метод errorMessage объекта исключения.

Реализация Args

Реализация класса Args приведена в листинге 14.2. Пожалуйста, очень внимательно прочитайте ее. Я основательно потрудился над стилем и структурой кода и надеюсь, что вы сочтете его достойным образцом для подражания.

Листинг 14.2. Args.java

```
package com.objectmentor.utilities.args;

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;
```

```
import java.util.*;

public class Args {
  private Map<Character, ArgumentMarshaler> marshalers;
  private Set<Character> argsFound;
  private ListIterator<String> currentArgument;

  public Args(String schema, String[] args) throws ArgsException {
    marshalers = new HashMap<Character, ArgumentMarshaler>();
    argsFound = new HashSet<Character>();
    parseSchema(schema);
    parseArgumentStrings(Arrays.asList(args));
  }

  private void parseSchema(String schema) throws ArgsException {
    for (String element : schema.split(","))
      if (element.length() > 0)
        parseSchemaElement(element.trim());
  }

  private void parseSchemaElement(String element) throws ArgsException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
      marshalers.put(elementId, new DoubleArgumentMarshaler());
    else if (elementTail.equals("[*]"))
      marshalers.put(elementId, new StringArrayArgumentMarshaler());
    else
      throw new ArgsException(INVALID_ARGUMENT_FORMAT, elementId, elementTail);
  }
  private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId))
      throw new ArgsException(INVALID_ARGUMENT_NAME, elementId, null);
  }

  private void parseArgumentStrings(List<String> argsList) throws ArgsException
  {
for (currentArgument = argsList.listIterator(); currentArgument.hasNext();)
    {
      String argString = currentArgument.next();
      if (argString.startsWith("-")) {
        parseArgumentCharacters(argString.substring(1));
      } else {
        currentArgument.previous();
```

продолжение ⇩

Листинг 14.2 (продолжение)

```
        break;
      }
    }
  }

  private void parseArgumentCharacters(String argChars) throws ArgsException {
    for (int i = 0; i < argChars.length(); i++)
      parseArgumentCharacter(argChars.charAt(i));
  }

  private void parseArgumentCharacter(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null) {
      throw new ArgsException(UNEXPECTED_ARGUMENT, argChar, null);
    } else {
      argsFound.add(argChar);
      try {
        m.set(currentArgument);
      } catch (ArgsException e) {
        e.setErrorArgumentId(argChar);
        throw e;
      }
    }
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public int nextArgument() {
    return currentArgument.nextIndex();
  }

  public boolean getBoolean(char arg) {
    return BooleanArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public String getString(char arg) {
    return StringArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public int getInt(char arg) {
    return IntegerArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public double getDouble(char arg) {
    return DoubleArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public String[] getStringArray(char arg) {
```

```
    return StringArrayArgumentMarshaler.getValue(marshalers.get(arg));
  }
}
```

Обратите внимание: код читается сверху вниз, и вам не приходится постоянно переходить туда-сюда или заглядывать вперед. Единственное место, где все же необходимо заглянуть вперед, — это определение `ArgumentMarshaler`, но и это было сделано намеренно. Внимательно прочитав этот код, вы поймете, что собой представляет интерфейс `ArgumentMarshaler` и что делают производные классы. Примеры таких классов приведены в листингах 14.3–14.6.

Листинг 14.3. ArgumentMarshaler.java

```
public interface ArgumentMarshaler {
  void set(Iterator<String> currentArgument) throws ArgsException;
}
```

Листинг 14.4. BooleanArgumentMarshaler.java

```
public class BooleanArgumentMarshaler implements ArgumentMarshaler {
  private boolean booleanValue = false;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    booleanValue = true;
  }

  public static boolean getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof BooleanArgumentMarshaler)
      return ((BooleanArgumentMarshaler) am).booleanValue;
    else
      return false;
  }
}
```

Листинг 14.5. StringArgumentMarshaler.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class StringArgumentMarshaler implements ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
    try {
      stringValue = currentArgument.next();
    } catch (NoSuchElementException e) {
      throw new ArgsException(MISSING_STRING);
    }
  }

  public static String getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof StringArgumentMarshaler)
```

продолжение ⇩

Листинг 14.5. (продолжение)

```
      return ((StringArgumentMarshaler) am).stringValue;
    else
      return "";
  }
}
```

Листинг 14.6. IntegerArgumentMarshaler.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class IntegerArgumentMarshaler implements ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      intValue = Integer.parseInt(parameter);
    } catch (NoSuchElementException e) {
      throw new ArgsException(MISSING_INTEGER);
    } catch (NumberFormatException e) {
      throw new ArgsException(INVALID_INTEGER, parameter);
    }
  }

  public static int getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof IntegerArgumentMarshaler)
      return ((IntegerArgumentMarshaler) am).intValue;
    else
      return 0;
  }
}
```

Другие классы, производные от `ArgumentMarshaler`, строятся по тому же шаблону, что и классы для массивов `double` и `String`. Здесь они не приводятся для экономии места. Оставляю их вам для самостоятельной работы.

Возможно, вы заметили еще одно обстоятельство: где определяются константы для кодов ошибок? Они находятся в классе `ArgsException` (листинг 14.7).

Листинг 14.7. ArgsException.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = null;
  private ErrorCode errorCode = OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}
```

```
public ArgsException(ErrorCode errorCode) {
  this.errorCode = errorCode;
}

public ArgsException(ErrorCode errorCode, String errorParameter) {
  this.errorCode = errorCode;
  this.errorParameter = errorParameter;
}

public ArgsException(ErrorCode errorCode,
                     char errorArgumentId, String errorParameter) {
  this.errorCode = errorCode;
  this.errorParameter = errorParameter;
  this.errorArgumentId = errorArgumentId;
}

public char getErrorArgumentId() {
  return errorArgumentId;
}

public void setErrorArgumentId(char errorArgumentId) {
  this.errorArgumentId = errorArgumentId;
}

public String getErrorParameter() {
  return errorParameter;
}

public void setErrorParameter(String errorParameter) {
  this.errorParameter = errorParameter;
}

public ErrorCode getErrorCode() {
  return errorCode;
}

public void setErrorCode(ErrorCode errorCode) {
  this.errorCode = errorCode;
}

public String errorMessage() {
  switch (errorCode) {
    case OK:
      return "TILT: Should not get here.";
    case UNEXPECTED_ARGUMENT:
      return String.format("Argument -%c unexpected.", errorArgumentId);
    case MISSING_STRING:
      return String.format("Could not find string parameter for -%c.",
                           errorArgumentId);
    case INVALID_INTEGER:
      return String.format("Argument -%c expects an integer but was '%s'.",
                           errorArgumentId, errorParameter);
```

продолжение ⇩

Листинг 14.7 (продолжение)

```
      case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
                             errorArgumentId);
      case INVALID_DOUBLE:
        return String.format("Argument -%c expects a double but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_DOUBLE:
        return String.format("Could not find double parameter for -%c.",
                             errorArgumentId);
      case INVALID_ARGUMENT_NAME:
        return String.format("'%c' is not a valid argument name.",
                             errorArgumentId);
      case INVALID_ARGUMENT_FORMAT:
        return String.format("'%s' is not a valid argument format.",
                             errorParameter);
    }
    return "";
  }

  public enum ErrorCode {
    OK, INVALID_ARGUMENT_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE}
}
```

Удивительно, какой объем кода понадобился для воплощения всех подробностей этой простой концепции. Одна из причин заключается в том, что мы используем весьма «многословный» язык. Поскольку Java относится к числу языков со статической типизацией, для удовлетворения требований системы типов в нем используется немалый объем кода. На таких языках, как Ruby, Python или Smalltalk, программа получится гораздо короче[1].

Пожалуйста, перечитайте код еще раз. Обратите особое внимание на выбор имен, размеры функций и форматирование кода. Возможно, опытные программисты найдут отдельные недочеты в стиле или структуре кода. Но я надеюсь, что в целом вы согласитесь с тем, что код хорошо написан, а его структура чиста и логична.

Скажем, после чтения кода вам должно быть очевидно, как добавить поддержку нового типа аргументов (например, дат или комплексных чисел), и это потребует относительно небольших усилий с вашей стороны. Для этого достаточно создать новый класс, производный от `ArgumentMarshaler`, новую функцию `getXXX` и включить новое условие `case` в функцию `parseSchemaElement`. Вероятно, также потребуется новое значение `ArgsException.ErrorCode` и новое сообщение об ошибке.

[1] Недавно я переписал этот модуль на Ruby. Код занимает в 7 раз меньше места и имеет более качественную структуру.

Как я это сделал?

Позвольте вас успокоить: я не написал эту программу от начала до конца в ее текущем виде. Более того, я не ожидаю, что вы сможете писать чистые и элегантные программы за один проход. Если мы чему-то и научились за последнюю пару десятилетий, так это тому, что программирование ближе к ремеслу, чем к науке. Чтобы написать чистый код, мы сначала пишем грязный код, а затем очищаем его.

Вряд ли вас это удивит. Мы усвоили эту истину еще в начальной школе, когда учителя заставляли нас (обычно безуспешно) писать планы сочинений. Предполагалось, что мы должны сначала написать первый вариант плана, затем второй, потом еще несколько версий, пока не придем к окончательной версии. Они пытались объяснить нам, что четкое и ясное сочинение появляется в результате последовательного усовершенствования.

Многие начинающие программисты (впрочем, как и большинство школьников, пишущих сочинения) не слишком усердно следуют этому совету. Они считают, что их главная цель — заставить программу работать. Когда программа «заработает», они переходят к следующей задаче, оставляя «работающую» программу в том состоянии, в котором она «заработала». Опытные программисты знают, что с профессиональной точки зрения такой подход равносилен самоубийству.

Args: черновик

В листинге 14.8 приведена более ранняя версия класса `Args`. Она «работает». И при этом выглядит крайне неряшливо.

Листинг 14.8. Args.java (первая версия)

```
import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private Map<Character, String> stringArgs = new HashMap<Character, String>();
  private Map<Character, Integer> intArgs = new HashMap<Character, Integer>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}
```

продолжение ⇩

Листинг 14.8 (продолжение)

```
public Args(String schema, String[] args) throws ParseException {
  this.schema = schema;
  this.args = args;
  valid = parse();
}

private boolean parse() throws ParseException {
  if (schema.length() == 0 && args.length == 0)
    return true;
  parseSchema();
  try {
    parseArguments();
  } catch (ArgsException e) {
  }
  return valid;
}

private boolean parseSchema() throws ParseException {
  for (String element : schema.split(",")) {
    if (element.length() > 0) {
      String trimmedElement = element.trim();
      parseSchemaElement(trimmedElement);
    }
  }
  return true;
}

private void parseSchemaElement(String element) throws ParseException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (isBooleanSchemaElement(elementTail))
    parseBooleanSchemaElement(elementId);
  else if (isStringSchemaElement(elementTail))
    parseStringSchemaElement(elementId);
  else if (isIntegerSchemaElement(elementTail)) {
    parseIntegerSchemaElement(elementId);
  } else {
    throw new ParseException(
      String.format("Argument: %c has invalid format: %s.",
                    elementId, elementTail), 0);
  }
}

private void validateSchemaElementId(char elementId) throws ParseException {
  if (!Character.isLetter(elementId)) {
    throw new ParseException(
      "Bad character:" + elementId + "in Args format: " + schema, 0);
  }
}
```

```
  private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, false);
  }

  private void parseIntegerSchemaElement(char elementId) {
    intArgs.put(elementId, 0);
  }

  private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, "");
  }

  private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
  }

  private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
  }

  private boolean isIntegerSchemaElement(String elementTail) {
    return elementTail.equals("#");
  }

  private boolean parseArguments() throws ArgsException {
for (currentArgument = 0; currentArgument < args.length; currentArgument++)
    {
      String arg = args[currentArgument];
      parseArgument(arg);
    }
    return true;
  }

  private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      unexpectedArguments.add(argChar);
      errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
      valid = false;
    }
  }
```

продолжение ⇘

Листинг 14.8 (продолжение)

```
private boolean setArgument(char argChar) throws ArgsException {
  if (isBooleanArg(argChar))
    setBooleanArg(argChar, true);
  else if (isStringArg(argChar))
    setStringArg(argChar);
  else if (isIntArg(argChar))
    setIntArg(argChar);
  else
    return false;

  return true;
}

private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

private void setIntArg(char argChar) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    intArgs.put(argChar, new Integer(parameter));
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
  } catch (NumberFormatException e) {
    valid = false;
    errorArgumentId = argChar;
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw new ArgsException();
  }
}

private void setStringArg(char argChar) throws ArgsException {
  currentArgument++;
  try {
    stringArgs.put(argChar, args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}

private boolean isStringArg(char argChar) {
  return stringArgs.containsKey(argChar);
}
```

```
private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.put(argChar, value);
}

private boolean isBooleanArg(char argChar) {
  return booleanArgs.containsKey(argChar);
}

public int cardinality() {
  return argsFound.size();
}

public String usage() {
  if (schema.length() > 0)
    return "-[" + schema + "]";
  else
    return "";
}

public String errorMessage() throws Exception {
  switch (errorCode) {
    case OK:
      throw new Exception("TILT: Should not get here.");
    case UNEXPECTED_ARGUMENT:
      return unexpectedArgumentMessage();
    case MISSING_STRING:
      return String.format("Could not find string parameter for -%c.",
                           errorArgumentId);
    case INVALID_INTEGER:
      return String.format("Argument -%c expects an integer but was '%s'.",
                           errorArgumentId, errorParameter);
    case MISSING_INTEGER:
      return String.format("Could not find integer parameter for -%c.",
                           errorArgumentId);
  }
  return "";
}

private String unexpectedArgumentMessage() {
  StringBuffer message = new StringBuffer("Argument(s) -");
  for (char c : unexpectedArguments) {
    message.append(c);
  }
  message.append(" unexpected.");
  return message.toString();
}

private boolean falseIfNull(Boolean b) {
  return b != null && b;
}
```

продолжение ⇩

Листинг 14.8 (продолжение)

```
  private int zeroIfNull(Integer i) {
    return i == null ? 0 : i;
  }

  private String blankIfNull(String s) {
    return s == null ? "" : s;
  }

  public String getString(char arg) {
    return blankIfNull(stringArgs.get(arg));
  }

  public int getInt(char arg) {
    return zeroIfNull(intArgs.get(arg));
  }

  public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg));
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public boolean isValid() {
    return valid;
  }

  private class ArgsException extends Exception {
  }
}
```

Надеюсь, при виде этой глыбы кода вам захотелось сказать: «Как хорошо, что она не осталась в таком виде!» Если вы почувствовали нечто подобное, вспомните, что другие люди чувствуют то же самое при виде вашего кода, оставшегося на стадии «черновика».

Вообще говоря, «черновик» — самое мягкое, что можно сказать об этом коде. Очевидно, что перед нами незавершенная работа. От одного количества переменных экземпляров можно прийти в ужас. Загадочные строки вроде `"TILT"`, контейнеры `HashSet` и `TreeSet`, конструкции `try-catch-catch` только увеличивают масштабы этого беспорядочного месива.

Я вовсе не собирался писать беспорядочное месиво. В самом деле, я постарался сохранить более или менее разумную организацию кода. Об этом свидетельствует хотя бы выбор имен функций и переменных, а также наличие у программы примитивной структуры. Но совершенно очевидно, что проблемы вышли из-под моего контроля.

Неразбериха накапливалась постепенно. Ранние версии выглядели вовсе не так отвратительно. Для примера в листинге 14.9 приведена начальная версия, поддерживающая только логические аргументы.

Листинг 14.9. Args.java (только Boolean)

```
package com.objectmentor.utilities.getopts;

import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private int numberOfArguments = 0;

  public Args(String schema, String[] args) {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  public boolean isValid() {
    return valid;
  }

  private boolean parse() {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    parseArguments();
    return unexpectedArguments.size() == 0;
  }

  private boolean parseSchema() {
    for (String element : schema.split(",")) {
      parseSchemaElement(element);
    }
    return true;
  }

  private void parseSchemaElement(String element) {
    if (element.length() == 1) {
      parseBooleanSchemaElement(element);
    }
  }
```

продолжение ⇩

Листинг 14.9 (продолжение)

```
private void parseBooleanSchemaElement(String element) {
  char c = element.charAt(0);
  if (Character.isLetter(c)) {
    booleanArgs.put(c, false);
  }
}

private boolean parseArguments() {
  for (String arg : args)
    parseArgument(arg);
  return true;
}

private void parseArgument(String arg) {
  if (arg.startsWith("-"))
    parseElements(arg);
}

private void parseElements(String arg) {
  for (int i = 1; i < arg.length(); i++)
    parseElement(arg.charAt(i));
}

private void parseElement(char argChar) {
  if (isBoolean(argChar)) {
    numberOfArguments++;
    setBooleanArg(argChar, true);
  } else
    unexpectedArguments.add(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.put(argChar, value);
}

private boolean isBoolean(char argChar) {
  return booleanArgs.containsKey(argChar);
}

public int cardinality() {
  return numberOfArguments;
}

public String usage() {
  if (schema.length() > 0)
     return "-["+schema+"]";
  else
    return "";
}
```

```
  public String errorMessage() {
    if (unexpectedArguments.size() > 0) {
      return unexpectedArgumentMessage();
    } else
      return "";
  }

  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
      message.append(c);
    }
    message.append(" unexpected.");

    return message.toString();
  }

  public boolean getBoolean(char arg) {
    return booleanArgs.get(arg);
  }
}
```

В этом коде можно найти множество недостатков, однако в целом он не так уж плох. Код компактен и прост, в нем легко разобраться. Тем не менее в этом коде легко прослеживаются зачатки будущего беспорядочного месива. Нетрудно понять, как из него выросла вся последующая неразбериха.

Обратите внимание: в последующей неразберихе добавились всего два новых типа аргументов, `String` и `integer`. Добавление всего двух типов аргументов имело огромные отрицательные последствия для кода. Более или менее понятный код превратился в запутанный клубок, наверняка кишащий множеством ошибок и недочетов.

Два новых типа аргументов добавлялись последовательно. Сначала я добавил поддержку `String`, что привело к следующему результату.

Листинг 14.10. Args.java (Boolean и String)

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private Map<Character, String> stringArgs =
```

продолжение ⇩

Листинг 14.10 (продолжение)

```
    new HashMap<Character, String>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgument = '\0';

  enum ErrorCode {
    OK, MISSING_STRING}

  private ErrorCode errorCode = ErrorCode.OK;

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    parseArguments();
    return valid;
  }

  private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        String trimmedElement = element.trim();
        parseSchemaElement(trimmedElement);
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      parseBooleanSchemaElement(elementId);
    else if (isStringSchemaElement(elementTail))
      parseStringSchemaElement(elementId);
  }

  private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
      throw new ParseException(
        "Bad character:" + elementId + "in Args format: " + schema, 0);
    }

  }
```

```
  private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, "");
  }

  private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
  }

  private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
  }

  private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, false);
  }

  private boolean parseArguments() {
for (currentArgument = 0; currentArgument < args.length; currentArgument++)
    {
      String arg = args[currentArgument];
      parseArgument(arg);
    }
    return true;
  }

  private void parseArgument(String arg) {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      unexpectedArguments.add(argChar);
      valid = false;
    }
  }

  private boolean setArgument(char argChar) {
    boolean set = true;
    if (isBoolean(argChar))
      setBooleanArg(argChar, true);
    else if (isString(argChar))
      setStringArg(argChar, "");
    else
```

продолжение ⇩

Листинг 14.10 (продолжение)

```
      set = false;
    return set;
  }

  private void setStringArg(char argChar, String s) {
    currentArgument++;
    try {
      stringArgs.put(argChar, args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgument = argChar;
      errorCode = ErrorCode.MISSING_STRING;
    }
  }

  private boolean isString(char argChar) {
    return stringArgs.containsKey(argChar);
  }

  private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
  }

  private boolean isBoolean(char argChar) {
    return booleanArgs.containsKey(argChar);
  }

  public int cardinality() {
    return argsFound.size();
  }

  public String usage() {
    if (schema.length() > 0)
      return "-[" + schema + "]";
    else
      return "";
  }

  public String errorMessage() throws Exception {
    if (unexpectedArguments.size() > 0) {
      return unexpectedArgumentMessage();
    } else
      switch (errorCode) {
        case MISSING_STRING:
          return String.format("Could not find string parameter for -%c.",
                               errorArgument);
        case OK:
          throw new Exception("TILT: Should not get here.");
      }
    return "";
  }
```

```
  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
      message.append(c);
    }
    message.append(" unexpected.");
    return message.toString();
  }

  public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg));
  }

  private boolean falseIfNull(Boolean b) {
    return b == null ? false : b;
  }

  public String getString(char arg) {
    return blankIfNull(stringArgs.get(arg));
  }

  private String blankIfNull(String s) {
    return s == null ? "" : s;
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public boolean isValid() {
    return valid;
  }
}
```

Ситуация явно выходит из-под контроля. Код все еще не ужасен, но путаница очевидно растет. Это уже клубок, хотя и не беспорядочное месиво. А чтобы месиво забродило и стало подниматься, хватило простого добавления целочисленных аргументов.

На этом я остановился

Мне предстояло добавить еще два типа аргументов. Было совершенно очевидно, что с ними все станет намного хуже. Если бы я с упорством бульдозера пошел вперед, скорее всего, мне удалось бы заставить программу работать, но разобраться в получившемся коде не удалось бы уже никому. Если я хотел, чтобы с моим кодом можно было работать, спасать положение нужно было именно сейчас.

Итак, я прекратил добавлять в программу новые возможности и взялся за переработку. После добавления типов `String` и `integer` я знал, что для каждого типа аргументов новый код должен добавляться в трех основных местах. Во-первых, для

каждого типа аргументов необходимо было обеспечить разбор соответствующего элемента форматной строки, чтобы выбрать объект HashMap для этого типа. Затем аргумент соответствующего типа необходимо было разобрать в командной строке и преобразовать к истинному типу. Наконец, для каждого типа аргументов требовался метод getXXX, возвращающий значение аргумента с его истинным типом.

Много разных типов, обладающих сходными методами... Наводит на мысли о классе. Так родилась концепция ArgumentMarshaler.

О постепенном усовершенствовании

Один из верных способов убить программу — вносить глобальные изменения в ее структуру с целью улучшения. Некоторые программы уже никогда не приходят в себя после таких «усовершенствований». Проблема в том, что код очень трудно заставить работать так же, как он работал до «усовершенствования».

Чтобы этого не произошло, я воспользовался методологией разработки через тестирование (TDD). Одна из центральных доктрин этой методологии гласит, что система должна работать в любой момент в процессе внесения изменений. Иначе говоря, при использовании TDD запрещено вносить в систему изменения, нарушающие работоспособность этой системы. С каждым вносимым изменением система должна работать так же, как она работала прежде.

Для этого был необходим пакет автоматизированных тестов. Запуская их в любой момент времени, я мог бы убедиться в том, что поведение системы осталось неизменным. Я уже создал пакет модульных и приемочных тестов для класса Args, пока работал над начальной версией (она же «беспорядочное месиво»). Модульные тесты были написаны на Java и находились под управлением JUnit. Приемочные тесты были оформлены в виде вики-страниц в FitNesse. Я мог запустить эти тесты в любой момент по своему усмотрению, и если они проходили — можно было не сомневаться в том, что система работает именно так, как положено.

И тогда я занялся внесением множества очень маленьких изменений. Каждое изменение продвигало структуру системы к концепции ArgumentMarshaler, но после каждого изменения система продолжала нормально работать. На первом этапе я добавил заготовку ArgumentMarshaller в конец месива (листинг 14.11).

Листинг 14.11. Класс ArgumentMarshaller, присоединенный к Args.java

```
private class ArgumentMarshaler {
    private boolean booleanValue = false;

    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {return booleanValue;}
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
```

```
  }
  private class StringArgumentMarshaler extends ArgumentMarshaler {
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  }
}
```

Понятно, что добавление класса ничего не нарушит. Поэтому я внес самое простейшее из всех возможных изменений — изменил контейнер `HashMap` для логических аргументов так, чтобы при конструировании передавался тип `ArgumentMarshaler`:

```
private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgumentMarshaler>();
```

Это нарушило работу нескольких команд, которые я быстро исправил.

```
...
private void parseBooleanSchemaElement(char elementId) {
  booleanArgs.put(elementId, new BooleanArgumentMarshaler());
}
..
private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.get(argChar).setBoolean(value);
}
...
public boolean getBoolean(char arg) {
  return falseIfNull(booleanArgs.get(arg).getBoolean());
}
```

Изменения вносятся в тех местах, о которых я упоминал ранее: методы `parse`, `set` и `get` для типа аргумента. К сожалению, при всей незначительности изменений некоторые тесты стали завершаться неудачей. Внимательно присмотревшись к `getBoolean`, вы увидите, что если при вызове метода с `'y'` аргумента `y` не существует, вызов `booleanArgs.get('y')` вернет `null`, а функция выдаст исключение `NullPointerException`. Функция `falseIfNull` защищала от подобных ситуаций, но в результате внесенных изменений она перестала работать.

Стратегия постепенных изменений требовала, чтобы я немедленно наладил работу программы, прежде чем вносить какие-либо дополнительные изменения. Действительно, проблема решалась просто: нужно было добавить проверку `null`. Но на этот раз проверять нужно было не логическое значение, а `ArgumentMarshaller`.

Сначала я убрал вызов `falseIfNull` из `getBoolean`. Функция `falseIfNull` стала бесполезной, поэтому я убрал и саму функцию. Тесты все равно не проходили, поэтому я был уверен, что новых ошибок от этого уже не прибавится.

```
public boolean getBoolean(char arg) {
    return booleanArgs.get(arg).getBoolean();
  }
```

Затем я разбил функцию `getBoolean` надвое и разместил `ArgumentMarshaller` в собственной переменной с именем `argumentMarshaller`. Длинное имя мне не понра-

вилось; во-первых, оно было избыточным, а во-вторых, загромождало функцию. Соответственно я сократил его до `am` [N5].

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am.getBoolean();
}
```

Наконец, я добавил логику проверки `null`:

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am != null && am.getBoolean();
}
```

Аргументы String

Добавление поддержки `String` было очень похоже на добавление поддержки `Boolean`. Мне предстояло изменить `HashMap` и заставить работать функции `parse`, `set` и `get`. Полагаю, следующий код понятен без пояснений — если не считать того, что я разместил всю реализацию компоновки аргументов в базовом клссе `ArgumentMarshaller`, вместо того чтобы распределять ее по производным классам.

```
private Map<Character, ArgumentMarshaler> stringArgs =
      new HashMap<Character, ArgumentMarshaler>();
...
private void parseStringSchemaElement(char elementId) {
  stringArgs.put(elementId, new StringArgumentMarshaler());
}
...
private void setStringArg(char argChar) throws ArgsException {
  currentArgument++;
  try {
    stringArgs.get(argChar).setString(args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}
...
public String getString(char arg) {
  Args.ArgumentMarshaler am = stringArgs.get(arg);
  return am == null ? "" : am.getString();
}
...
private class ArgumentMarshaler {
  private boolean booleanValue = false;
  private String stringValue;

  public void setBoolean(boolean value) {
    booleanValue = value;
  }
```

```
  public boolean getBoolean() {
    return booleanValue;
  }

  public void setString(String s) {
    stringValue = s;
  }

  public String getString() {
    return stringValue == null ? "" : stringValue;
  }
}
```

И снова изменения вносились последовательно и только так, чтобы тесты по крайней мере хотя бы запускались (даже если и не проходили). Если работоспособность теста была нарушена, я сначала добивался того, чтобы он работал, и только потом переходил к следующему изменению.

Вероятно, вы уже поняли, что я собираюсь сделать. Собрав все текущее поведение компоновки аргументов в базовом классе `ArgumentMarshaler`, я намерен перемещать его вниз в производные классы. Это позволит мне сохранить работоспособность программы в ходе постепенного изменения ее структуры.

Очевидным следующим шагом стало перемещение функциональности аргумента `int` в `ArgumentMarshaler`. И снова все обошлось без сюрпризов:

```
private Map<Character, ArgumentMarshaler> intArgs =
     new HashMap<Character, ArgumentMarshaler>();
...
private void parseIntegerSchemaElement(char elementId) {
  intArgs.put(elementId, new IntegerArgumentMarshaler());
}
...
private void setIntArg(char argChar) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    intArgs.get(argChar).setInteger(Integer.parseInt(parameter));
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
  } catch (NumberFormatException e) {
    valid = false;
    errorArgumentId = argChar;
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw new ArgsException();
  }
}
...
public int getInt(char arg) {
```

```
    Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : am.getInteger();
  }
  ...
  private class ArgumentMarshaler {
    private boolean booleanValue = false;
    private String stringValue;
    private int integerValue;

    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {
      return booleanValue;
    }

    public void setString(String s) {
      stringValue = s;
    }

    public String getString() {
      return stringValue == null ? "" : stringValue;
    }

    public void setInteger(int i) {
      integerValue = i;
    }

    public int getInteger() {
      return integerValue;
    }
  }
```

Переместив всю логику компоновки аргументов в `ArgumentMarshaler`, я занялся перемещением функциональности в производные классы. На первом этапе я должен был переместить функцию `setBoolean` в `BooleanArgumentMarshaller` и позаботиться о том, чтобы она правильно вызывалась. Для этого был создан абстрактный метод `set`.

```
private abstract class ArgumentMarshaler {
  protected boolean booleanValue = false;
  private String stringValue;
  private int integerValue;

  public void setBoolean(boolean value) {
    booleanValue = value;
  }
  public boolean getBoolean() {
```

```
    return booleanValue;
  }

  public void setString(String s) {
    stringValue = s;
  }

  public String getString() {
    return stringValue == null ? "" : stringValue;
  }

  public void setInteger(int i) {
    integerValue = i;
  }

  public int getInteger() {
    return integerValue;
  }

  public abstract void set(String s);
}
```

Затем метод set был реализован в BooleanArgumentMarshaller.

```
private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  public void set(String s) {
    booleanValue = true;
  }
}
```

Наконец, вызов setBoolean был заменен вызовом set.

```
private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.get(argChar).set("true");
}
```

Все тесты прошли успешно. Так как изменения привели к перемещению set в BooleanArgumentMarshaler, я удалил метод setBoolean из базового класса ArgumentMarshaler.

Обратите внимание: абстрактная функция set получает аргумент String, но реализация в классе BooleanArgumentMarshaller его не использует. Я добавил этот аргумент, потому что знал, что он *будет* использоваться классами StringArgumentMarshaller и IntegerArgumentMarshaller.

На следующем шаге я решил разместить метод get в BooleanArgumentMarshaler. Подобные размещения get всегда выглядят уродливо, потому что фактически возвращается тип Object, который в данном случае приходится преобразовывать в Boolean.

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am != null && (Boolean)am.get();
}
```

Просто для того, чтобы программа компилировалась, я добавил в ArgumentMarshaler функцию get.

```
private abstract class ArgumentMarshaler {
   ...
  public Object get() {
    return null;
  }
}
```

Программа компилировалась, а тесты, разумеется, не проходили. Чтобы тесты снова заработали, достаточно объявить метод get абстрактным и реализовать его в BooleanAgumentMarshaler.

```
private abstract class ArgumentMarshaler {
  protected boolean booleanValue = false;
    ...
    public abstract Object get();
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    public void set(String s) {
      booleanValue = true;
    }

    public Object get() {
      return booleanValue;
    }
  }
```

Итак, тесты снова проходят успешно. Теперь оба метода get и set размещаются в BooleanArgumentMarshaler! Это позволило мне удалить старую функцию getBoolean из ArgumentMarshaler, переместить защищенную переменную booleanValue в BooleanArgumentMarshaler и объявить ее приватной.

Аналогичные изменения были внесены для типа String. Я реализовал методы set и get, удалил ненужные функции и переместил переменные.

```
private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
      stringArgs.get(argChar).set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
...
  public String getString(char arg) {
    Args.ArgumentMarshaler am = stringArgs.get(arg);
    return am == null ? "" : (String) am.get();
  }
```

```
...
  private abstract class ArgumentMarshaler {
    private int integerValue;

    public void setInteger(int i) {
      integerValue = i;
    }

    public int getInteger() {
      return integerValue;
    }

    public abstract void set(String s);

    public abstract Object get();
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set(String s) {
      booleanValue = true;
    }

    public Object get() {
      return booleanValue;
    }
  }

  private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";

    public void set(String s) {
      stringValue = s;
    }

    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {

    public void set(String s) {
    }

    public Object get() {
      return null;
    }
  }
}
```

Осталось лишь повторить этот процесс для integer. На этот раз задача немного усложняется: целые числа необходимо разбирать, а в ходе разбора возможны исключения. Но внешний вид кода улучшается тем, что вся концепция NumberFormatException скрыта в классе IntegerArgumentMarshaler.

```
private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

  private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      intArgs.get(argChar).set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }
...
  private void setBooleanArg(char argChar) {
    try {
      booleanArgs.get(argChar).set("true");
    } catch (ArgsException e) {
    }
  }
...
  public int getInt(char arg) {
    Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : (Integer) am.get();
  }
...
  private abstract class ArgumentMarshaler {
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
  }
...
  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(String s) throws ArgsException {
      try {
        intValue = Integer.parseInt(s);
      } catch (NumberFormatException e) {
```

```
        throw new ArgsException();
      }
    }
    public Object get() {
      return intValue;
    }
  }
```

Конечно, тесты по-прежнему проходили. Далее я избавился от трех разновидностей Map в начале алгоритма, отчего система стала намного более универсальной. Впрочем, я не мог их просто удалить, поскольку это нарушило бы работу системы. Вместо этого я добавил новый объект Map для ArgumentMarshaler, а затем последовательно изменял методы, чтобы они использовали этот объект вместо трех исходных.

```
public class Args {
...
  private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
...
  private void parseBooleanSchemaElement(char elementId) {
    ArgumentMarshaler m = new BooleanArgumentMarshaler();
    booleanArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }

  private void parseIntegerSchemaElement(char elementId) {
    ArgumentMarshaler m = new IntegerArgumentMarshaler();
    intArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }

  private void parseStringSchemaElement(char elementId) {
    ArgumentMarshaler m = new StringArgumentMarshaler();
    stringArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }
```

Разумеется, тесты проходили успешно. Далее я привел метод isBooleanArg:

```
private boolean isBooleanArg(char argChar) {
  return booleanArgs.containsKey(argChar);
}
```

к следующему виду:

```
private boolean isBooleanArg(char argChar) {
  ArgumentMarshaler m = marshalers.get(argChar);
```

```
    return m instanceof BooleanArgumentMarshaler;
  }
```

Тесты по-прежнему проходят. Я внес аналогичные изменения в `isIntArg` и `isStringArg`.

```
  private boolean isIntArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof IntegerArgumentMarshaler;
  }

  private boolean isStringArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof StringArgumentMarshaler;
  }
```

Тесты проходят. Я удалил все повторяющиеся вызовы `marshalers.get`:

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (isBooleanArg(m))
      setBooleanArg(argChar);
    else if (isStringArg(m))
      setStringArg(argChar);
    else if (isIntArg(m))
      setIntArg(argChar);
    else
      return false;

    return true;
  }

  private boolean isIntArg(ArgumentMarshaler m) {
    return m instanceof IntegerArgumentMarshaler;
  }

  private boolean isStringArg(ArgumentMarshaler m) {
    return m instanceof StringArgumentMarshaler;
  }

  private boolean isBooleanArg(ArgumentMarshaler m) {
    return m instanceof BooleanArgumentMarshaler;
  }
```

Причин для существования трех методов `isxxxArg` не осталось. Я оформил их в виде встроенного кода:

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(argChar);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(argChar);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(argChar);
    else
```

```
    return false;
  return true;
}
```

На следующем шаге я перешел на использование ассоциативного массива marshalers в функциях set, отказываясь от использования трех старых контейнеров. Преобразование началось с Boolean:

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m instanceof BooleanArgumentMarshaler)
    setBooleanArg(m);
  else if (m instanceof StringArgumentMarshaler)
    setStringArg(argChar);
  else if (m instanceof IntegerArgumentMarshaler)
    setIntArg(argChar);
  else
    return false;

  return true;
}
...
private void setBooleanArg(ArgumentMarshaler m) {
  try {
    m.set("true"); // было: booleanArgs.get(argChar).set(«true»);
  } catch (ArgsException e) {
  }
}
```

Тесты проходили успешно, и я сделал то же самое для типов String и Integer. Это позволило мне интегрировать часть некрасивого кода обработки исключений в функцию setArgument.

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  try {
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(m);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);
    else
      return false;
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}

private void setIntArg(ArgumentMarshaler m) throws ArgsException {
  currentArgument++;
```

```
    String parameter = null;
    try {
      parameter = args[currentArgument];
      m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    try {
      m.set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
```

Я вплотную подошел к удалению трех старых объектов `Map`. Прежде всего было необходимо привести функцию `getBoolean`:

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am != null && (Boolean) am.get();
}
```

к следующему виду:

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  boolean b = false;
  try {
    b = am != null && (Boolean) am.get();
  } catch (ClassCastException e) {
    b = false;
  }
  return b;
}
```

Возможно, последнее изменение вас удивило. Почему я вдруг решил обрабатывать `ClassCastException`? Дело в том, что наряду с набором модульных тестов у меня был отдельный набор приемочных тестов, написанных для FitNesse. Оказалось, что тесты FitNesse проверяли, что при вызове `getBoolean` для аргумента с типом, отличным от `Boolean`, возвращается `false`. Модульные тесты этого не делали. До этого момента я запускал только модульные тесты[1].

[1] Чтобы предотвратить подобные сюрпризы в будущем, я добавил новый модульный тест, который запускал все тесты FitNesse.

Последнее изменение позволило исключить еще одну точку использования объекта Map для типа Boolean:

```
private void parseBooleanSchemaElement(char elementId) {
  ArgumentMarshaler m = new BooleanArgumentMarshaler();
- booleanArgs.put(elementId, m);
  marshalers.put(elementId, m);
}
```

Теперь объект Map для типа Boolean можно было удалить:

```
public class Args {
...
- private Map<Character, ArgumentMarshaler> booleanArgs =
- new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
...
```

Далее я проделал аналогичную процедуру для аргументов String и Integer и немного подчистил код:

```
  private void parseBooleanSchemaElement(char elementId) {
    marshalers.put(elementId, new BooleanArgumentMarshaler());
  }

  private void parseIntegerSchemaElement(char elementId) {
    marshalers.put(elementId, new IntegerArgumentMarshaler());
  }
  private void parseStringSchemaElement(char elementId) {
    marshalers.put(elementId, new StringArgumentMarshaler());
  }
...
  public String getString(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
       return "";
    }
  }
...
public class Args {
...
- private Map<Character, ArgumentMarshaler> stringArgs =
-    new HashMap<Character, ArgumentMarshaler>();
- private Map<Character, ArgumentMarshaler> intArgs =
-    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
...
```

Затем я подставил в parseSchemaElement код трех методов parse, сократившихся до одной команды:

```
private void parseSchemaElement(String element) throws ParseException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (isBooleanSchemaElement(elementTail))
    marshalers.put(elementId, new BooleanArgumentMarshaler());
  else if (isStringSchemaElement(elementTail))
    marshalers.put(elementId, new StringArgumentMarshaler());
  else if (isIntegerSchemaElement(elementTail)) {
    marshalers.put(elementId, new IntegerArgumentMarshaler());
  } else {
    throw new ParseException(String.format(
      "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
  }
}
```

Давайте взглянем на общую картину. В листинге 14.12 представлена текущая форма класса Args.

Листинг 14.12. Args.java (после первой переработки)

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, ArgumentMarshaler> marshalers =
new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";

  private ErrorCode errorCode = ErrorCode.OK;
  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}
  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
```

```
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }

  private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        String trimmedElement = element.trim();
        parseSchemaElement(trimmedElement);
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (isStringSchemaElement(elementTail))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (isIntegerSchemaElement(elementTail)) {
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    } else {
      throw new ParseException(String.format(
    "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
    }
  }

  private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
      throw new ParseException(
    "Bad character:" + elementId + "in Args format: " + schema, 0);
    }
  }

  private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
  }

  private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
  }

  private boolean isIntegerSchemaElement(String elementTail) {
    return elementTail.equals("#");
  }
```

продолжение ⇩

Листинг 14.12 (продолжение)

```
private boolean parseArguments() throws ArgsException {
  for (currentArgument=0; currentArgument<args.length; currentArgument++) {
    String arg = args[currentArgument];
    parseArgument(arg);
  }
  return true;
}

private void parseArgument(String arg) throws ArgsException {
  if (arg.startsWith("-"))
    parseElements(arg);
}

private void parseElements(String arg) throws ArgsException {
  for (int i = 1; i < arg.length(); i++)
    parseElement(arg.charAt(i));
}

private void parseElement(char argChar) throws ArgsException {
  if (setArgument(argChar))
    argsFound.add(argChar);
  else {
    unexpectedArguments.add(argChar);
    errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
    valid = false;
  }
}

private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  try {
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(m);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);
    else
      return false;
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}

private void setIntArg(ArgumentMarshaler m) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
```

```
      parameter = args[currentArgument];
      m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    try {
      m.set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }

  private void setBooleanArg(ArgumentMarshaler m) {
    try {
      m.set("true");
    } catch (ArgsException e) {
    }
  }

  public int cardinality() {
    return argsFound.size();
  }

  public String usage() {
    if (schema.length() > 0)
      return "-[" + schema + "]";
    else
      return "";
  }

  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Should not get here.");
      case UNEXPECTED_ARGUMENT:
        return unexpectedArgumentMessage();
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
                             errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer but was '%s'.",
                             errorArgumentId, errorParameter);
```

продолжение ⇩

Листинг 14.12 (продолжение)

```
      case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
                             errorArgumentId);
    }
    return "";
  }

  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
      message.append(c);
    }
    message.append(" unexpected.");
    return message.toString();
  }

  public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
      b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
      b = false;
    }
    return b;
  }

  public String getString(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
      return "";
    }
  }

  public int getInt(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
      return 0;
    }
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public boolean isValid() {
    return valid;
  }
```

```
  private class ArgsException extends Exception {
  }

  private abstract class ArgumentMarshaler {
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;
    public void set(String s) {
      booleanValue = true;
    }
    public Object get() {
      return booleanValue;
    }
  }

  private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";
    public void set(String s) {
      stringValue = s;
    }
    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;
    public void set(String s) throws ArgsException {
      try {
        intValue = Integer.parseInt(s);
      } catch (NumberFormatException e) {
        throw new ArgsException();
      }
    }
    public Object get() {
      return intValue;
    }
  }
}
```

Вроде бы проделана большая работа, а результат не впечатляет. Структура кода немного улучшилась, но в начале листинга по-прежнему объявляются многочисленные переменные; в `setArgument` осталась кошмарная конструкция проверки типа; функции `set` выглядят просто ужасно. Я уже не говорю об обработке ошибок... Нам еще предстоит большая работа.

Прежде всего хотелось бы избавиться от конструкции выбора в `setArgument` [G23]. В идеале она должна быть заменена единственным вызовом `ArgumentMarshaler.set`. Это означает, что код `setIntArg`, `setStringArg` и `setBooleanArg` должен быть пере-

мещен в соответствующие классы, производные от ArgumentMarshaler. Однако при этом возникает одна проблема.

Внимательно присмотревшись к функции setIntArg, можно заметить, что в ней используются две переменные экземпляров: args и currentArg. Чтобы переместить setIntArg в BooleanArgumentMarshaler, мне придется передать args и currentArgs в аргументах при вызове. Решение получается «грязным» [F1]. Я бы предпочел передать один аргумент вместо двух. К счастью, у проблемы существует простое решение: мы можем преобразовать массив args в list и передать Iterator функциям set. Следующее преобразование было проведено за десять шагов, с обязательным выполнением всех тестов после каждого шага. Здесь я приведу только конечный результат, но вы легко сможете опознать большинство промежуточных шагов по этому листингу.

```
public class Args {
  private String schema;
- private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, ArgumentMarshaler> marshalers =
new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private Iterator<String> currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;
  private List<String> argsList;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && argsList.size() == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }
---
  private boolean parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
      String arg = currentArgument.next();
```

```
      parseArgument(arg);
    }

    return true;
  }
---
  private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      m.set(parameter);
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    try {
      m.set(currentArgument.next());
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
```

Все изменения были простыми и не нарушали работы тестов. Теперь можно заняться перемещением функций в соответствующие производные классы. Начнем с внесения изменений в setArgument:

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(m);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);
    else
      return false;
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}
```

Это изменение важно, потому что мы хотим полностью устранить цепочку if-else. Для этого из нее необходимо вывести состояние ошибки.

Теперь можно переходить к перемещению функций set. Функция setBooleanArg тривиальна, поэтому начнем с нее. Наша задача — изменить функцию setBooleanArg так, чтобы она просто передавала управление BooleanArgumentMarshaler.

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
      if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m, currentArgument);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);

    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }
---
  private void setBooleanArg(ArgumentMarshaler m,
                             Iterator<String> currentArgument)
                             throws ArgsException {
--- try {
      m.set("true");
--- catch (ArgsException e) {
--- }
 }
```

Но ведь мы только что перенесли обработку исключения в функцию? Ситуация с включением того, что вы намереваетесь вскоре исключить, весьма часто встречается при переработке кода. Малый размер шагов и необходимость прохождения тестов означает, что вам придется часто перемещать туда-сюда фрагменты кода. Переработка кода напоминает кубик Рубика: чтобы добиться большой цели, необходимо выполнить множество мелких операций. Каждая операция делает возможной следующую.

Зачем передавать итератор, если setBooleanArg он не нужен? Потому что он нужен setIntArg и setStringArg! И если я хочу организовать доступ ко всем трем функциям через абстрактный метод в ArgumentMarshaller, мне не обойтись без его передачи setBooleanArg.

Итак, функция setBooleanArg стала бесполезной. Если бы в ArgumentMarshaler присутствовала функция set, то мы могли бы вызвать ее напрямую. Значит, нужно создать такую функцию! Первым шагом станет включение нового абстрактного метода в ArgumentMarshaler.

```
private abstract class ArgumentMarshaler {
  public abstract void set(Iterator<String> currentArgument)
                      throws ArgsException;
  public abstract void set(String s) throws ArgsException;
  public abstract Object get();
}
```

Конечно, это нарушает работу всех производных классов, поэтому мы добавим реализацию нового метода в каждый из них.

```
private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  private boolean booleanValue = false;

  public void set(Iterator<String> currentArgument) throws ArgsException {
  booleanValue = true;
  }

  public void set(String s) {
    booleanValue = true;
  }

  public Object get() {
    return booleanValue;
  }
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
  }

  public void set(String s) {
    stringValue = s;
  }

  public Object get() {
    return stringValue;
  }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
  }

  public void set(String s) throws ArgsException {
    try {
      intValue = Integer.parseInt(s);
    } catch (NumberFormatException e) {
      throw new ArgsException();
    }
  }
```

```
  public Object get() {
    return intValue;
  }
}
```

А теперь setBooleanArg можно удалить!

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    if (m instanceof BooleanArgumentMarshaler)
      m.set(currentArgument);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);

  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}
```

Все тесты проходят, а функция set размещается в BooleanArgumentMarshaler! Теперь можно сделать то же самое для String and Integer.

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    if (m instanceof BooleanArgumentMarshaler)
      m.set(currentArgument);
    else if (m instanceof StringArgumentMarshaler)
      m.set(currentArgument);
    else if (m instanceof IntegerArgumentMarshaler)
      m.set(currentArgument);

  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}
---
private class StringArgumentMarshaler extends ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
```

```
    try {
      stringValue = currentArgument.next();
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }

  public void set(String s) {
  }

  public Object get() {
    return stringValue;
  }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0;
  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      set(parameter);
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  public void set(String s) throws ArgsException {
    try {
      intValue = Integer.parseInt(s);
    } catch (NumberFormatException e) {
      throw new ArgsException();
    }
  }

  public Object get() {
    return intValue;
  }
}
```

А теперь завершающий штрих: убираем цепочку `if-else`!

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
```

```
    m.set(currentArgument);
    return true;
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
}
```

Избавляемся от лишних функций в `IntegerArgumentMarshaler` и слегка чистим код:

```
private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0

    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        intValue = Integer.parseInt(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }
 }
```

`ArgumentMarshaler` преобразуется в интерфейс:

```
private interface ArgumentMarshaler {
  void set(Iterator<String> currentArgument) throws ArgsException;
  Object get();
}
```

А теперь посмотрите, как легко добавлять новые типы аргументов в эту структуру. Количество изменений минимально, а все изменения логически изолированы. Начнем с добавления нового тестового сценария, проверяющего правильность работы аргументов `double`:

```
public void testSimpleDoublePresent() throws Exception {
  Args args = new Args("x##", new String[] {"-x","42.3"});
  assertTrue(args.isValid());
  assertEquals(1, args.cardinality());
  assertTrue(args.has('x'));
  assertEquals(42.3, args.getDouble('x'), .001);
}
```

Чистим код разбора форматной строки и добавляем обнаружение ## для аргументов типа double.

```
private void parseSchemaElement(String element) throws ParseException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (elementTail.length() == 0)
    marshalers.put(elementId, new BooleanArgumentMarshaler());
  else if (elementTail.equals("*"))
    marshalers.put(elementId, new StringArgumentMarshaler());
  else if (elementTail.equals("#"))
    marshalers.put(elementId, new IntegerArgumentMarshaler());
  else if (elementTail.equals("##"))
    marshalers.put(elementId, new DoubleArgumentMarshaler());
  else
    throw new ParseException(String.format(
      "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
}
```

Затем пишется класс DoubleArgumentMarshaler.

```
private class DoubleArgumentMarshaler implements ArgumentMarshaler {
  private double doubleValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      doubleValue = Double.parseDouble(parameter);
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_DOUBLE;
      throw new ArgsException();
    } catch (NumberFormatException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_DOUBLE;
      throw new ArgsException();
    }
  }

  public Object get() {
    return doubleValue;
  }
}
```

Для нового типа добавляются новые коды ошибок:

```
private enum ErrorCode {
  OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
  MISSING_DOUBLE, INVALID_DOUBLE}
```

А еще понадобится функция getDouble:

```
public double getDouble(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  try {
```

```
    return am == null ? 0 : (Double) am.get();
  } catch (Exception e) {
    return 0.0;
  }
}
```

И все тесты успешно проходят! Добавление нового типа прошло в целом безболезненно. Теперь давайте убедимся в том, что обработка ошибок работает правильно. Следующий тестовый сценарий проверяет, что при передаче неразбираемой строки с аргументом ## выдается соответствующая ошибка:

```
  public void testInvalidDouble() throws Exception {
    Args args = new Args("x##", new String[] {"-x","Forty two"});
    assertFalse(args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0, args.getInt('x'));
    assertEquals("Argument -x expects a double but was 'Forty two'.",
                 args.errorMessage());
  }
---
  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Should not get here.");
      case UNEXPECTED_ARGUMENT:
        return unexpectedArgumentMessage();
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
                             errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
                             errorArgumentId);
      case INVALID_DOUBLE:
        return String.format("Argument -%c expects a double but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_DOUBLE:
        return String.format("Could not find double parameter for -%c.",
                             errorArgumentId);
    }
    return "";
  }
```

Тесты успешно проходят. Следующий тест проверяет, что ошибка с отсутствующим аргументом double будет успешно обнаружена:

```
public void testMissingDouble() throws Exception {
  Args args = new Args("x##", new String[]{"-x"});
  assertFalse(args.isValid());
  assertEquals(0, args.cardinality());
```

```
  assertFalse(args.has('x'));
  assertEquals(0.0, args.getDouble('x'), 0.01);
  assertEquals("Could not find double parameter for -x.",
               args.errorMessage());
}
```

Как и ожидалось, все проходит успешно. Этот тест был написан просто для полноты картины.

Код исключения некрасив, и в классе `Args` ему не место. Также в коде инициируется исключение `ParseException`, которое на самом деле нам не принадлежит. Давайте объединим все исключения в один класс `ArgsException` и переместим его в отдельный модуль.

```
public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
    MISSING_DOUBLE, INVALID_DOUBLE}
}
---

public class Args {
  ...
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ArgsException.ErrorCode errorCode = ArgsException.ErrorCode.OK;
  private List<String> argsList;

  public Args(String schema, String[] args) throws ArgsException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    valid = parse();
  }

  private boolean parse() throws ArgsException {
    if (schema.length() == 0 && argsList.size() == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }
```

```
private boolean parseSchema() throws ArgsException {
  ...
}

private void parseSchemaElement(String element) throws ArgsException {
  ...
  else
    throw new ArgsException(
      String.format("Argument: %c has invalid format: %s.",
                    elementId,elementTail));
}

private void validateSchemaElementId(char elementId) throws ArgsException {
  if (!Character.isLetter(elementId)) {
    throw new ArgsException(
      "Bad character:" + elementId + "in Args format: " + schema);
  }
}

...
private void parseElement(char argChar) throws ArgsException {
  if (setArgument(argChar))
    argsFound.add(argChar);
  else {
    unexpectedArguments.add(argChar);
    errorCode = ArgsException.ErrorCode.UNEXPECTED_ARGUMENT;
    valid = false;
  }
}
...

private class StringArgumentMarshaler implements ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
    try {
      stringValue = currentArgument.next();
    } catch (NoSuchElementException e) {
      errorCode = ArgsException.ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }

  public Object get() {
    return stringValue;
  }
}

private class IntegerArgumentMarshaler implements ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
```

```
      try {
        parameter = currentArgument.next();
        intValue = Integer.parseInt(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ArgsException.ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ArgsException.ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }

  private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        doubleValue = Double.parseDouble(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ArgsException.ErrorCode.MISSING_DOUBLE;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ArgsException.ErrorCode.INVALID_DOUBLE;
        throw new ArgsException();
      }
    }

    public Object get() {
      return doubleValue;
    }
  }
}
```

Хорошо — теперь Args выдает единственное исключение ArgsException. Выделение ArgsException в отдельный модуль приведет к тому, что большой объем вспомогательного кода обработки ошибок переместится из модуля Args в этот модуль. Это наиболее естественное и очевидное место для размещения этого кода, вдобавок перемещение поможет очистить перерабатываемый модуль Args.

Итак, нам удалось полностью отделить код исключений и ошибок от модуля Args (листинги 14.13–14.16). Для решения этой задачи понадобилось примерно 30 промежуточных шагов, и после каждого шага проверялось прохождение всех тестов.

Листинг 14.13. ArgsTest.java

```
package com.objectmentor.utilities.args;

import junit.framework.TestCase;

public class ArgsTest extends TestCase {
  public void testCreateWithNoSchemaOrArguments() throws Exception {
    Args args = new Args("", new String[0]);
    assertEquals(0, args.cardinality());
  }

  public void testWithNoSchemaButWithOneArgument() throws Exception {
    try {
      new Args("", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                   e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }

  public void testWithNoSchemaButWithMultipleArguments() throws Exception {
    try {
      new Args("", new String[]{"-x", "-y"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                   e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }

  }

  public void testNonLetterSchema() throws Exception {
    try {
      new Args("*", new String[]{});
      fail("Args constructor should have thrown exception");
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                   e.getErrorCode());
      assertEquals('*', e.getErrorArgumentId());
    }
  }

  public void testInvalidArgumentFormat() throws Exception {
    try {
      new Args("f~", new String[]{});
      fail("Args constructor should have throws exception");
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_FORMAT, e.getErrorCode());
```

```
    assertEquals('f', e.getErrorArgumentId());
  }
}

public void testSimpleBooleanPresent() throws Exception {
  Args args = new Args("x", new String[]{"-x"});
  assertEquals(1, args.cardinality());
  assertEquals(true, args.getBoolean('x'));
}

public void testSimpleStringPresent() throws Exception {
  Args args = new Args("x*", new String[]{"-x", "param"});
  assertEquals(1, args.cardinality());
  assertTrue(args.has('x'));
  assertEquals("param", args.getString('x'));
}

public void testMissingStringArgument() throws Exception {
  try {
    new Args("x*", new String[]{"-x"});
    fail();
  } catch (ArgsException e) {
    assertEquals(ArgsException.ErrorCode.MISSING_STRING, e.getErrorCode());
    assertEquals('x', e.getErrorArgumentId());
  }
}

public void testSpacesInFormat() throws Exception {
  Args args = new Args("x, y", new String[]{"-xy"});
  assertEquals(2, args.cardinality());
  assertTrue(args.has('x'));
  assertTrue(args.has('y'));
}

public void testSimpleIntPresent() throws Exception {
  Args args = new Args("x#", new String[]{"-x", "42"});
  assertEquals(1, args.cardinality());
  assertTrue(args.has('x'));
  assertEquals(42, args.getInt('x'));
}
public void testInvalidInteger() throws Exception {
  try {
    new Args("x#", new String[]{"-x", "Forty two"});
    fail();
  } catch (ArgsException e) {
    assertEquals(ArgsException.ErrorCode.INVALID_INTEGER, e.getErrorCode());
    assertEquals('x', e.getErrorArgumentId());
    assertEquals("Forty two", e.getErrorParameter());
  }
}
```

продолжение ⇩

Листинг 14.13 (продолжение)

```
  public void testMissingInteger() throws Exception {
    try {
      new Args("x#", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.MISSING_INTEGER, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }

  public void testSimpleDoublePresent() throws Exception {
    Args args = new Args("x##", new String[]{"-x", "42.3"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42.3, args.getDouble('x'), .001);
  }

  public void testInvalidDouble() throws Exception {
    try {
      new Args("x##", new String[]{"-x", "Forty two"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_DOUBLE, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
      assertEquals("Forty two", e.getErrorParameter());
    }
  }

  public void testMissingDouble() throws Exception {
    try {
      new Args("x##", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.MISSING_DOUBLE, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }
}
```

Листинг 14.14. ArgsExceptionTest.java

```
public class ArgsExceptionTest extends TestCase {
  public void testUnexpectedMessage() throws Exception {
    ArgsException e =
      new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                        'x', null);
    assertEquals("Argument -x unexpected.", e.errorMessage());
  }

  public void testMissingStringMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_STRING,
                                        'x', null);
```

```
    assertEquals("Could not find string parameter for -x.", e.errorMessage());
  }

  public void testInvalidIntegerMessage() throws Exception {
    ArgsException e =
      new ArgsException(ArgsException.ErrorCode.INVALID_INTEGER,
                        'x', "Forty two");
    assertEquals("Argument -x expects an integer but was 'Forty two'.",
                 e.errorMessage());
  }

  public void testMissingIntegerMessage() throws Exception {
    ArgsException e =
      new ArgsException(ArgsException.ErrorCode.MISSING_INTEGER, 'x', null);
    assertEquals("Could not find integer parameter for -x.", e.errorMessage());
  }

  public void testInvalidDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.INVALID_DOUBLE,
                                        'x', "Forty two");
    assertEquals("Argument -x expects a double but was 'Forty two'.",
                 e.errorMessage());
  }

  public void testMissingDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_DOUBLE,
                                        'x', null);
    assertEquals("Could not find double parameter for -x.", e.errorMessage());
  }
}
```

Листинг 14.15. ArgsException.java

```
public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public ArgsException(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public ArgsException(ErrorCode errorCode, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
  }

  public ArgsException(ErrorCode errorCode, char errorArgumentId,
                       String errorParameter) {
```

продолжение ⇩

Листинг 14.15 (продолжение)

```
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
  }

  public char getErrorArgumentId() {
    return errorArgumentId;
  }

  public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
  }

  public String getErrorParameter() {
    return errorParameter;
  }

  public void setErrorParameter(String errorParameter) {
    this.errorParameter = errorParameter;
  }

  public ErrorCode getErrorCode() {
    return errorCode;
  }

  public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Should not get here.");
      case UNEXPECTED_ARGUMENT:
        return String.format("Argument -%c unexpected.", errorArgumentId);
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
                             errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
                             errorArgumentId);
      case INVALID_DOUBLE:
        return String.format("Argument -%c expects a double but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_DOUBLE:
        return String.format("Could not find double parameter for -%c.",
                             errorArgumentId);
```

```
    }
    return "";
  }

  public enum ErrorCode {
    OK, INVALID_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE}
}
```

Листинг 14.16. Args.java

```
public class Args {
  private String schema;
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private Iterator<String> currentArgument;
  private List<String> argsList;

  public Args(String schema, String[] args) throws ArgsException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    parse();
  }

  private void parse() throws ArgsException {
    parseSchema();
    parseArguments();
  }

  private boolean parseSchema() throws ArgsException {
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        parseSchemaElement(element.trim());
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ArgsException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
```

продолжение ⇩

Листинг 14.16 (продолжение)

```
      marshalers.put(elementId, new DoubleArgumentMarshaler());
    else
      throw new ArgsException(ArgsException.ErrorCode.INVALID_FORMAT,
                              elementId, elementTail);
  }

  private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId)) {
      throw new ArgsException(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                              elementId, null);
    }
  }

  private void parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
      String arg = currentArgument.next();
      parseArgument(arg);
    }
  }

  private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      throw new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                              argChar, null);
    }
  }

  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
      m.set(currentArgument);
      return true;
    } catch (ArgsException e) {
      e.setErrorArgumentId(argChar);
      throw e;
    }
  }
```

```
  public int cardinality() {
    return argsFound.size();
  }

  public String usage() {
    if (schema.length() > 0)
      return "-[" + schema + "]";
    else
      return "";
  }

  public boolean getBoolean(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
      b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
      b = false;
    }
    return b;
  }

  public String getString(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
      return "";
    }
  }

  public int getInt(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
      return 0;
    }
  }

  public double getDouble(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Double) am.get();
    } catch (Exception e) {
      return 0.0;
    }
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }
}
```

Основные изменения в классе `Args` свелись к удалениям. Большая часть кода ушла из `Args` в `ArgsException`. Хорошо. Мы также переместили все разновидности `ArgumentMarshaller` в отдельные файлы. Еще лучше!

Одним из важнейших аспектов хорошей программной архитектуры является логическое разбиение кода — создание подходящих мест для размещения разных кодовых блоков. Разделение ответственности заметно упрощает понимание и сопровождение кода.

Обратите внимание на метод `errorMessage` класса `ArgsException`. Очевидно, размещение форматирования сообщения об ошибках нарушает принцип единой ответственности. Класс `Args` должен заниматься обработкой аргументов, а не форматом сообщений об ошибках. Но насколько логично размещать код форматирования сообщений в `ArgsException`?

Откровенно говоря, это компромиссное решение. Пользователям, которым не нравится, что сообщения об ошибках поставляет класс `ArgsException`, придется написать собственную реализацию.

К этому моменту мы уже вплотную подошли к окончательному решению, приведенному в начале этой главы. Завершающие преобразования остаются читателю для самостоятельных упражнений.

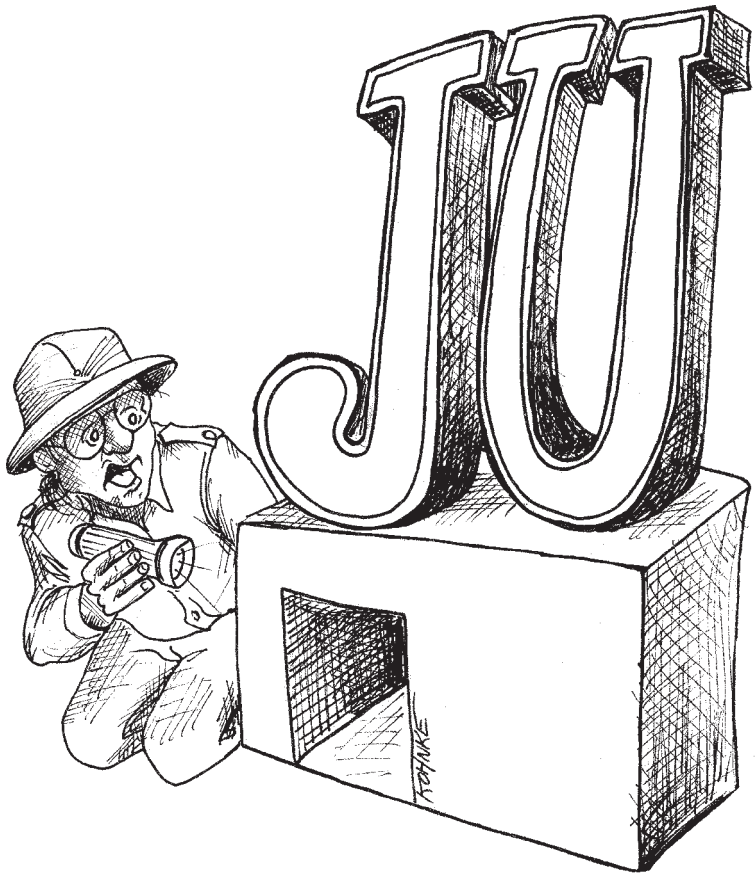
Заключение

Заставить код работать недостаточно. Работоспособный код часто несовершенен. Программисты, которые заставляют свой код работать и на этом считают свою задачу выполненной, ведут себя непрофессионально. Возможно, они опасаются, что у них не хватит времени для совершенствования структуры и архитектуры кода, но я не могу с этим согласиться. Ничто не оказывает настолько всестороннего и длительного отрицательного влияния на судьбу программного проекта, как плохой код. Плохой график можно переделать, плохие требования можно переопределить. Плохую динамику рабочей группы еще можно исправить. Плохой код загнивает и разбухает, превращаясь в беспощадный груз, который тянет группу ко дну. Сколько раз я видел, как работа заходит в тупик по одной причине: в спешке вместо добротного кода создавалась какая-то безобразная мешанина, которая после этого обрекала группу на бесконечные мучения.

Конечно, плохой код можно вычистить. Но это обходится очень дорого. В процессе загнивания кода модули постепенно проникают друг в друга, образуется множество скрытых и запутанных зависимостей. Поиск и разрыв старых зависимостей — длительная, тяжелая работа. С другой стороны, поддерживать чистоту в коде относительно несложно. Если утром вы устроили беспорядок в модуле, то его будет легко вычистить днем. Или еще лучше, если вы устроили беспорядок пять минут назад, то его будет очень легко вычистить прямо сейчас.

Итак, постоянно следите за тем, чтобы ваш код оставался как можно более простым и чистым. Не допускайте, чтобы он начал загнивать.

15 Внутреннее строение JUnit



JUnit — одна из самых известных инфраструктур для языка Java. Как и положено нормальной инфраструктуре, она концептуально проста, точна в определениях и элегантна в реализации. Но как выглядит ее код? В этой главе мы покритикуем пример, взятый из инфраструктуры JUnit.

Инфраструктура JUnit

У JUnit много авторов, но все началось с совместного перелета Кента Бека и Эрика Гамма в Атланту. Кент хотел освоить Java, а Эрик собирался заняться изучением тестовой инфраструктуры Кента для языка Smalltalk. «А что может быть более естественным для двух „технарей“, запертых в тесном пространстве, чем достать портативные компьютеры и взяться за программирование?[1]» За три часа «высотной работы» были написаны основы JUnit.

Модуль, который мы рассмотрим в этой главе, предназначен для выявления ошибок сравнения строк. Он называется `ComparisonCompactor`. Получив две различающиеся строки (например, `ABCDE` и `ABXDE`), он выдает сводку различий между ними, генерируя строку вида `<...B[X]D...>`.

Я мог бы объяснить и подробнее, но тестовые сценарии сделают это лучше. Просмотрите листинг 15.1 и вы отлично поймете требования этого модуля. А заодно критически проанализируйте структуру тестов. Нельзя ли упростить их, сделать более наглядными?

Листинг 15.1. ComparisonCompactorTest.java

```
package junit.tests.framework;

import junit.framework.ComparisonCompactor;
import junit.framework.TestCase;

public class ComparisonCompactorTest extends TestCase {
  public void testMessage() {
    String failure= new ComparisonCompactor(0, "b", "c").compact("a");
    assertTrue("a expected:<[b]> but was:<[c]>".equals(failure));
  }

  public void testStartSame() {
    String failure= new ComparisonCompactor(1, "ba", "bc").compact(null);
    assertEquals(«expected:<b[a]> but was:<b[c]>», failure);
  }

  public void testEndSame() {
    String  failure= new ComparisonCompactor(1, "ab", "cb").compact(null);
    assertEquals("expected:<[a]b> but was:<[c]b>", failure);
  }

  public void testSame() {
    String failure= new ComparisonCompactor(1, "ab", "ab").compact(null);
    assertEquals("expected:<ab> but was:<ab>", failure);
  }

  public void testNoContextStartAndEndSame() {
```

[1] JUnit Pocket Guide, Kent Beck, O'Reilly, 2004, с. 43.

```
    String failure= new ComparisonCompactor(0, "abc", "adc").compact(null);
    assertEquals("expected:<...[b]...> but was:<...[d]...>", failure);
  }

  public void testStartAndEndContext() {
    String failure= new ComparisonCompactor(1, "abc", "adc").compact(null);
    assertEquals("expected:<a[b]c> but was:<a[d]c>", failure);
  }

  public void testStartAndEndContextWithEllipses() {
    String failure=
       new ComparisonCompactor(1, "abcde", "abfde").compact(null);
    assertEquals("expected:<...b[c]d...> but was:<...b[f]d...>", failure);
  }

  public void testComparisonErrorStartSameComplete() {
    String failure= new ComparisonCompactor(2, "ab", "abc").compact(null);
    assertEquals("expected:<ab[]> but was:<ab[c]>", failure);
  }

  public void testComparisonErrorEndSameComplete() {
    String failure= new ComparisonCompactor(0, "bc", "abc").compact(null);
    assertEquals("expected:<[]...> but was:<[a]...>", failure);
  }

  public void testComparisonErrorEndSameCompleteContext() {
    String failure= new ComparisonCompactor(2, "bc", "abc").compact(null);
    assertEquals("expected:<[]bc> but was:<[a]bc>", failure);
  }

  public void testComparisonErrorOverlapingMatches() {
    String failure= new ComparisonCompactor(0, "abc", "abbc").compact(null);
    assertEquals("expected:<...[]...> but was:<...[b]...>", failure);
  }

  public void testComparisonErrorOverlapingMatchesContext() {
    String failure= new ComparisonCompactor(2, "abc", "abbc").compact(null);
    assertEquals("expected:<ab[]c> but was:<ab[b]c>", failure);
  }

  public void testComparisonErrorOverlapingMatches2() {
    String failure= new ComparisonCompactor(0, "abcdde",
    "abcde").compact(null);
    assertEquals("expected:<...[d]...> but was:<...[]...>", failure);
  }

  public void testComparisonErrorOverlapingMatches2Context() {
    String failure=
       new ComparisonCompactor(2, "abcdde", "abcde").compact(null);
    assertEquals("expected:<...cd[d]e> but was:<...cd[]e>", failure);
  }
```

продолжение ⇩

Листинг 15.1 (продолжение)

```
  public void testComparisonErrorWithActualNull() {
    String failure= new ComparisonCompactor(0, "a", null).compact(null);
    assertEquals("expected:<a> but was:<null>", failure);
  }

  public void testComparisonErrorWithActualNullContext() {
    String failure= new ComparisonCompactor(2, "a", null).compact(null);
    assertEquals("expected:<a> but was:<null>", failure);
  }

  public void testComparisonErrorWithExpectedNull() {
    String failure= new ComparisonCompactor(0, null, "a").compact(null);
    assertEquals("expected:<null> but was:<a>", failure);
  }

  public void testComparisonErrorWithExpectedNullContext() {
    String failure= new ComparisonCompactor(2, null, "a").compact(null);
    assertEquals("expected:<null> but was:<a>", failure);
  }

  public void testBug609972() {
    String failure= new ComparisonCompactor(10, "S&P500", "0").compact(null);
    assertEquals("expected:<[S&P50]0> but was:<[]0>", failure);
  }
}
```

Я провел для `ComparisonCompactor` анализ покрытия кода на основе этих тестов. В ходе тестирования обеспечивалось 100%-ное покрытие: была выполнена каждая строка кода, каждая команда `if` и цикл `for`. Я удостоверился в том, что код работает правильно, а также преисполнился уважения к мастерству его авторов.

Код `ComparisonCompactor` приведен в листинге 15.2. Не жалейте времени и как следует разберитесь в нем. Вероятно, вы согласитесь с тем, что код достаточно выразителен, обладает логичным разбиением и простой структурой. А когда вы закончите, мы вместе начнем придираться к мелочам.

Листинг 15.2. ComparisonCompactor.java (исходный код)

```
package junit.framework;

public class ComparisonCompactor {

  private static final String ELLIPSIS = "...";
  private static final String DELTA_END = "]";
  private static final String DELTA_START = "[";

  private int fContextLength;
  private String fExpected;
  private String fActual;
  private int fPrefix;
  private int fSuffix;
```

```
public ComparisonCompactor(int contextLength,
                           String expected,
                             String actual) {
  fContextLength = contextLength;
  fExpected = expected;
  fActual = actual;
}

public String compact(String message) {
  if (fExpected == null || fActual == null || areStringsEqual())
    return Assert.format(message, fExpected, fActual);
  findCommonPrefix();
  findCommonSuffix();
  String expected = compactString(fExpected);
  String actual = compactString(fActual);
  return Assert.format(message, expected, actual);
}

private String compactString(String source) {
  String result = DELTA_START +
                  source.substring(fPrefix, source.length() -
                    fSuffix + 1) + DELTA_END;
  if (fPrefix > 0)
    result = computeCommonPrefix() + result;
  if (fSuffix > 0)
    result = result + computeCommonSuffix();
  return result;
}

private void findCommonPrefix() {
  fPrefix = 0;
  int end = Math.min(fExpected.length(), fActual.length());
  for (; fPrefix < end; fPrefix++) {
    if (fExpected.charAt(fPrefix) != fActual.charAt(fPrefix))
      break;
  }
}

private void findCommonSuffix() {
  int expectedSuffix = fExpected.length() - 1;
  int actualSuffix = fActual.length() - 1;
  for (;
       actualSuffix >= fPrefix && expectedSuffix >= fPrefix;
        actualSuffix--, expectedSuffix--) {
    if (fExpected.charAt(expectedSuffix) != fActual.charAt(actualSuffix))
      break;
  }
  fSuffix = fExpected.length() - expectedSuffix;
}
private String computeCommonPrefix() {
```

продолжение ⇩

Листинг 15.2 (продолжение)

```
    return (fPrefix > fContextLength ? ELLIPSIS : "") +
             fExpected.substring(Math.max(0, fPrefix - fContextLength),
                                 fPrefix);
  }

  private String computeCommonSuffix() {
    int end = Math.min(fExpected.length() - fSuffix + 1 + fContextLength,
                       fExpected.length());
    return fExpected.substring(fExpected.length() - fSuffix + 1, end) +
           (fExpected.length() - fSuffix + 1 < fExpected.length() -
            fContextLength ? ELLIPSIS : "");
  }

  private boolean areStringsEqual() {
    return fExpected.equals(fActual);
  }
}
```

Вероятно, вы найдете в этом модуле некоторые недочеты. В нем встречаются длинные выражения, какие-то малопонятные +1 и т. д. Но в целом модуль весьма хорош. В конце концов, он мог бы выглядеть и так, как показано в листинге 15.3.

Листинг 15.3. ComparisonCompator.java (переработанная версия)

```
package junit.framework;

public class ComparisonCompactor {
  private int ctxt;
  private String s1;
  private String s2;
  private int pfx;
  private int sfx;

  public ComparisonCompactor(int ctxt, String s1, String s2) {
    this.ctxt = ctxt;
    this.s1 = s1;
    this.s2 = s2;
  }

  public String compact(String msg) {
    if (s1 == null || s2 == null || s1.equals(s2))
      return Assert.format(msg, s1, s2);

    pfx = 0;
    for (; pfx < Math.min(s1.length(), s2.length()); pfx++) {
      if (s1.charAt(pfx) != s2.charAt(pfx))
        break;
    }
    int sfx1 = s1.length() - 1;
    int sfx2 = s2.length() - 1;
```

```
    for (; sfx2 >= pfx && sfx1 >= pfx; sfx2--, sfx1--) {
      if (s1.charAt(sfx1) != s2.charAt(sfx2))
        break;
    }
    sfx = s1.length() - sfx1;
    String cmp1 = compactString(s1);
    String cmp2 = compactString(s2);
    return Assert.format(msg, cmp1, cmp2);
  }

  private String compactString(String s) {
    String result =
      "[" + s.substring(pfx, s.length() - sfx + 1) + "]";
    if (pfx > 0)
      result = (pfx > ctxt ? "..." : "") +
        s1.substring(Math.max(0, pfx - ctxt), pfx) + result;
    if (sfx > 0) {
      int end = Math.min(s1.length() - sfx + 1 + ctxt, s1.length());
      result = result + (s1.substring(s1.length() - sfx + 1, end) +
        (s1.length() - sfx + 1 < s1.length() - ctxt ? "..." : ""));
    }
    return result;
  }
}
```

Авторы оставили эту модуль в очень хорошей форме. И все же «правило бойскаута[1]» гласит: все нужно оставлять чище, чем было до вашего прихода. Итак, как же улучшить исходный код в листинге 15.2?

Первое, что мне решительно не понравилось, — префикс `f` у имен переменных классов [N6]. В современных средах разработки подобное кодирование области видимости излишне. Давайте уберем все префиксы:

```
private int contextLength;
private String expected;
private String actual;
private int prefix;
private int suffix;
```

Также бросается в глаза неинкапсулированная условная команда в начале функции `compact` [G28].

```
public String compact(String message) {
  if (expected == null || actual == null || areStringsEqual())
    return Assert.format(message, expected, actual);
  findCommonPrefix();
  findCommonSuffix();
  String expected = compactString(this.expected);
  String actual = compactString(this.actual);
  return Assert.format(message, expected, actual);
}
```

[1] См. раздел «Правило бойскаута» на с. 37.

Инкапсуляция поможет лучше выразить намерения разработчика. Поэтому я создал метод с именем, поясняющим его смысл:

```
public String compact(String message) {
  if (shouldNotCompact())
    return Assert.format(message, expected, actual);
  findCommonPrefix();
  findCommonSuffix();
  String expected = compactString(this.expected);
  String actual = compactString(this.actual);
  return Assert.format(message, expected, actual);
}

private boolean shouldNotCompact() {
  return expected == null || actual == null || areStringsEqual();
}
```

Запись `this.expected` и `this.actual` в функции `compact` тоже оставляет желать лучшего. Это произошло, когда мы переименовали `fExpected` в `expected`. Зачем в функции используются переменные с именами, совпадающими с именами переменных класса? Ведь они имеют разный смысл [N4]? Неоднозначность в именах следует исключить.

```
String compactExpected = compactString(expected);
String compactActual = compactString(actual);
```

Отрицательные условия чуть сложнее для понимания, чем положительные [G29]. Чтобы проверяемое условие стало более понятным, мы инвертируем его:

```
public String compact(String message) {
  if (canBeCompacted()) {
    findCommonPrefix();
    findCommonSuffix();
    String compactExpected = compactString(expected);
    String compactActual = compactString(actual);
    return Assert.format(message, compactExpected, compactActual);
  } else {
    return Assert.format(message, expected, actual);
  }
}

private boolean canBeCompacted() {
  return expected != null && actual != null && !areStringsEqual();
}
```

Имя функции `compact` выглядит немного странно [N7]. Хотя она выполняет сжатие строк, этого не произойдет, если `canBeCompacted` вернет `false`. Таким образом, выбор имени `compact` скрывает побочный эффект проверки. Также обратите внимание на то, что функция возвращает отформатированное сообщение, а не просто сжатые строки. Следовательно, функцию было бы правильнее назвать `formatCompactedComparison`. В этом случае она гораздо лучше читается вместе с аргументом:

```
public String formatCompactedComparison(String message) {
```

Тело команды if — то место, где выполняется фактическое сжатие строк expected и actual. Мы извлечем этот код в метод compactExpectedAndActual. Тем не менее все форматирование должно происходить в функции formatCompactedComparison. Функция compact... не должна делать ничего, кроме сжатия [G30]. Разобьем ее следующим образом:

```
...
  private String compactExpected;
  private String compactActual;
...
  public String formatCompactedComparison(String message) {
    if (canBeCompacted()) {
      compactExpectedAndActual();
      return Assert.format(message, compactExpected, compactActual);
    } else {
      return Assert.format(message, expected, actual);
    }
  }

  private void compactExpectedAndActual() {
    findCommonPrefix();
    findCommonSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
  }
```

Обратите внимание: это преобразование заставило нас повысить compactExpected и compactActual до переменных класса. Еще мне не нравится то, что в двух последних строках новой функции возвращаются переменные, а в первых двух — нет. Это противоречит рекомендациям по использованию единых конвенций [G11]. Значит, функции findCommonPrefix и findCommonSuffix следует изменить так, чтобы они возвращали значения префикса и суффикса.

```
private void compactExpectedAndActual() {
  prefixIndex = findCommonPrefix();
  suffixIndex = findCommonSuffix();
  compactExpected = compactString(expected);
  compactActual = compactString(actual);
}

private int findCommonPrefix() {
  int prefixIndex = 0;
  int end = Math.min(expected.length(), actual.length());
  for (; prefixIndex < end; prefixIndex++) {
    if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
      break;
  }
  return prefixIndex;
}
```

```
private int findCommonSuffix() {
  int expectedSuffix = expected.length() - 1;
  int actualSuffix = actual.length() - 1;
  for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
    actualSuffix--, expectedSuffix--) {
    if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
      break;
  }
  return expected.length() - expectedSuffix;
}
```

Также следует изменить имена переменных класса так, чтобы они стали чуть более точными [N1]; в конце концов, обе переменные представляют собой индексы.

Тщательное изучение `findCommonSuffix` выявляет скрытую временную привязку [G31]; работа функции зависит от того, что значение `prefixIndex` вычисляется функцией `findCommonPrefix`. Если вызвать эти две функции в неверном порядке, вам предстоит непростой сеанс отладки. Чтобы эта временная привязка стала очевидной, значение `prefixIndex` будет передаваться при вызове `findCommonSuffix` в аргументе.

```
private void compactExpectedAndActual() {
  prefixIndex = findCommonPrefix();
  suffixIndex = findCommonSuffix(prefixIndex);
  compactExpected = compactString(expected);
  compactActual = compactString(actual);
}

private int findCommonSuffix(int prefixIndex) {
  int expectedSuffix = expected.length() - 1;
  int actualSuffix = actual.length() - 1;
  for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
       actualSuffix--, expectedSuffix--) {
    if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
      break;
  }
  return expected.length() - expectedSuffix;
}
```

Но и такое решение оставляет желать лучшего. Передача аргумента `prefixIndex` выглядит нелогично [G32]. Она устанавливает порядок вызова, но никоим образом не объясняет необходимость именно такого порядка. Другой программист может отменить внесенное изменение, так как ничто не указывает на то, что этот параметр действительно необходим.

```
private void compactExpectedAndActual() {
  findCommonPrefixAndSuffix();
  compactExpected = compactString(expected);
  compactActual = compactString(actual);
}

private void findCommonPrefixAndSuffix() {
```

```
    findCommonPrefix();
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (;
         actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
         actualSuffix--, expectedSuffix--
      ) {
      if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
        break;
    }
    suffixIndex = expected.length() - expectedSuffix;
  }

  private void findCommonPrefix() {
    prefixIndex = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixIndex < end; prefixIndex++)
      if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
        break;
  }
```

Функции `findCommonPrefix` и `findCommonSuffix` возвращаются к прежнему виду, функция `findCommonSuffix` переименовывается в `findCommonPrefixAndSuffix`, и в нее включается вызов `findCommonPrefix` до выполнения каких-либо других действий. Тем самым временная связь двух функций устанавливается гораздо более радикально, чем в предыдущем решении. Кроме того, новое решение со всей очевидностью демонстрирует, насколько уродлива функция `findCommonPrefixAndSuffix`. Давайте немного почистим ее.

```
private void findCommonPrefixAndSuffix() {
  findCommonPrefix();
  int suffixLength = 1;
  for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
    if (charFromEnd(expected, suffixLength) !=
        charFromEnd(actual, suffixLength))
      break;
  }
  suffixIndex = suffixLength;
}

private char charFromEnd(String s, int i) {
    return s.charAt(s.length()-i);}

private boolean suffixOverlapsPrefix(int suffixLength) {
  return actual.length() - suffixLength < prefixLength ||
    expected.length() - suffixLength < prefixLength;
}
```

Так гораздо лучше. Новая версия кода очевидно показывает, что `suffixIndex` в действительности определяет *длину* суффикса, а прежнее имя было выбрано неудачно. Это относится и к `prefixIndex`, хотя в данном случае «индекс» и «длина» являются синонимами. Несмотря на это, использование термина «длина»

выглядит более последовательно. Проблема в том, что значение переменной `suffixIndex` отсчитывается не от 0, а от 1, так что называть его «длиной» не совсем корректно (кстати, этим же обстоятельством объясняются загадочные прибавления +1 в `computeCommonSuffix` [G33]). Давайте исправим этот недостаток. Результат показан в листинге 15.4.

Листинг 15.4. ComparisonCompactor.java (промежуточная версия)

```
public class ComparisonCompactor {
...
  private int suffixLength;
...
  private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    suffixLength = 0;
    for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
      if (charFromEnd(expected, suffixLength) !=
          charFromEnd(actual, suffixLength))
        break;
    }
  }

  private char charFromEnd(String s, int i) {
    return s.charAt(s.length() - i - 1);
  }

  private boolean suffixOverlapsPrefix(int suffixLength) {
    return actual.length() - suffixLength <= prefixLength ||
      expected.length() - suffixLength <= prefixLength;
  }

...
  private String compactString(String source) {
    String result =
      DELTA_START +
      source.substring(prefixLength, source.length() - suffixLength) +
      DELTA_END;
    if (prefixLength > 0)
      result = computeCommonPrefix() + result;
    if (suffixLength > 0)
      result = result + computeCommonSuffix();
    return result;
  }
...
  private String computeCommonSuffix() {
    int end = Math.min(expected.length() - suffixLength +
      contextLength, expected.length()
    );
    return
      expected.substring(expected.length() - suffixLength, end) +
      (expected.length() - suffixLength <
```

```
        expected.length() - contextLength ?
        ELLIPSIS : "");
  }
```

Все +1 в computeCommonSuffix были заменены на -1 в charFromEnd, где это смотрится абсолютно логично; также были изменены два оператора <= в suffixOverlapsPrefix, где это тоже абсолютно логично. Это позволило переименовать suffixIndex в suffixLength, с заметным улучшением удобочитаемости кода.

Однако здесь возникла одна проблема. В ходе устранения +1 я заметил в compactString следующую строку:

```
if (suffixLength > 0)
```

Найдите ее в листинге 15.4. Так как suffixLength стало на 1 меньше, чем было прежде, мне следовало бы заменить оператор > оператором >=, но это выглядит нелогично. При более внимательном анализе мы видим, что команда if предотвращает присоединение суффикса с нулевой длиной. Но до внесения изменений команда if была бесполезной, потому что значение suffixIndex не могло быть меньше 1!

Это ставит под сомнение полезность обеих команд if в compactString! Похоже, обе команды можно исключить. Закомментируем их и проведем тестирование. Тесты прошли! Давайте изменим структуру compactString, чтобы удалить лишние команды if и значительно упростить самую функцию [G9].

```
private String compactString(String source) {
  return
    computeCommonPrefix() +
    DELTA_START +
    source.substring(prefixLength, source.length() - suffixLength) +
    DELTA_END +
    computeCommonSuffix();
}
```

Стало гораздо лучше! Теперь мы видим, что функция compactString просто соединяет фрагменты строки. Вероятно, этот факт можно сделать еще более очевидным. Осталось еще много мелких улучшений, которые можно было бы внести в код. Но я не стану мучить вас подробными описаниями остальных изменений и просто приведу окончательный результат в листинге 15.5.

Листинг 15.5. ComparisonCompactor.java (окончательная версия)

```
package junit.framework;

public class ComparisonCompactor {

  private static final String ELLIPSIS = "...";
  private static final String DELTA_END = "]";
  private static final String DELTA_START = "[";

  private int contextLength;
  private String expected;
  private String actual;
```

продолжение ⇩

Листинг 15.5 (продолжение)

```
  private int prefixLength;
  private int suffixLength;

  public ComparisonCompactor(
    int contextLength, String expected, String actual
  ) {
    this.contextLength = contextLength;
    this.expected = expected;
    this.actual = actual;
  }

  public String formatCompactedComparison(String message) {
    String compactExpected = expected;
    String compactActual = actual;
    if (shouldBeCompacted()) {
      findCommonPrefixAndSuffix();
      compactExpected = compact(expected);
      compactActual = compact(actual);
    }
    return Assert.format(message, compactExpected, compactActual);
  }

  private boolean shouldBeCompacted() {
    return !shouldNotBeCompacted();
  }

  private boolean shouldNotBeCompacted() {
    return expected == null ||
           actual == null ||
           expected.equals(actual);
  }

  private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    suffixLength = 0;
    for (; !suffixOverlapsPrefix(); suffixLength++) {
      if (charFromEnd(expected, suffixLength) !=
          charFromEnd(actual, suffixLength)
      )
        break;
    }
  }

  private char charFromEnd(String s, int i) {
    return s.charAt(s.length() - i - 1);
  }

  private boolean suffixOverlapsPrefix() {
    return actual.length() - suffixLength <= prefixLength ||
      expected.length() - suffixLength <= prefixLength;
  }
```

```
  private void findCommonPrefix() {
    prefixLength = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixLength < end; prefixLength++)
      if (expected.charAt(prefixLength) != actual.charAt(prefixLength))
        break;
  }

  private String compact(String s) {
    return new StringBuilder()
      .append(startingEllipsis())
      .append(startingContext())
      .append(DELTA_START)
      .append(delta(s))
      .append(DELTA_END)
      .append(endingContext())
      .append(endingEllipsis())
      .toString();
  }

  private String startingEllipsis() {
    return prefixLength > contextLength ? ELLIPSIS : "";
  }

  private String startingContext() {
    int contextStart = Math.max(0, prefixLength - contextLength);
    int contextEnd = prefixLength;
    return expected.substring(contextStart, contextEnd);
  }

  private String delta(String s) {
    int deltaStart = prefixLength;
    int deltaEnd = s.length() - suffixLength;
    return s.substring(deltaStart, deltaEnd);
  }

  private String endingContext() {
    int contextStart = expected.length() - suffixLength;
    int contextEnd =
      Math.min(contextStart + contextLength, expected.length());
    return expected.substring(contextStart, contextEnd);
  }

  private String endingEllipsis() {
    return (suffixLength > contextLength ? ELLIPSIS : "");
  }
}
```

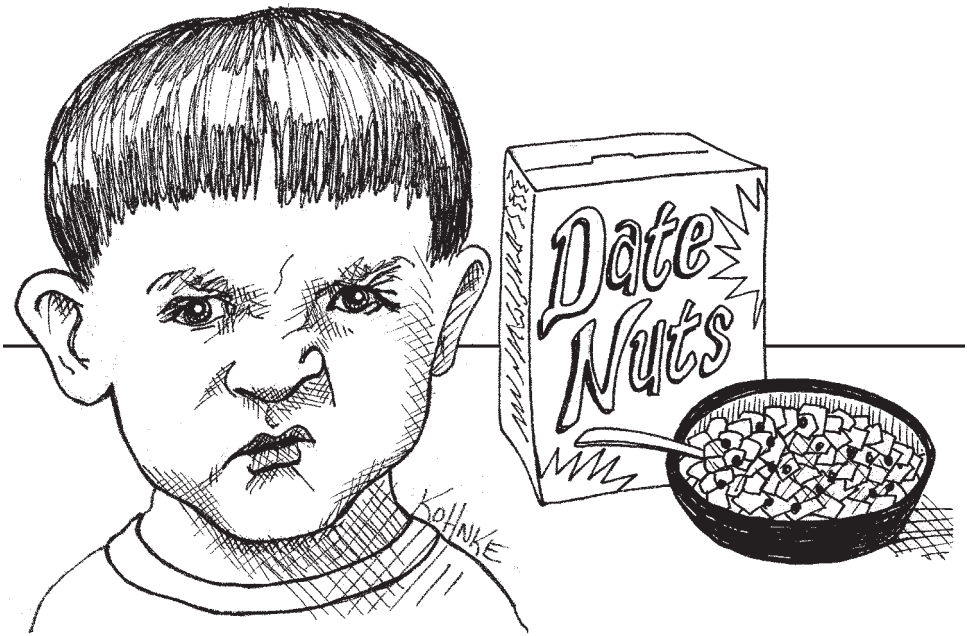
Результат выглядит вполне симпатично. Модуль делится на группы: первую группу составляют функции анализа, а вторую — функции синтеза. Функции топологически отсортированы таким образом, что определение каждой функции размещается перед ее первым использованием. Сначала определяются все функции анализа, а за ними следуют функции синтеза.

Внимательно присмотревшись, можно заметить, что я отменил некоторые решения, принятые ранее в этой главе. Например, некоторые извлеченные методы были снова встроены в `formatCompactedComparison`, а смысл выражения `shouldNotBeCompacted` снова изменился. Это типичная ситуация. Одна переработка часто приводит к другой, отменяющей первую. Переработка представляет собой итеративный процесс, полный проб и ошибок, но этот процесс неизбежно приводит к формированию кода, достойного настоящего профессионала.

Заключение

Итак, «правило бойскаута» выполнено: модуль стал чище, чем был до нашего прихода. И дело не в том, что он был недостаточно чист, — авторы отлично потрудились над ним. Однако не существует модуля, который нельзя было бы улучшить, и каждый из нас обязан оставить чужой код хотя бы немного лучше, чем он был.

16 Переработка SerialDate



Посетив страницу *http://www.jfree.org/jcommon/index.php*, вы найдете на ней описание библиотеки JCommon. Глубоко в недрах этой библиотеки скрыт пакет *org.jfree.date*. Пакет содержит класс с именем `SerialDate`. В этой главе мы займемся анализом этого класса.

Класс `SerialDate` написан Дэвидом Гилбертом (David Gilbert). Несомненно, Дэвид является опытным и компетентным программистом. Как вы сами убедитесь, в этом коде он проявил значительную степень профессионализма и дисциплины. Во всех отношениях это «хороший код». А сейчас я намерен разнести его в пух и прах.

Дело вовсе не в злом умысле. И я вовсе не считаю, что я намного лучше Дэвида и поэтому имею право критиковать его код. Действительно, если заглянуть в мой код, я уверен, что вы найдете в нем немало поводов для критики.

Нет, дело не в моем скверном характере или надменности. Я всего лишь намерен проанализировать код с профессиональной точки зрения, не более и не менее. Это то, что все мы должны делать спокойно и без угрызений совести. И все мы должны только приветствовать, когда такой анализ кто-то проводит за нас. Только после подобной критики мы узнаем нечто новое. Это делают врачи. Это делают пилоты. Это делают адвокаты. И мы, программисты, тоже должны этому научиться.

И еще одно замечание по поводу Дэвида Гилберта: Дэвид — не просто хороший программист. У него хватило смелости и доброй воли на то, чтобы бесплатно предоставить свой код сообществу. Дэвид разместил свой код в открытом доступе и предложил всем желающим использовать и обсуждать его. Отличная работа!

Класс `SerialDate` (листинг Б.1, с. 390) представляет даты в языке Java. Зачем нужен класс для представления дат, если в Java уже имеются готовые классы `java.util.Date`, `java.util.Calendar` и т. д.? Автор написал свой класс из-за проблемы, с которой часто сталкивался сам. Ее суть хорошо разъясняется в открывающем комментарии Javadoc (строка 67). Возможно, кому-то такое решение покажется радикальным, но мне и самому приходилось сталкиваться с этой проблемой, и я приветствую класс, ориентированный на работу с датой вместо времени.

Прежде всего — заставить работать

В классе `SerialDateTests` содержится набор модульных тестов (листинг Б.2, с. 411). Все тесты проходят. К сожалению, беглое изучение тестов показывает, что тестирование не покрывает часть кода [T1]. Например, поиск показывает, что метод `MonthCodeToQuarter` (строка 334) не используется [F4]. Соответственно он не включается в модульные тесты.

Итак, я запустил Clover, чтобы узнать, какая часть кода реально покрывается модульными тестами. Clover сообщает, что модульные тесты выполняют только 91 из 185 исполняемых команд `SerialDate` (около 50%) [T2]. Карта покрытия напоминала лоскутное одеяло, а по всему классы были разбросаны большие пятна невыполняемого кода.

Моей целью было полное понимание и переработка кода этого класса. Я не мог добиться этого без значительного улучшения тестового покрытия, поэтому мне пришлось написать собственный набор абсолютно независимых модульных тестов (листинг Б.4, с. 419).

Просматривая код тестов, можно заметить, что многие из них закомментированы. Эти тесты не проходили в исходном варианте. Однако они представляют поведение, которым, на мой взгляд, должен обладать класс `SerialDate`. Соответственно, в ходе переработки `SerialDate` я буду работать над тем, чтобы эти тесты тоже проходили.

Даже с несколькими закомментированными тестами Clover сообщает, что новые модульные тесты покрывают 170 (92%) из 185 исполняемых команд. Неплохо, хотя я думаю, что и этот показатель можно улучшить.

Возможно, в нескольких первых закомментированных тестах (строки 23–63) я слегка хватил через край. Их прохождение не было формально заложено при проектировании программы, но данное поведение казалось мне абсолютно очевидным [G2].

Я не знаю, зачем создавался метод `testWeekdayCodeToString`, но раз уж он был написан, казалось очевидным, что в работе метода не должен учитываться регистр символов. Написать соответствующий тест было элементарно [T3]. Заставить его работать было еще проще; я просто изменил строки 259 и 263, чтобы в них использовалась функция `equalsIgnoreCase`.

Тесты в строках 32 и 45 остались закомментированными, так как мне было неясно, нужно ли поддерживать сокращения вида «tues» и «thurs».

Тесты в строках 153 и 154 не проходят. Хотя, естественно, должны проходить [G2]. Проблема (а заодно и тесты в строках 163–213) легко исправляется внесением следующих изменений в функцию `stringToMonthCode`.

```
457         if ((result < 1) || (result > 12)) {
                result = -1;
458             for (int i = 0; i < monthNames.length; i++) {
459                 if (s.equalsIgnoreCase(shortMonthNames[i])) {
460                     result = i + 1;
461                     break;
462                 }
463                 if (s.equalsIgnoreCase(monthNames[i])) {
464                     result = i + 1;
465                     break;
466                 }
467             }
468         }
```

Закомментированный тест в строке 318 выявляет ошибку в методе `getFollowingDayOfWeek` (строка 672). 25 декабря 2004 года было субботой. Следующей субботой было 1 января 2005 года. Тем не менее при запуске теста `getFollowingDayOfWeek` утверждает, что первой субботой, предшествующей 25 декабря, было 25 декабря. Разумеется, это неверно [G3],[T1]. Проблема — типичная ошибка граничного условия [T5] — кроется в строке 685. Строка должна читаться следующим образом:

```
685         if (baseDOW >= targetWeekday) {
```

Интересно, что проблемы с этой функцией возникали и раньше. Из истории изменений (строка 43) видно, что в функциях `getPreviousDayOfWeek`, `getFollowingDayOfWeek` и `getNearestDayOfWeek` [T6] «исправлялись ошибки».

Модульный тест `testGetNearestDayOfWeek` (строка 329), проверяющий работу метода `getNearestDayOfWeek` (строка 705), изначально был не таким длинным

и исчерпывающим, как в окончательной версии. Я включил в него много дополнительных тестовых сценариев, потому что не все исходные тесты проходили успешно [T6]. Посмотрите, какие тестовые сценарии были закомментированы — закономерность проявляется достаточно очевидно [T7]. Сбой в алгоритме происходит в том случае, если ближайший день находится в будущем. Очевидно, и здесь происходит какая-то ошибка граничного условия [T5].

Результаты тестового покрытия кода, полученные от Clover, тоже весьма интересны [T8]. Строка 719 никогда не выполняется! Следовательно, условие `if` в строке 718 всегда ложно. С первого взгляда на код понятно, что это действительно так. Переменная `adjust` всегда отрицательна, она не может быть больше либо равна 4. Значит, алгоритм попросту неверен.

Правильный алгоритм выглядит так:

```
int delta = targetDOW - base.getDayOfWeek();
int positiveDelta = delta + 7;
int adjust = positiveDelta % 7;
if (adjust > 3)
  adjust -= 7;
return SerialDate.addDays(adjust, base);
```

Наконец, для прохождения тестов в строках 417 и 429 достаточно инициировать исключение `IllegalArgumentException` вместо возвращения строки ошибки в функциях `weekInMonthToString` и `relativeToString`.

После таких изменений все модульные тесты проходят. Вероятно, класс `SerialDate` теперь действительно работает. Теперь пришло время «довести его до ума».

...Потом очистить код

Мы проанализируем код `SerialDate` от начала до конца, усовершенствуя его в ходе просмотра. Хотя ниже об этом не упоминается, после каждого вносимого изменения выполнялись все модульные тесты JCommon, включая мой доработанный модульный тест `SerialDate`. Итак, вы можете быть уверены в том, что вносимые изменения не нарушают работы JCommon.

Начнем со строки 1. В ней приводятся многословные комментарии с информацией о лицензии, авторских правах, авторах и истории изменений. Бесспорно, существуют некоторые юридические формальности, которые необходимо соблюдать, поэтому авторские права и лицензии должны остаться. С другой стороны, история изменений является пережитком из 1960-х годов. Сегодня у нас имеются системы управления исходным кодом, которые все это делают за нас. Историю следует удалить [C1].

Список импорта, начинающийся в строке 61, следует сократить при помощи конструкций `java.text.*` и `java.util.*` [J1].

К форматированию HTML в Javadoc (строка 67) я отношусь без восторга. Меня беспокоят исходные файлы, написанные более чем на одном языке. В этом ком-

ментарии встречаются четыре языка: Java, английский, Javadoc и HTML [G1]. При таком количестве языков трудно поддерживать порядок в коде. Например, аккуратное размещение строк 71 и 72 теряется при генерировании кода Javadoc, да и кому захочется видеть теги <ul> и <li> в исходном коде? Правильнее было бы просто заключить весь комментарий в теги <pre>, чтобы форматирование в исходном коде сохранилось в Javadoc[1].

Строка 86 содержит объявление класса. Почему этот класс называется `SerialDate`? Почему в нем присутствует слово «serial» — только потому, что класс объявлен производным от `Serializable`? Это выглядит маловероятно.

Не буду держать вас в неведении. Я знаю (или по крайней мере полагаю, что знаю), почему было использовано слово «serial». На это указывают константы `SERIAL_LOWER_BOUND` и `SERIAL_UPPER_BOUND` в строках 98 и 101. Еще более очевидная подсказка содержится в комментарии, начинающемся в строке 830. Класс назван `SerialDate`, потому что его реализация построена на использовании «порядкового номера» (serial number), то есть количества дней с 30 декабря 1899 года.

На мой взгляд, у такого решения два недостатка. Во-первых, термин «порядковый номер» некорректен. Кому-то это покажется пустяком, но выбранное представление представляет собой относительное смещение, а не порядковый номер. Термин «порядковый номер» скорее относится к маркировке промышленных изделий, а не к датам. Так что, на мой взгляд, название получилось не слишком содержательным [N1].

Второй недостаток более важен. Имя `SerialDate` подразумевает определенную реализацию. Однако класс является абстрактным и для него реализацию скорее нужно скрывать! Я считаю, что выбранное имя находится на неверном уровне абстракции [N2]. По моему мнению, класс было бы лучше назвать `Date`.

К сожалению, в библиотеку Java входит слишком много классов с именем `Date`; вероятно, это не лучший вариант. Поскольку класс скорее ориентирован на работу с сутками, я подумывал о том, чтобы назвать его `Day`, но и это имя часто используется в других местах. В конечном итоге я решил, что лучшим компромиссом будет имя `DayDate`.

В дальнейшем обсуждении будет использоваться имя `DayDate`. Не забывайте, что в листингах, на которые вы будете смотреть, класс по-прежнему называется `SerialDate`.

Я понимаю, почему `DayDate` наследует от `Comparable` и `Serializable`. Но почему он наследует от `MonthConstants`? Класс `MonthConstants` (листинг Б.3, с. 417) представляет собой простой набор статических констант, определяющих месяцы. Наследование от классов с константами — старый трюк, который использовался Java-программистами, чтобы избежать выражений вида `MonthConstants.January`,

[1] А еще правильнее было бы считать в Javadoc все комментарии заранее отформатированными, чтобы они одинаково смотрелись в коде и в документации.

но это неудачная мысль [J2]. MonthConstants следовало бы оформить в виде перечисления.

```
public abstract class DayDate implements Comparable,
                                         Serializable {
  public static enum Month {
    JANUARY(1),
    FEBRUARY(2),
    MARCH(3),
    APRIL(4),
    MAY(5),
    JUNE(6),
    JULY(7),
    AUGUST(8),
    SEPTEMBER(9),
    OCTOBER(10),
    NOVEMBER(11),
    DECEMBER(12);
    Month(int index) {
      this.index = index;
    }
    public static Month make(int monthIndex) {
      for (Month m : Month.values()) {
        if (m.index == monthIndex)
          return m;
      }
      throw new IllegalArgumentException("Invalid month index " + monthIndex);
    }
    public final int index;
  }
```

Преобразование MonthConstants в enum инициирует ряд изменений в классе DayDate и всех его пользователях. На внесение всех изменений мне потребовалось около часа. Однако теперь любая функция, прежде получавшая int вместо месяца, теперь получает значение из перечисления Month. Это означает, что мы можем удалить метод isValidMonthCode (строка 326), а также все проверки ошибок кодов месяцев — например, monthCodeToQuarter (строка 356) [G5].

Далее возьмем строку 91, serialVersionUID. Переменная используется для управления сериализацией данных. Если изменить ее, то данные DayDate, записанные старой версией программы, перестанут читаться, а попытки приведут к исключению InvalidClassException. Если вы не объявите переменную serialVersionUID, компилятор автоматически сгенерирует ее за вас, причем значение переменной будет различаться при каждом внесении изменений в модуль. Я знаю, что во всей документации рекомендуется управлять этой переменной вручную, но мне кажется, что автоматическое управление сериализацией надежнее [G4]. В конце концов, я предпочитаю отлаживать исключение InvalidClassException, чем необъяснимое поведение программы в результате того, что я забыл изменить serialVersionUID. Итак, я собираюсь удалить эту переменную — по крайней мере пока.

Комментарий в строке 93 выглядит избыточным. Избыточные комментарии только распространяют лживую и недостоверную информацию [C2]. Соответственно, я удаляю его вместе со всеми аналогами.

В комментариях в строках 97 и 100 упоминаются порядковые номера, о которых говорилось ранее [C1]. Комментарии описывают самую раннюю и самую позднюю дату, представляемую классом DayDate. Их можно сделать более понятными [N1].

```
public static final int EARLIEST_DATE_ORDINAL = 2;     // 1/1/1900
public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
```

Мне неясно, почему значение EARLIEST_DATE_ORDINAL равно 2, а не 0. Комментарий в строке 829 подсказывает, что это как-то связано с представлением дат в Microsoft Excel. Более подробное объяснение содержится в производном от DayDate классе с именем SpreadsheetDate (листинг Б.5, с. 428). Комментарий в строке 71 хорошо объясняет суть дела.

Проблема в том, что такой выбор относится к реализации SpreadsheetDate и не имеет ничего общего с DayDate. Из этого я заключаю, что EARLIEST_DATE_ORDINAL и LATEST_DATE_ORDINAL реально не относятся к DayDate и их следует переместить в SpreadsheetDate [G6].

Поиск по коду показывает, что эти переменные используются только в SpreadsheetDate. Они не используются ни в DayDate, ни в других классах JCommon. Соответственно, я перемещаю их в SpreadsheetDate.

Со следующими переменными, MINIMUM_YEAR_SUPPORTED и MAXIMUM_YEAR_SUPPORTED (строки 104 и 107), возникает дилемма. Вроде бы понятно, что если DayDate является абстрактным классом, то он не должен содержать информации о минимальном или максимальном годе. У меня снова возникло искушение переместить эти переменные в SpreadsheetDate [G6]. Тем не менее поиск показал, что эти переменные используются еще в одном классе: RelativeDayOfWeekRule (листинг Б.6, с. 438). В строках 177 и 178 функция getDate проверяет, что в ее аргументе передается действительный год. Дилемма состоит в том, что пользователю абстрактного класса необходима информация о его реализации.

Наша задача — предоставить эту информацию, не загрязняя самого класса DayDate. В общем случае мы могли бы получить данные реализации из экземпляра производного класса, однако функция getDate не получает экземпляр DayDate. С другой стороны, она возвращает такой экземпляр, а это означает, что она его где-то создает. Из строк 187–205 можно заключить, что экземпляр DayDate создается при вызове одной из трех функций: getPreviousDayOfWeek, getNearestDayOfWeek или getFollowingDayOfWeek. Обратившись к листингу DayDate, мы видим, что все эти функции (строки 638–724) возвращают дату, созданную функцией addDays (строка 571), которая вызывает createInstance (строка 808), которая создает SpreadsheetDate! [G7].

В общем случае базовые классы не должны располагать информацией о своих производных классах. Проблема решается применением паттерна АБСТРАКТ-

НАЯ ФАБРИКА [GOF] и созданием класса DayDateFactory. Фабрика создает экземпляры DayDate, а также предоставляет информацию по поводу реализации — в частности, минимальное и максимальное значение даты.

```
public abstract class DayDateFactory {
  private static DayDateFactory factory = new SpreadsheetDateFactory();
  public static void setInstance(DayDateFactory factory) {
    DayDateFactory.factory = factory;
  }
  protected abstract DayDate _makeDate(int ordinal);
  protected abstract DayDate _makeDate(int day, DayDate.Month month, int year);
  protected abstract DayDate _makeDate(int day, int month, int year);
  protected abstract DayDate _makeDate(java.util.Date date);
  protected abstract int _getMinimumYear();
  protected abstract int _getMaximumYear();
  public static DayDate makeDate(int ordinal) {
    return factory._makeDate(ordinal);
  }
  public static DayDate makeDate(int day, DayDate.Month month, int year) {
    return factory._makeDate(day, month, year);
  }
  public static DayDate makeDate(int day, int month, int year) {
    return factory._makeDate(day, month, year);
  }
  public static DayDate makeDate(java.util.Date date) {
    return factory._makeDate(date);
  }
  public static int getMinimumYear() {
    return factory._getMinimumYear();
  }
  public static int getMaximumYear() {
    return factory._getMaximumYear();
  }
}
```

Фабрика заменяет методы createInstance методами makeDate, в результате чего имена выглядят гораздо лучше [N1]. По умолчанию используется SpreadsheetDateFactory, но этот класс можно в любой момент заменить другой фабрикой. Статические методы, делегирующие выполнение операций абстрактным методам, используют комбинацию паттернов СИНГЛЕТ [GOF], ДЕКОРАТОР [GOF] и АБСТРАКТНАЯ ФАБРИКА.

Класс SpreadsheetDateFactory выглядит так:

```
public class SpreadsheetDateFactory extends DayDateFactory {
  public DayDate _makeDate(int ordinal) {
    return new SpreadsheetDate(ordinal);
  }
  public DayDate _makeDate(int day, DayDate.Month month, int year) {
    return new SpreadsheetDate(day, month, year);
  }
  public DayDate _makeDate(int day, int month, int year) {
    return new SpreadsheetDate(day, month, year);
  }
```

```
  public DayDate _makeDate(Date date) {
    final GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(date);
    return new SpreadsheetDate(
      calendar.get(Calendar.DATE),
      DayDate.Month.make(calendar.get(Calendar.MONTH) + 1),
      calendar.get(Calendar.YEAR));
  }
  protected int _getMinimumYear() {
    return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
  }
  protected int _getMaximumYear() {
    return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
  }
}
```

Как видите, я уже переместил переменные `MINIMUM_YEAR_SUPPORTED` и `MAXIMUM_YEAR_SUPPORTED` в класс `SpreadsheetDate`, в котором им положено находиться [G6].

Следующая проблема `DayDate` — константы дней, начинающиеся со строки 109. Их следует оформить в виде другого перечисления [J3]. Мы уже видели, как это делается, поэтому я не буду повторяться. При желании посмотрите в итоговом листинге.

Далее мы видим серию таблиц, начинающуюся с `LAST_DAY_OF_MONTH` в строке 140. Моя первая претензия к этим таблицам состоит в том, что описывающие их комментарии избыточны [C3]. Одних имен вполне достаточно, поэтому я собираюсь удалить комментарии.

Также неясно, почему эта таблица не объявлена приватной [G8], потому что в классе имеется статическая функция `lastDayOfMonth`, предоставляющая те же данные.

Следующая таблица, `AGGREGATE_DAYS_TO_END_OF_MONTH`, выглядит загадочно — она ни разу не используется в JCommon [G9]. Я удалил ее.

То же произошло с `LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH`.
Следующая таблица, `AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH`, используется только в `SpreadsheetDate` (строки 434 и 473). Так почему бы не переместить ее в `SpreadsheetDate`? Против перемещения говорит тот факт, что таблица не привязана ни к какой конкретной реализации [G6]. С другой стороны, никаких реализаций, кроме `SpreadsheetDate`, фактически не существует, поэтому таблицу следует переместить ближе к месту ее использования [G10].

Для меня решающим обстоятельством является то, что для обеспечения логической согласованности [G11] таблицу следует объявить приватной и предоставить доступ к ней через функцию вида `julianDateOfLastDayOfMonth`. Но похоже, такая функция никому не нужна. Более того, если этого потребует новая реализация `DayDate`, таблицу можно будет легко вернуть на место. Поэтому я ее переместил.

Далее следуют три группы констант, которые можно преобразовать в перечисления (строки 162–205).

Первая из трех групп предназначена для выбора недели в месяце. Я преобразовал ее в перечисление с именем `WeekInMonth`.

```
public enum WeekInMonth {
    FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
    public final int index;

    WeekInMonth(int index) {
      this.index = index;
    }
  }
```

Со второй группой констант (строки 177–187) дело обстоит сложнее. Константы `INCLUDE_NONE`, `INCLUDE_FIRST`, `INCLUDE_SECOND` и `INCLUDE_BOTH` определяют, должны ли включаться в диапазон конечные даты. В математике в подобных случаях используются термины «открытый интервал», «полуоткрытый интервал» и «замкнутый интервал». Мне кажется, что математические названия выглядят более понятно [N3], поэтому я преобразовал группу в перечисление `DateInterval` с элементами `CLOSED`, `CLOSED_LEFT`, `CLOSED_RIGHT` и `OPEN`.

Третья группа констант (строки 18–205) определяет, должно ли в результате поиска конкретного дня недели возвращаться последнее, предыдущее или ближайшее вхождение. Выбрать подходящее имя для такого перечисления непросто. В итоге я остановился на имени `WeekdayRange` с элементами `LAST`, `NEXT` и `NEAREST`.

Возможно, вы не согласитесь с выбранными мной именами. Мне они кажутся логичными, но у вас может быть свое мнение. Однако сейчас константы приведены к форме, которая позволяет легко изменить их в случае необходимости [J3]. Они передаются не в виде целых чисел, а в виде символических имен. Я могу воспользоваться функцией переименования своей рабочей среды для изменения имен или типов, не беспокоясь о том, что я пропустил где-то в коде `-1` или `2` или объявление аргумента `int` осталось плохо описанным.

Поле `description` в строке 208 нигде не используется. Я удалил его вместе с методами доступа [G9].

Также был удален вырожденный конструктор по умолчанию в строке 213 [G12]. Компилятор сгенерирует его за нас.

Метод `isValidWeekdayCode` (строки 216–238) пропускаем — мы удалили его при создании перечисления `Day`.

Мы подходим к методу `stringToWeekdayCode` (строки 242–270). Комментарии Javadoc, не добавляющие полезной информации к сигнатуре метода, только загромождают код [C3],[G12]. В комментарии есть всего один содержательный момент — он описывает возвращаемое значение `-1`. Но после перехода на перечисление `Day` этот комментарий стал неверным [C2]. Сейчас метод сообщает об ошибке, выдавая исключение `IllegalArgumentException`. Я удалил комментарий.

Также я удалил все ключевые слова final в объявлениях аргументов и переменных. На мой взгляд, реальной пользы от них не было, а программу они загромождают [G12]. Удаление final противоречит мнению некоторых экспертов. Например, Роберт Симмонс (Robert Simmons) [Simmons04, p. 73] настоятельно рекомендует «...почаще вставлять final в своем коде». Разумеется, я с этим не согласен. У final имеются свои полезные применения (например, при объявлении отдельных констант), но в остальных случаях это ключевое слово не приносит реальной пользы. Возможно, я так считаю еще и потому, что типичные ошибки, выявляемые при помощи final, уже выявляются написанными мной модульными тестами.

Мне не понравились повторяющиеся команды if [G5] внутри цикла for (строки 259 и 263); они были объединены в одну команду if при помощи оператора ||. Также я использовал перечисление Day для управления циклом for и внес ряд других косметических изменений.

Мне пришло в голову, что метод в действительности не принадлежит DayDate. Фактически это функция разбора Day, поэтому я переместил ее в перечисление Day. Но после этого перечисление Day стало занимать довольно много места. Поскольку концепция Day не зависит от DayDate, я вывел перечисление Day из класса DayDate в собственный исходный файл [G13].

Кроме того, я переместил следующую функцию weekdayCodeToString (строки 272–286) в перечисление Day и переименовал ее в toString.

```
public enum Day {
  MONDAY(Calendar.MONDAY),
  TUESDAY(Calendar.TUESDAY),
  WEDNESDAY(Calendar.WEDNESDAY),s
  THURSDAY(Calendar.THURSDAY),
  FRIDAY(Calendar.FRIDAY),
  SATURDAY(Calendar.SATURDAY),
  SUNDAY(Calendar.SUNDAY);

  public final int index;
  private static DateFormatSymbols dateSymbols = new DateFormatSymbols();

  Day(int day) {
    index = day;
  }

  public static Day make(int index) throws IllegalArgumentException {
    for (Day d : Day.values())
      if (d.index == index)
        return d;
    throw new IllegalArgumentException(
      String.format("Illegal day index: %d.", index));
  }

  public static Day parse(String s) throws IllegalArgumentException {
    String[] shortWeekdayNames =
```

```
      dateSymbols.getShortWeekdays();
    String[] weekDayNames =
      dateSymbols.getWeekdays();

    s = s.trim();
    for (Day day : Day.values()) {
      if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
          s.equalsIgnoreCase(weekDayNames[day.index])) {
        return day;
      }
    }
    throw new IllegalArgumentException(
      String.format("%s is not a valid weekday string", s));
  }

  public String toString() {
    return dateSymbols.getWeekdays()[index];
  }
}
```

В программе две функции `getMonths` (строки 288–316); первая функция вызывает вторую. Вторая функция не вызывается никем, кроме первой функцией. Я свернул две функции в одну, что привело к значительному упрощению кода [G9],[G12],[F4]. В завершение я переименовал итоговую функцию, присвоив ей более содержательное имя [N1].

```
public static String[] getMonthNames() {
  return dateFormatSymbols.getMonths();
}
```

Функция `isValidMonthCode` (строки 326–346) потеряла актуальность после введения перечисления `Month`, поэтому я ее удалил [G9].

Функция `monthCodeToQuarter` (строки 356–375) отдает ФУНКЦИОНАЛЬНОЙ ЗАВИСТЬЮ [Refactoring]; вероятно, ее логичнее включить в перечисление `Month` в виде метода с именем `quarter`. Я выполнил замену.

```
public int quarter() {
  return 1 + (index-1)/3;
}
```

В результате перечисление `Month` стало достаточно большим для выделения в отдельный класс. Я убрал его из `DayDate` по образцу перечисления `Day` [G11],[G13].

Следующие два метода называются `monthCodeToString` (строки 377–426). И снова мы видим, как один метод вызывает своего «двойника» с передачей флага. Обычно передавать флаг в аргументе не рекомендуется, особенно если он просто выбирает формат вывода [G15]. Я переименовал, упростил и реструктурировал эти функции и переместил их в перечисление `Month` [N1],[N3],[C3],[G14].

```
public String toString() {
  return dateFormatSymbols.getMonths()[index - 1];
}
```

```
public String toShortString() {
  return dateFormatSymbols.getShortMonths()[index - 1];
}
```

Далее в листинге идет метод stringToMonthCode (строки 428–472). Я переименовал его, переместил в перечисление Month и упростил [N1],[N3],[C3],[G14],[G12].

```
public static Month parse(String s) {
  s = s.trim();
  for (Month m : Month.values())
    if (m.matches(s))
      return m;

  try {
    return make(Integer.parseInt(s));
  }
  catch (NumberFormatException e) {}
  throw new IllegalArgumentException("Invalid month " + s);
}

private boolean matches(String s) {
  return s.equalsIgnoreCase(toString()) ||
         s.equalsIgnoreCase(toShortString());
}
```

Метод isLeapYear (строки 495–517) можно сделать более выразительным [G16].

```
public static boolean isLeapYear(int year) {
  boolean fourth = year % 4 == 0;
  boolean hundredth = year % 100 == 0;
  boolean fourHundredth = year % 400 == 0;
  return fourth && (!hundredth || fourHundredth);
}
```

Следующая функция, leapYearCount (строки 519–536), не принадлежит DayDate. Она не вызывается никем, кроме двух методов SpreadsheetDate. Я переместил ее в производный класс [G6].

Функция lastDayOfMonth (строки 538–560) использует массив LAST_DAY_OF_MONTH. Этот массив принадлежит перечислению Month [G17], поэтому функция была перемещена. Заодно я упростил ее код и сделал его более выразительным [G16].

```
public static int lastDayOfMonth(Month month, int year) {
  if (month == Month.FEBRUARY && isLeapYear(year))
    return month.lastDay() + 1;
  else
    return month.lastDay();
}
```

Начинается самое интересное. Далее в листинге идет функция addDays (строки 562–576). Прежде всего, поскольку эта функция работает с переменными DayDate, она не должна быть статической [G18]. Соответственно, я преобразовал ее в ме-

тод экземпляра. Также она вызывает функцию `toSerial`, которую правильнее называть `toOrdinal` [N1]. Наконец, метод можно несколько упростить.

```
public DayDate addDays(int days) {
  return DayDateFactory.makeDate(toOrdinal() + days);
}
```

Сказанное относится и к функции `addMonths` (строки 578–602). Она должна быть оформлена в виде метода экземпляра [G18]. Алгоритм относительно сложен, поэтому я воспользовался ПОЯСНИТЕЛЬНЫМИ ВРЕМЕННЫМИ ПЕРЕМЕННЫМИ [Beck97] [G19], чтобы сделать его смысл более прозрачным. Заодно метод `getYYY` был переименован в `getYear` [N1].

```
public DayDate addMonths(int months) {
  int thisMonthAsOrdinal = 12 * getYear() + getMonth().index - 1;
  int resultMonthAsOrdinal = thisMonthAsOrdinal + months;
  int resultYear = resultMonthAsOrdinal / 12;
  Month resultMonth = Month.make(resultMonthAsOrdinal % 12 + 1);
  int lastDayOfResultMonth = lastDayOfMonth(resultMonth, resultYear);
  int resultDay = Math.min(getDayOfMonth(), lastDayOfResultMonth);
  return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
}
```

Функция `addYears` (строки 604–626) преобразуется по тем же принципам, что и ее аналоги.

```
public DayDate plusYears(int years) {
  int resultYear = getYear() + years;
  int lastDayOfMonthInResultYear = lastDayOfMonth(getMonth(), resultYear);
  int resultDay = Math.min(getDayOfMonth(), lastDayOfMonthInResultYear);
  return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
}
```

Преобразование статических методов в методы экземпляров вызвало у меня некоторое беспокойство. Поймет ли читатель при виде выражения `date.addDays(5)`, что объект `date` не изменяется, а вместо этого возвращается новый экземпляр `DayDate`? Или он ошибочно решит, что к объекту `date` прибавляются пять дней? Казалось бы, проблема не столь серьезна, но конструкции вроде следующей могут оказаться очень коварными [G20].

```
DayDate date = DateFactory.makeDate(5, Month.DECEMBER, 1952);
date.addDays(7); // Смещение date на одну неделю.
```

Скорее всего, читатель кода предположит, что вызов `addDays` изменяет объект `date`. Значит, нам понадобится имя, разрушающее эту двусмысленность [N4]. Я переименовал методы в `plusDays` и `plusMonths`. Мне кажется, что предназначение данного метода отлично отражается конструкциями вида

```
DayDate date = oldDate.plusDays(5);
```

С другой стороны, следующая конструкция читается недостаточно бегло, чтобы читатель сразу предположил, что изменяется объект `date`:

```
date.plusDays(5);
```

Алгоритмы становятся все интереснее. Функция `getPreviousDayOfWeek` (строки 628–660) работает, но выглядит слишком сложно. После некоторых размышлений относительно того, что же в действительности происходит в этой функции [G21], мне удалось упростить ее и воспользоваться ПОЯСНИТЕЛЬНЫМИ ВРЕМЕННЫМИ ПЕРЕМЕННЫМИ [G19], чтобы сделать код более понятным. Я также преобразовал статический метод в метод экземпляра [G18] и избавился от дублирующего метода экземпляра [G5] (строки 997–1008).

```
public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
  int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;
  if (offsetToTarget >= 0)
    offsetToTarget -= 7;
  return plusDays(offsetToTarget);
}
```

Абсолютно такой же анализ с тем же результатом был проведен для метода `getFollowingDayOfWeek` (строки 662–693).

```
public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
  int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;
  if (offsetToTarget <= 0)
    offsetToTarget += 7;
  return plusDays(offsetToTarget);
}
```

Далее идет функция `getNearestDayOfWeek` (строки 695–726), которую мы исправляли на с. 309. Внесенные тогда изменения не соответствуют тому шаблону, по которому были преобразованы две последние функции [G11]. Я преобразовал функцию по тем же правилам, а также воспользовался ПОЯСНИТЕЛЬНЫМИ ВРЕМЕННЫМИ ПЕРЕМЕННЫМИ [G19] для разъяснения алгоритма.

```
public DayDate getNearestDayOfWeek(final Day targetDay) {
  int offsetToThisWeeksTarget = targetDay.index - getDayOfWeek().index;
  int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
  int offsetToPreviousTarget = offsetToFutureTarget - 7;

  if (offsetToFutureTarget > 3)
    return plusDays(offsetToPreviousTarget);
  else
    return plusDays(offsetToFutureTarget);
}
```

Метод `getEndOfCurrentMonth` (строки 728–740) выглядит немного странно — перед нами метод экземпляра, который «завидует» [G14] собственному классу, получая аргумент `DayDate`. Я преобразовал его в полноценный метод экземпляра, а также заменил несколько имен более содержательными.

```
public DayDate getEndOfMonth() {
  Month month = getMonth();
  int year = getYear();
```

```
  int lastDay = lastDayOfMonth(month, year);
  return DayDateFactory.makeDate(lastDay, month, year);
}
```

Переработка `weekInMonthToString` (строки 742–761) оказалась очень интересным делом. Используя средства рефакторинга своей IDE, я сначала переместил метод в перечисление `WeekInMonth`, созданное ранее на с. 312. Затем я переименовал его в `toString` и преобразовал из статического метода в метод экземпляра. Все тесты прошли успешно. (Догадываетесь, к чему я клоню?)

Затем я полностью удалил метод! Пять проверок завершились неудачей (строки 411–415, листинг Б.4, с. 417). Я изменил эти строки так, чтобы в них использовались имена из перечисления (`FIRST`, `SECOND`, . . .). И все тесты прошли. А вы догадываетесь, почему? И понимаете ли вы, почему каждый из этих шагов был необходим? Функция рефакторинга проследила за тем, чтобы все предыдущие вызовы `weekInMonthToString` были заменены вызовами `toString` для перечисления `weekInMonth`, а во всех перечислениях реализация `toString` просто возвращает имена...

К сожалению, мои ухищрения ни к чему не привели. Как бы элегантно ни выглядела эта замечательная цепочка рефакторинга, в итоге я понял, что единственными пользователями этой функции были тесты, которые я только что изменил. И я удалил тесты.

Стыдно дважды наступать на одни грабли! Определив, что функция `relativeToString` (строки 765–781) не вызывается нигде, кроме тестов, я просто удалил функцию вместе с ее тестами.

Наконец-то мы добрались до абстрактных методов абстрактного класса. Первый метод выглядит знакомо: `toSerial` (строки 838–844). На с. 316 я присвоил ему имя `toOrdinal`. Рассматривая его в новом контексте, я решил, что его лучше переименовать в `getOrdinalDay`.

Следующий абстрактный метод, `toDate` (строки 838–844), преобразует `DayDate` в `java.util.Date`. Почему метод объявлен абстрактным? Присмотревшись к его реализации в `SpreadsheetDate` (строки 198–207, листинг Б.5, с. 428), мы видим, что он не зависит ни от каких подробностей реализации класса [G6]. Поэтому я поднял его на более высокий уровень абстракции.

Методы `getYYYY`, `getMonth` и `getDayOfMonth` абстрактны. Метод `getDayOfWeek` — еще один метод, который следовало бы извлечь из `SpreadSheetDate` — тоже не зависит от `DayDate`. Или все-таки зависит? Присмотревшись внимательно (строка 247, листинг Б.5, с. 428), мы видим, что алгоритм неявно зависит от «точки отсчета» дней недели (иначе говоря, от того, какой день недели считается днем 0). Таким образом, хотя функция не имеет физических зависимостей, которые нельзя было бы переместить в `DayDate`, у нее имеются логические зависимости.

Подобные логические зависимости беспокоят меня [G22]. Если что-то зависит от реализации на логическом уровне, то что-то должно зависеть и на физическом уровне. Кроме того, мне кажется, что сам алгоритм можно было бы сделать более

универсальным, чтобы существенно меньшая его часть зависела от реализации [G6].

Я создал в DayDate абстрактный метод с именем getDayOfWeekForOrdinalZero и реализовал его в SpreadsheetDate так, чтобы он возвращал Day.SATURDAY. Затем я переместил метод getDayOfWeek наверх по цепочке в DayDate и изменил его так, чтобы в нем вызывались методы getOrdinalDay и getDayOfWeekForOrdinalZero.

```
public Day getDayOfWeek() {
  Day startingDay = getDayOfWeekForOrdinalZero();
  int startingOffset = startingDay.index - Day.SUNDAY.index;
  return Day.make((getOrdinalDay() + startingOffset) % 7 + 1);
}
```

Заодно присмотритесь к комментарию в строках с 895 по 899. Так ли необходимо это повторение? Как и в предыдущих случаях, я удалил этот комментарий вместе со всеми остальными.

Переходим к следующему методу compare (строки 902–913). Уровень абстракции этого метода снова выбран неправильно [G6], поэтому я поднял его реализацию в DayDate. Кроме того, его имя недостаточно содержательно [N1]. В действительности этот метод возвращает промежуток в днях, начиная с аргумента, поэтому я переименовал его в daysSince. Также я заметил, что для этого метода нет ни одного теста, и написал их.

Следующие шесть функций (строки 915–980) представляют собой абстрактные методы, которые должны реализовываться в DayDate. Я извлек из SpreadsheetDate.

Последнюю функцию isInRange (строки 982–995) также необходимо извлечь и переработать. Команда switch выглядит некрасиво [G23]; ее можно заменить, переместив условия в перечисление DateInterval.

```
public enum DateInterval {
  OPEN {
    public boolean isIn(int d, int left, int right) {
      return d > left && d < right;
    }
  },
  CLOSED_LEFT {
    public boolean isIn(int d, int left, int right) {
      return d >= left && d < right;
    }
  },
  CLOSED_RIGHT {
    public boolean isIn(int d, int left, int right) {
      return d > left && d <= right;
    }
  },
  CLOSED {
    public boolean isIn(int d, int left, int right) {
      return d >= left && d <= right;
    }
  };
```

```
  public abstract boolean isIn(int d, int left, int right);
}

public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
  int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
  int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
  return interval.isIn(getOrdinalDay(), left, right);
}
```

Мы подошли к концу класса `DayDate`. Сейчас я еще раз пройдусь по всему классу и напомню, что было сделано.

Открывающий комментарий был слишком длинным и неактуальным; я сократил и доработал его [C2].

Затем все оставшиеся перечисления были выделены в отдельные файлы [G12].

Статическая переменная (`dateFormatSymbols`) и три статических метода (`getMonthNames`, `isLeapYear`, `lastDayOfMonth`) были выделены в новый класс с именем `DateUtil` [G6].

Абстрактные методы были перемещены на более высокий уровень абстракции, где они были более уместными [G24].

Я переименовал `Month.make` в `Month.fromInt` [N1] и проделал то же самое для всех остальных перечислений.

Для всех перечислений был создан метод доступа `toInt()`, а поле `index` было объявлено приватным.

В `plusYears` и `plusMonths` присутствовало дублирование кода [G5], которое мне удалось устранить введением нового метода `correctLastDayOfMonth`. При этом код всех трех методов стал более понятным.

«Волшебное число» 1 [G25] было заменено соответствующей конструкцией `Month.JANUARY.toInt()` или `Day.SUNDAY.toInt()`. Я потратил некоторое время на доработку класса `SpreadsheetDate` и чистку алгоритмов. Конечный результат представлен в листингах с Б.7 (с. 442) по Б.16 (с. 451).

Интересно заметить, что покрытие кода в `DayDate` *уменьшилось* до 84,9 %! Это объясняется не снижением объема тестируемой функциональности; просто класс сократился, и несколько непокрытых строк имеют больший удельный вес. В классе `DayDate` тесты покрывают 45 из 53 исполняемых команд. Непокрытые строки настолько тривиальны, что не нуждаются в тестировании.

Заключение

Мы снова последовали «правилу бойскаута»: код стал немного чище, чем был до нашего прихода. На это потребовалось время, но результат того стоил. Тестовое покрытие кода увеличилось, были исправлены некоторые ошибки, код стал

чище и компактнее. Хочется верить, что следующему человеку, который будет читать этот код, будет проще в нем разобраться, чем нам. И возможно, этот человек сможет сделать этот код еще чище, чем удалось нам.

Литература

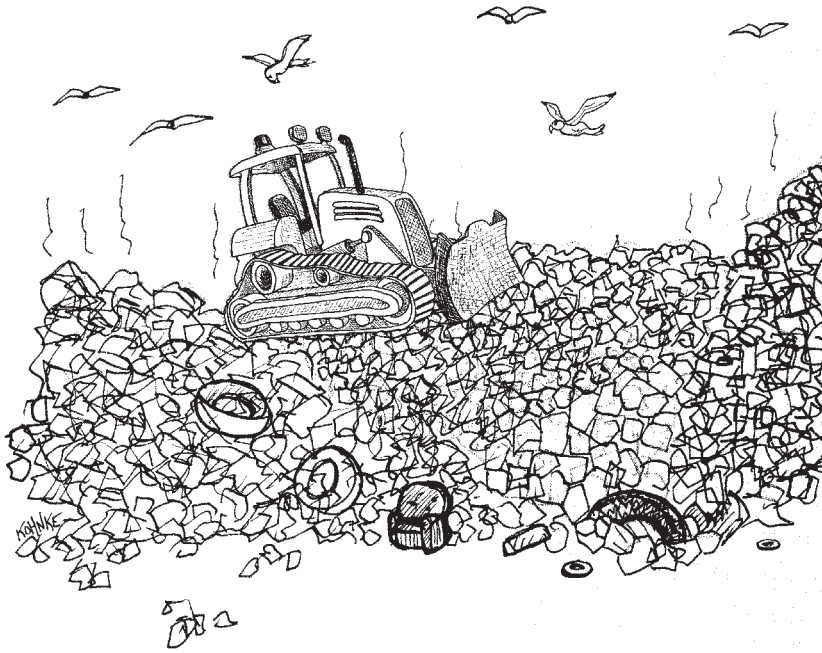
[GOF]: Design Patterns: Elements of Reusable Object Oriented Software, Gamma et al., Addison-Wesley, 1996.

[Simmons04]: Hardcore Java, Robert Simmons, Jr., O'Reilly, 2004.

[Refactoring]: Refactoring: Improving the Design of Existing Code, Martin Fowler et al., Addison-Wesley, 1999.

[Beck97]: Smalltalk Best Practice Patterns, Kent Beck, Prentice Hall, 1997.

17 Запахи и эвристические правила



В своей замечательной книге «Refactoring» [Refactporing] Мартин Фаулер описывает много различных «запахов кода». Следующий список содержит много «запахов», предложенных Мартином, а также ряд моих собственных дополнений. Кроме того, в него были включены некоторые приемы и эвристические правила, которые я часто применяю в своей работе.

Чтобы построить этот список, я просмотрел и переработал несколько разных программ. При внесении каждого изменения я спрашивал себя, почему я это делаю, и записывал результат. Так появился довольно длинный список того, что, на мой взгляд, «дурно пахнет» при чтении кода.

Предполагается, что вы будете читать список от начала к концу, а также использовать его как краткий справочник. Обратитесь к приложению В на с. 455, где собраны перекрестные ссылки, указывающие, где в тексте книги упоминалось то или иное эвристическое правило.

Комментарии

C1: Неуместная информация

В комментариях неуместно размещать информацию, которую удобнее хранить в других источниках: в системах управления исходным кодом, в системах контроля версий и в других системах протоколирования. Например, история изменений только загромождает исходные файлы длинным историческим и малоинтересным текстом. Метаданные (авторы, дата последней модификации и т. д.) в общем случае также неуместны в комментариях. Комментарии должны быть зарезервированы для технической информации о коде и его архитектуре.

C2: Устаревший комментарий

Комментарий, содержимое которого потеряло актуальность, считается устаревшим. Комментарии стареют довольно быстро. Не пишите комментарии, которые с течением времени устареют. Обнаружив устаревший комментарий, обновите его или избавьтесь от него как можно быстрее. Устаревшие комментарии часто «отрываются» от кода, который они когда-то описывали. Так в вашем коде появляются плавучие островки недостоверности и бесполезности.

C3: Избыточный комментарий

Избыточным считается комментарий, описывающий то, что и так очевидно. Например:

```
i++; // Увеличение переменной i
```

Или другой пример — комментарий Javadoc, который содержит не больше (а вернее, меньше) полезной информации, чем простая сигнатура функции:

```
/**
 * @param sellRequest
 * @return
 * @throws ManagedComponentException
 */
public SellResponse beginSellItem(SellRequest sellRequest)
  throws ManagedComponentException
```

Комментарии должны говорить то, что не может сказать сам код.

C4: Плохо написанный комментарий

Если уж вы беретесь за написание комментария, напишите его хорошо. Не жалейте времени и позаботьтесь о том, чтобы это был лучший комментарий, который вы способны создать. Тщательно выбирайте слова. Следите за правильностью орфографии и пунктуации. Не пишите сумбурно. Не объясняйте очевидное. Будьте лаконичны.

C5: Закомментированный код

Фрагменты закомментированного кода выводят меня из себя. Кто знает, когда был написан этот код? Кто знает, есть от него какая-нибудь польза или нет? Однако никто не удаляет закомментированный код — все считают, что он понадобится кому-то другому.

Этот код только попусту занимает место, «загнивая» и утрачивая актуальность с каждым днем. В нем вызываются несуществующие функции. В нем используются переменные, имена которых давно изменились. В нем соблюдаются устаревшие конвенции. Он загрязняет модуль, в котором он содержится, и отвлекает людей, которые пытаются его читать. Закомментированный код *отвратителен*!

Увидев закомментированный код, *удалите его*! Не беспокойтесь, система управления исходным кодом его не забудет. Если кому-то этот код действительно понадобится, то он сможет вернуться к предыдущей версии. Не позволяйте закомментированному коду портить вам жизнь.

Рабочая среда

E1: Построение состоит из нескольких этапов

Построение проекта должно быть одной тривиальной операцией. Без выборки многочисленных фрагментов из системы управления исходным кодом. Без длинных серий невразумительных команд или контекстно-зависимых сценариев для построения отдельных элементов. Без поиска дополнительных файлов в формате JAR, XML и других артефактов, необходимых для вашей системы. Сначала вы проверяете систему одной простой командой, а потом вводите другую простую команду для ее построения.

```
svn get mySystem
cd mySystem
ant all
```

E2: Тестирование состоит из нескольких этапов

Все модульные тесты должны выполняться всего одной командой. В лучшем случае все тесты запускаются одной кнопкой в IDE. В худшем случае одна простая команда вводится в командной строке. Запуск всех тестов — настолько важная и фундаментальная операция, что она должна быть быстрой, простой и очевидной.

Функции

F1: Слишком много аргументов

Функции должны иметь небольшое количество аргументов. Лучше всего, когда аргументов вообще нет; далее следуют функции с одним, двумя и тремя аргументами. Функции с четырьмя и более аргументами весьма сомнительны; старайтесь не использовать их в своих программах (см. «Аргументы функций» на с. 64).

F2: Выходные аргументы

Выходные аргументы противоестественны. Читатель кода ожидает, что аргументы используются для передачи входной, а не выходной информации. Если ваша функция должна изменять чье-либо состояние, пусть она изменяет состояние объекта, для которого она вызывалась (см. «Выходные аргументы», с. 70).

F3: Флаги в аргументах

Логические аргументы явно указывают на то, что функция выполняет более одной операции. Они сильно запутывают код. Исключите их из своих программ (см. «Аргументы-флаги», с. 66).

F4: Мертвые функции

Если метод ни разу не вызывается в программе, то его следует удалить. Хранить «мертвый код» расточительно. Не бойтесь удалять мертвые функции. Не забудьте, что система управления исходным кодом позволит восстановить их в случае необходимости.

Разное

G1: Несколько языков в одном исходном файле

Современные среды программирования позволяют объединять в одном исходном файле код, написанный на разных языках. Например, исходный файл на языке Java может содержать вставки XML, HTML, YAML, JavaDoc, English, JavaScript и т. д. Или, скажем, наряду с кодом HTML в файле JSP может присутствовать код Java, синтаксис библиотеки тегов, комментарии на английском языке, комментарии Javadoc, XML, JavaScript и т. д. В лучшем случае результат получается запутанным, а в худшем — неаккуратным и ненадежным.

В идеале исходный файл должен содержать код на одном — и только одном! — языке. На практике без смешения языков обойтись, скорее всего, не удастся. Но по крайней мере следует свести к минимуму как количество, так и объем кода на дополнительных языках в исходных файлах.

G2: Очевидное поведение не реализовано

Согласно «принципу наименьшего удивления[1]», любая функция или класс должны реализовать то поведение, которого от них вправе ожидать программист. Допустим, имеется функция, которая преобразует название дня недели в элемент перечисления, представляющий этот день.

```
Day day = DayDate.StringToDay(String dayName);
```

Логично ожидать, что строка `"Monday"` будет преобразована в `Day.MONDAY`. Также можно ожидать, что будут поддерживаться стандартные сокращения дней недели, а регистр символов будет игнорироваться.

Если очевидное поведение не реализовано, читатели и пользователи кода перестают полагаться на свою интуицию в отношении имен функций. Они теряют доверие к автору кода и им приходится разбираться во всех подробностях реализации.

G3: Некорректное граничное поведение

Код должен работать правильно — вроде бы очевидное утверждение. Беда в том, что мы редко понимаем, насколько сложным бывает правильное поведение. Разработчики часто пишут функции, которые в их представлении работают, а затем доверяются своей интуиции вместо того, чтобы тщательно проверить работоспособность своего кода во всех граничных и особых ситуациях.

Усердие и терпение ничем не заменить. Каждая граничная ситуация, каждый необычный и особый случай способны нарушить работу элегантного и интуитивного алгоритма. Не полагайтесь на свою интуицию. Найдите каждое граничное условие и напишите для него тест.

G4: Отключенные средства безопасности

Авария на Чернобыльской станции произошла из-за того, что директор завода отключил все механизмы безопасности, один за другим. Они усложняли проведение эксперимента. Результат — эксперимент так и не состоялся, а мир столкнулся с первой серьезной катастрофой в гражданской атомной энергетике.

---

[1] http://en.wikipedia.org/wiki/Principle_of_least_astonishment

Отключать средства безопасности рискованно. Ручное управление `serialVersionUID` бывает необходимо, но оно всегда сопряжено с риском. Иногда отключение некоторых (или всех!) предупреждений компилятора позволяет успешно построить программу, но при этом вы рискуете бесконечными отладочными сеансами. Не отключайте сбойные тесты, обещая себе, что вы заставите их проходить позднее, — это так же неразумно, как считать кредитную карту источником бесплатных денег.

G5: Дублирование

Это одно из самых важных правил в книге и к нему следует относиться очень серьезно. Практически каждый автор, пишущий о проектировании программного обеспечения, упоминает это правило. Дэйв Томас (Dave Thomas) и Энди Хант (Andy Hunt) назвали его *принципом DRY* («Don't Repeat Yourself», то есть «не повторяйтесь») [PRAG]. Кент Бек сделал его одним из основных принципов экстремального программирования в формулировке «Один, и только один раз». Рон Джеффрис (Ron Jeffries) ставит это правило на второе место, после требования о прохождении всех тестов.

Каждый раз, когда в программе встречается повторяющийся код, он указывает на упущенную возможность для абстракции. Возможно, дубликат мог бы стать функцией или даже отдельным классом. «Сворачивая» дублирование в подобные абстракции, вы расширяете лексикон языка программирования. Другие программисты могут воспользоваться созданными вами абстрактными концепциями. Повышение уровня абстракции ускоряет программирование и снижает вероятность ошибок.

Простейшая форма дублирования — куски одинакового кода. Программа выглядит так, словно у программиста дрожат руки, и он снова и снова вставляет один и тот же фрагмент. Такие дубликаты заменяются простыми методами.

Менее тривиальная форма дублирования — цепочки `switch/case` или `if/else`, снова и снова встречающиеся в разных модулях и всегда проверяющие одинаковые наборы условий. Вместо них надлежит применять полиморфизм.

Еще сложнее модули со сходными алгоритмами, но содержащие похожих строк кода. Однако дублирование присутствует и в этом случае. Проблема решается применением паттернов ШАБЛОННЫЙ МЕТОД или СТРАТЕГИЯ [GOF].

В сущности, большинство паттернов проектирования, появившихся за последние 15 лет, представляет собой хорошо известные способы борьбы с дублированием. Нормальные формы Кодда устраняют дублирование в схемах баз данных. Само объектно-ориентированное программирование может рассматриваться как стратегия модульной организации кода и устранения дубликатов. Естественно, это относится и к структурному программированию.

Надеюсь, я достаточно четко выразил свою мысль. Ищите и устраняйте дубликаты повсюду, где это возможно.

G6: Код на неверном уровне абстракции

В программировании важную роль играют абстракции, отделяющие высокоуровневые общие концепции от низкоуровневых подробностей. Иногда эта задача решается созданием абстрактных классов, содержащих высокоуровневые концепции, и производных классов, в которых хранятся низкоуровневые концепции. Действуя подобным образом, необходимо позаботиться о том, чтобы разделение было полным. Все низкоуровневые концепции должны быть сосредоточены в производных классах, а все высокоуровневые концепции объединяются в базовом классе.

Например, константы, переменные и вспомогательные функции, относящиеся только к конкретной реализации, исключаются из базового класса. Базовый класс не должен ничего знать о них.

Правило также относится к исходным файлам, компонентам и модулям. Качественное проектирование требует, чтобы концепции разделялись на разных уровнях и размещались в разных контейнерах. Иногда такими контейнерами являются базовые и производные классы; в других случаях это могут быть исходные файлы, модули или компоненты. Но какое бы решение ни было выбрано в конкретном случае, разделение должно быть полным. Высокоуровневые и низкоуровневые концепции не должны смешиваться.

Рассмотрим следующий фрагмент:

```
public interface Stack {
  Object pop() throws EmptyException;
  void push(Object o) throws FullException;
  double percentFull();
  class EmptyException extends Exception {}
  class FullException extends Exception {}
}
```

Функция `percentFull` находится на неверном уровне абстракции. Существует много реализаций стека, в которых концепция заполнения выглядит разумно, однако другие реализации могут *не знать*, до какой степени заполнен стек. Следовательно, эта функция должна располагаться в производном интерфейсе — например, `BoundedStack`.

Возможно, вы думаете, что для неограниченного стека реализация может просто вернуть 0? Проблема в том, что абсолютно неограниченного стека не существует. Вам не удастся предотвратить исключение `OutOfMemoryException`, проверив условие

```
stack.percentFull() < 50.0.
```

Если ваша реализация функции возвращает 0, то она попросту врет.

Суть в том, что ложь и фикции не способны компенсировать неверного размещения абстракций. Разделение абстракций — одна из самых сложных задач, решаемых разработчиками. Если выбор сделан неверно, не надейтесь, что вам удастся найти простое обходное решение.

G7: Базовые классы, зависящие от производных

Самая распространенная причина для разбиения концепций на базовые и производные классы состоит в том, чтобы концепции базового класса, относящиеся к более высокому уровню, были независимы от низкоуровневых концепций производных классов. Следовательно, когда в базовом классе встречаются упоминания имен производных классов, значит, в проектировании что-то сделано не так. В общем случае базовые классы не должны ничего знать о своих производных классах.

Конечно, у этого правила имеются свои исключения. Иногда количество производных классов жестко фиксировано, а в базовом классе присутствует код для выбора между производными классами. Подобная ситуация часто встречается в реализациях конечных автоматов. Однако в этом случае между базовым и производными классами существует жесткая привязка, и они всегда размещаются вместе в одном файле *jar*. В общем случае нам хотелось бы иметь возможность размещения производных и базовых классов в разных файлах *jar*.

Размещение производных и базовых классов в разных файлах *jar*, при котором базовые файлы *jar* ничего не знают о содержимом производных файлов *jar*, позволяет организовать развертывание систем в формате дискретных, независимых компонентов. Если в такие компоненты будут внесены изменения, то они развертываются заново без необходимости повторного развертывания базовых компонентов. Такая архитектура значительно сокращает последствия от вносимых изменений и упрощает сопровождение систем в условиях реальной эксплуатации.

G8: Слишком много информации

Хорошо определенные модули обладают компактными интерфейсами, позволяющими сделать много минимальными средствами. Для плохо определенных модулей характерны широкие, глубокие интерфейсы, которые заставляют пользователя выполнять много разных операций для решения простых задач. Хорошо определенный интерфейс предоставляет относительно небольшое количество функций, поэтому степень логической привязки при его использовании относительно невелика. Плохо определенный интерфейс предоставляет множество функций, которые необходимо вызывать, поэтому его использование сопряжено с высокой степенью логической привязки.

Хорошие разработчики умеют ограничивать интерфейсы своих классов и модулей. Чем меньше методов содержит класс, тем лучше. Чем меньше переменных известно функции, тем лучше. Чем меньше переменных экземпляров содержит класс, тем лучше.

Скрывайте свои данные. Скрывайте вспомогательные функции. Скрывайте константы и временные переменные. Не создавайте классы с большим количеством методов или переменных экземпляров. Не создавайте большого количества защищенных переменных и функций в субклассах. Сосредоточьтесь на создании

очень компактных, концентрированных интерфейсов. Сокращайте логические привязки за счет ограничения информации.

G9: Мертвый код

Мертвым кодом называется код, не выполняемый в ходе работы программы. Он содержится в теле команды `if`, проверяющей невозможное условие. Он содержится в секции `catch` для блока `try`, никогда не инициирующего исключения. Он содержится в маленьких вспомогательных методах, которые никогда не вызываются, или в никогда не встречающихся условиях `switch/case`.

Мертвый код плох тем, что спустя некоторое время он начинает «плохо пахнуть». Чем древнее код, тем сильнее и резче запах. Дело в том, что мертвый код не обновляется при изменении архитектуры. Он компилируется, но не соответствует более новым конвенциям и правилам. Он был написан в то время, когда система была другой. Обнаружив мертвый код, сделайте то, что положено делать в таких случаях: достойно похороните его. Удалите его из системы.

G10: Вертикальное разделение

Переменные и функции должны определяться вблизи от места их использования. Локальные переменные должны объявляться непосредственно перед первым использованием и должны обладать небольшой вертикальной областью видимости. Объявление локальной переменной не должно отдаляться от места ее использования на сотню строк.

Приватные функции должны определяться сразу же после первого использования. Приватные функции принадлежат области видимости всего класса, но вертикальное расстояние между вызовами и определениями все равно должно быть минимальным. Приватная функция должна обнаруживаться простым просмотром кода от места первого использования.

G11: Непоследовательность

Если некая операция выполняется определенным образом, то и все похожие операции должны выполняться так же. Это правило возвращает нас к «принципу наименьшего удивления». Ответственно подходите к выбору новых схем и обозначений, а если уж выбрали — продолжайте следовать им.

Если в функцию включена переменная `response` для хранения данных `HttpServletResponse`, будьте последовательны и используйте такое же имя переменной в других функциях, работающих с объектами `HttpServletResponse`. Если метод называется `processVerificationRequest`, присваивайте похожие имена (например, `processDeletionRequest`) методам, обрабатывающим другие запросы.

Последовательное соблюдение подобных схем и правил существенно упрощает чтение и модификацию кода.

G12: Балласт

Какой прок от конструктора по умолчанию, не имеющего реализации? Он только попусту загромождает код. Неиспользуемые переменные, невызываемые функции, бессодержательные комментарии — все это бесполезный балласт, который следует удалить. Поддерживайте чистоту в своих исходных файлах, следите за их структурой и не допускайте появления балласта.

G13: Искусственные привязки

То, что не зависит друг от друга, не должно объединяться искусственными привязками. Например, обобщенные перечисления не должны содержаться в более конкретных классах, потому что в этом случае информация о конкретном классе должна быть доступна в любой точке приложения, в которой используется перечисление. То же относится и к статическим функциям общего назначения, объявляемым в конкретных классах.

В общем случае искусственной считается привязка между двумя модулями, не имеющая явной, непосредственной цели. Искусственная привязка возникает в результате размещения переменной, константы или функции во временно удобном, но неподходящем месте. Главные причины для появления искусственных привязок — лень и небрежность.

Не жалейте времени — разберитесь, где должно располагаться объявление той или иной функции, константы или переменной. Слишком часто мы размещаем их в удобном месте «под рукой», а потом оставляем там навсегда.

G14: Функциональная зависть

Это один из «запахов кода», описанных у Мартина Фаулера [Refactoring]. Для методов класса должны быть важны переменные и функции того класса, которому они принадлежат, а не переменные и функции других классов. Когда метод использует методы доступа другого объекта для манипуляций с его данными, то он *завидует* области видимости класса этого объекта. Он словно мечтает находиться в другом классе, чтобы иметь прямой доступ к переменным, с которыми он работает. Пример:

```
public class HourlyPayCalculator {
  public Money calculateWeeklyPay(HourlyEmployee e) {
    int tenthRate = e.getTenthRate().getPennies();
    int tenthsWorked = e.getTenthsWorked();
    int straightTime = Math.min(400, tenthsWorked);
    int overTime = Math.max(0, tenthsWorked - straightTime);
    int straightPay = straightTime * tenthRate;
    int overtimePay = (int)Math.round(overTime*tenthRate*1.5);
    return new Money(straightPay + overtimePay);
  }
}
```

Метод `calculateWeeklyPay` обращается к объекту `HourlyEmployee` за данными для обработки. Метод `calculateWeeklyPay` *завидует* области видимости `HourlyEmployee`. Он «желает» получить доступ к внутренней реализации `HourlyEmployee`.

В общем случае от функциональной зависти следует избавиться, потому что она предоставляет доступ к «внутренностям» класса другому классу. Впрочем, иногда функциональная зависть оказывается неизбежным злом. Рассмотрим следующий пример:

```
public class HourlyEmployeeReport {
  private HourlyEmployee employee ;

  public HourlyEmployeeReport(HourlyEmployee e) {
    this.employee = e;
  }

  String reportHours() {
    return String.format(
      «Name: %s\tHours:%d.%1d\n»,
      employee.getName(),
      employee.getTenthsWorked()/10,
      employee.getTenthsWorked()%10);
  }
}
```

Очевидно, метод `reportHours` завидует классу `HourlyEmployee`. С другой стороны, мы не хотим, чтобы класс `HourlyEmployee` знал о формате отчета. Перемещение форматной строки в класс `HourlyEmployee` нарушает некоторые принципы объектно-ориентированного проектирования[1]. Такое размещение привязывает `HourlyEmployee` к формату отчета и делает его уязвимым для изменений в этом формате.

G15: Аргументы-селекторы

Ничто так не раздражает, как висящий в конце вызова функции аргумент `false`. Зачем он здесь? Что изменится, если этот аргумент будет равен `true`? Смысл селектора трудно запомнить, но дело не только в этом — селектор указывает на объединение нескольких функций в одну. Аргументы-селекторы помогают ленивому программисту избежать разбиения большой функции на несколько меньших. Пример:

```
public int calculateWeeklyPay(boolean overtime) {
  int tenthRate = getTenthRate();
  int tenthsWorked = getTenthsWorked();
  int straightTime = Math.min(400, tenthsWorked);
  int overTime = Math.max(0, tenthsWorked - straightTime);
```

[1] А конкретно — принцип единой ответственности, принцип открытости/закрытости и принцип сокрытия реализаций. См. [PPP].

```
  int straightPay = straightTime * tenthRate;
  double overtimeRate = overtime ? 1.5 : 1.0 * tenthRate;
  int overtimePay = (int)Math.round(overTime*overtimeRate);
  return straightPay + overtimePay;
}
```

Функция вызывается с аргументом `true` при оплате сверхурочной работы по полуторному тарифу или с аргументом `false` при оплате по стандартному тарифу. Каждый раз, когда вы встречаете вызов `calculateWeeklyPay(false)`, вам приходится вспоминать, что он означает, и это само по себе неприятно. Но по-настоящему плохо то, что автор поленился использовать решение следующего вида:

```
public int straightPay() {
  return getTenthsWorked() * getTenthRate();
}

public int overTimePay() {
  int overTimeTenths = Math.max(0, getTenthsWorked() - 400);
  int overTimePay = overTimeBonus(overTimeTenths);
  return straightPay() + overTimePay;
}

private int overTimeBonus(int overTimeTenths) {
  double bonus = 0.5 * getTenthRate() * overTimeTenths;
  return (int) Math.round(bonus);
}
```

Конечно, селекторы не обязаны быть логическими величинами. Это могут быть элементы перечислений, целые числа или любые другие типы аргументов, в зависимости от которых выбирается поведение функции. В общем случае лучше иметь несколько функций, чем передавать функции признак для выбора поведения.

G16: Непонятные намерения

Код должен быть как можно более выразительным. Слишком длинные выражения, венгерская запись, «волшебные числа» — все это скрывает намерения автора. Например, приводившаяся ранее функция `overTimePay` могла бы выглядеть и так:

```
public int m_otCalc() {
  return iThsWkd * iThsRte +
    (int) Math.round(0.5 * iThsRte *
      Math.max(0, iThsWkd - 400)
    );
}
```

Такая запись выглядит компактной и плотной, но разбираться в ней — сущее мучение. Не жалейте времени на то, чтобы сделать намерения своего кода максимально прозрачными для читателей.

G17: Неверное размещение

Одно из самых важных решений, принимаемых разработчиком, — выбор места для размещения кода. Например, где следует объявить константу `PI`? В классе `Math`? А может, ей место в классе `Trigonometry`? Или в классе `Circle`?

В игру вступает принцип наименьшего удивления. Код следует размещать там, где читатель ожидает его увидеть. Константа `PI` должна находиться там, где объявляются тригонометрические функции. Константа `OVERTIME_RATE` объявляется в классе `HourlyPayCalculator`.

Иногда мы пытаемся «творчески» подойти к размещению функциональности. Мы размещаем ее в месте, удобном для нас, но это не всегда выглядит естественно для читателя кода. Предположим, потребовалось напечатать отчет с общим количеством отработанных часов. Мы можем просуммировать часы в коде, печатающем отчет, или же накапливать сумму в коде обработки учетных карточек рабочего времени.

Чтобы принять решение, можно посмотреть на имена функций. Допустим, в модуле отчетов присутствует функция с именем `getTotalHours`, а в модуле обработки учетных карточек присутствует функция `saveTimeCard`. Какая из этих двух функций, если судить по имени, наводит на мысль о вычислении суммы? Ответ очевиден.

Очевидно, по соображениям производительности сумму правильнее вычислять при обработке карточек, а не при печати отчета. Все верно, но этот факт должен быть отражен в именах функций. Например, в модуле обработки учетных карточек должна присутствовать функция `computeRunningTotalOfHours`.

G18: Неуместные статические методы

`Math.max(double a, double b)` — хороший статический метод. Он работает не с одним экземпляром; в самом деле, запись вида `new Math().max(a,b)` или даже `a.max(b)` выглядела бы довольно глупо. Все данные, используемые `max`, берутся из двух аргументов, а не из некоего объекта-«владельца». А главное, что метод `Math.max` почти наверняка не потребуется делать полиморфным.

Но иногда мы пишем статические функции, которые статическими быть не должны. Пример:

```
HourlyPayCalculator.calculatePay(employee, overtimeRate)
```

Эта статическая функция тоже выглядит вполне разумно. Она не работает ни с каким конкретным объектом и получает все данные из своих аргументов. Однако нельзя исключать, что эту функцию потребуется сделать полиморфной. Возможно, в будущем потребуется реализовать несколько разных алгоритмов для вычисления почасовой оплаты — скажем, `OvertimeHourlyPayCalculator` и `StraightTimeHourlyPayCalculator`. В этом случае данная функция не может быть статической. Ее следует оформить как нестатическую функцию `Employee`.

В общем случае отдавайте предпочтение нестатическим методам перед статическими. Если сомневаетесь, сделайте функцию нестатической. Если вы твердо уверены, что функция должна быть статической, удостоверьтесь в том, что от нее не потребуется полиморфное поведение.

G19: Используйте пояснительные переменные

Кент Бек писал об этом в своей великой книге «Smalltalk Best Practice Patterns» [Beck97, p. 108], а затем позднее — в столь же великой книге «Implementation Patterns» [Beck07]. Один из самых эффективных способов улучшения удобочитаемости программы заключается в том, чтобы разбить обработку данных на промежуточные значения, хранящиеся в переменных с содержательными именами.

Возьмем следующий пример из FitNesse:

```
Matcher match = headerPattern.matcher(line);
if(match.find())
{
  String key = match.group(1);
  String value = match.group(2);
  headers.put(key.toLowerCase(), value);
}
```

Простое использование пояснительных переменных четко объясняет, что первое совпадение содержит ключ (`key`), а второе — значение (`value`).

Перестараться в применении пояснительных переменных трудно. Как правило, чем больше пояснительных переменных, тем лучше. Поразительно, насколько очевидным иногда становится самый невразумительный модуль от простого разбиения обработки данных на промежуточные значения с удачно выбранными именами.

G20: Имена функций должны описывать выполняемую операцию

Взгляните на следующий код:

```
Date newDate = date.add(5);
```

Как вы думаете, что он делает — прибавляет пять дней к `date`? А может, пять недель или часов? Изменяется ли экземпляр `date`, или функция возвращает новое значение `Date` без изменения старого? По вызову невозможно понять, что делает эта функция.

Если функция прибавляет пять дней с изменением `date`, то она должна называться `addDaysTo` или `increaseByDays`. С другой стороны, если функция возвращает новую дату, смещенную на пять дней, но не изменяет исходного экземпляра `date`, то она должна называться `daysLater` или `daysSince`.

Если вам приходится обращаться к реализации (или документации), чтобы понять, что делает та или иная функция, постарайтесь найти более удачное имя или разбейте функциональность на меньшие функции с более понятными именами.

G21: Понимание алгоритма

Очень много странного кода пишется из-за того, что люди не утруждают себя пониманием алгоритмов. Они заставляют программу работать «грубой силой», набивая ее командами `if` и флагами, вместо того чтобы остановиться и подумать, что же в действительности происходит.

Программирование часто сопряжено с исследованиями. Вы думаете, что знаете подходящий алгоритм для решения задачи, но потом вам приходится возиться с ним, подправлять и затыкать щели, пока вы не заставите его «работать». А как вы определили, что он «работает»? Потому что алгоритм прошел все тесты, которые вы смогли придумать.

В этом подходе нет ничего плохого. Более того, часто только так удается заставить функцию делать то, что она должна делать (по вашему мнению). Однако ограничиться «работой» в кавычках недостаточно.

Прежде чем откладывать в сторону готовую функцию, убедитесь в том, что вы *понимаете*, как она работает. Прохождения всех тестов недостаточно. Вы должны *знать*[1], что ваше решение правильно.

Один из лучших способов достичь этого знания и понимания — разбить функцию на фрагменты настолько чистые и выразительные, что вам станет совершенно очевидно, как работает данная функция.

G22: Преобразование логических зависимостей в физические

Если один модуль зависит от другого, зависимость должна быть не только логической, но и физической. Зависимый модуль не должен делать никаких предположений (иначе говоря, создавать логические зависимости) относительно того модуля, от которого он зависит. Вместо этого он должен явно запросить у этого модуля всю необходимую информацию.

Допустим, вы пишете функцию, которая выводит текстовый отчет об отработанном времени. Класс с именем `HourlyReporter` собирает все данные в удобной форме и передает их классу `HourlyReportFormatter` для вывода (листинг 17.1).

[1] Знать, как работает ваш код, и знать, делает ли алгоритм то, что требуется, — не одно и то же. Не уверены в правильности выбора алгоритма? Нередко это суровая правда жизни. Но если вы не уверены в том, что делает ваш код, то это обычная лень.

Листинг 17.1. HourlyReporter.java

```
public class HourlyReporter {
  private HourlyReportFormatter formatter;
  private List<LineItem> page;
  private final int PAGE_SIZE = 55;

  public HourlyReporter(HourlyReportFormatter formatter) {
    this.formatter = formatter;
    page = new ArrayList<LineItem>();
  }

  public void generateReport(List<HourlyEmployee> employees) {
    for (HourlyEmployee e : employees) {
      addLineItemToPage(e);
      if (page.size() == PAGE_SIZE)
        printAndClearItemList();
    }
    if (page.size() > 0)
      printAndClearItemList();
  }

  private void printAndClearItemList() {
    formatter.format(page);
    page.clear();
  }

  private void addLineItemToPage(HourlyEmployee e) {
    LineItem item = new LineItem();
    item.name = e.getName();
    item.hours = e.getTenthsWorked() / 10;
    item.tenths = e.getTenthsWorked() % 10;
    page.add(item);
  }

  public class LineItem {
    public String name;
    public int hours;
    public int tenths;
  }
}
```

У этого кода имеется логическая зависимость, лишенная физического воплощения. Удастся ли вам найти ее? Речь идет о константе `PAGE_SIZE`. Почему класс `HourlyReporter` должен знать размер страницы? За размер страницы должен отвечать класс `HourlyReportFormatter`.

Факт объявления `PAGE_SIZE` в `HourlyReporter` указывает на неверное размещение [G17]. Разработчик полагает, что классу `HourlyReporter` необходимо знать размер страницы. Такое предположение является логической зависимостью. Работа `HourlyReporter` зависит от того факта, что `HourlyReportFormatter` поддерживает размер страницы до 55. Если какая-то реализация `HourlyReportFormatter` не сможет работать с такими страницами, произойдет ошибка.

Чтобы создать физическое представление этой зависимости, мы можем включить в `HourlyReportFormatter` новый метод с именем `getMaxPageSize()`. В дальнейшем `HourlyReporter` вызывает эту функцию вместо того, чтобы использовать константу `PAGE_SIZE`.

G23: Используйте полиморфизм вместо if/Else или switch/Case

Эта рекомендация может показаться странной, если вспомнить главу 6. Ведь в этой главе говорилось, что в тех частях системы, в которые с большей вероятностью будут добавляться новые функции, а не новые типы, команды `switch` вполне уместны.

Во-первых, команды `switch` чаще всего используются только потому, что они представляют очевидное решение методом «грубой силы», а не самое уместное решение для конкретной ситуации. Таким образом, это эвристическое правило напоминает нам о том, что до применения `switch` следует рассмотреть возможность применения полиморфизма.

Во-вторых, ситуации, в которых состав функций менее стабилен, чем состав типов, встречаются относительно редко. Следовательно, к каждой конструкции `switch` следует относиться с подозрением.

Я использую правило «ОДНОЙ КОМАНДЫ SWITCH»: *для каждого типа выбора программа не должна содержать более одной команды switch. Множественные конструкции switch следует заменять полиморфными объектами.*

G24: Соблюдайте стандартные конвенции

Все рабочие группы должны соблюдать единые стандарты кодирования, основанные на отраслевых нормах. Стандарт кодирования определяет, где объявляются переменные экземпляров; как присваиваются имена классов, методов и переменных; где размещаются фигурные скобки и т. д. Документ с явным описанием этих правил не нужен — сам код служит примером оформления.

Правила должны соблюдаться всеми участниками группы. Это означает, что каждый участник группы должен быть достаточно разумным, чтобы понимать: неважно, как именно размещаются фигурные скобки, если только все согласились размещать их одинаковым образом.

Если вам захочется узнать, какие конвенции оформления использую я, обратитесь к переработанному коду в листингах Б.7 (с. 442) — Б.14.

G25: Заменяйте «волшебные числа» именованными константами

Вероятно, это одно из самых древних правил разработки. Помню, оно встречалось мне еще в 60-х годах, в учебниках COBOL, FORTRAN и PL/1 для начинающих. В общем случае присутствие «сырых» чисел в коде нежелательно. Числа следует скрыть в константах с содержательными именами.

Например, число 86,400 следует скрыть в константе SECONDS_PER_DAY. Если в странице отчета выводится 55 строк, число 55 следует скрыть в константе LINES_PER_PAGE.

Некоторые числа так легко узнаются, что их не обязательно скрывать за именованными константами — при условии, что они используются в сочетании с предельно ясным кодом. Пример:

```
double milesWalked = feetWalked/5280.0;
int dailyPay = hourlyRate * 8;
double circumference = radius * Math.PI * 2;
```

Нужны ли константы FEET_PER_MILE, WORK_HOURS_PER_DAY и TWO в этих примерах? Разумеется, последний случай выглядит особенно абсурдно. В некоторых формулах константы попросту лучше воспринимаются в числовой записи. По поводу WORK_HOURS_PER_DAY можно спорить, потому что законы и нормативы могут изменяться. С другой стороны, формула с числом 8 читается настолько удобно, что мне просто не хочется нагружать читателя кода лишними 17 символами. А число 5280 — количество футов в миле — настолько хорошо известно и уникально, что читатель сразу узнает его, даже если оно будет располагаться вне какого-либо контекста.

Такие константы, как 3.141592653589793, тоже хорошо известны и легко узнаваемы. Однако вероятность ошибки слишком велика, чтобы оставлять их в числовой форме. Встречая значение 3.1415927535890793, вы сразу догадываетесь, что перед вами число π, и не проверяете его (а вы заметили ошибку в одной цифре?). Также мы не хотим, чтобы в программах использовались сокращения 3.14, 3.14159, 3.142 и т. д. К счастью, значение Math.PI уже определено за нас.

Термин «волшебное число» относится не только к числам. Он распространяется на все лексемы, значения которых не являются самодокументирующими. Пример:

```
assertEquals(7777, Employee.find("John Doe").employeeNumber());
```

В этом проверочном условии задействованы два «волшебных числа». Очевидно, первое — 7777, хотя его смысл далеко не так очевиден. Второе «волшебное число» — строка "John Doe". Ее смысл тоже выглядит весьма загадочно.

Оказывается, "John Doe" — имя работника с табельным номером 7777 в тестовой базе данных, созданной нашей группой. Все участники группы знают, как под-

ключаться к этой базе данных. В базе уже хранятся тестовые записи с заранее известными значениями и атрибутами. Также выясняется, что `"John Doe"` — единственный работник с почасовой оплатой в тестовой базе данных. Следовательно, эта проверка должна выглядеть так:

```
assertEquals(
  HOURLY_EMPLOYEE_ID,
  Employee.find(HOURLY_EMPLOYEE_NAME).employeeNumber());
```

G26: Будьте точны

Наивно ожидать, что первая запись, возвращаемая по запросу, является единственной. Использовать числа с плавающей точкой для представления денежных сумм — почти преступление. Отсутствие блокировок и/или управления транзакциями только потому, что вы думаете, что одновременное обновление маловероятно — в лучшем случае халатность. Объявление переменной с типом `ArrayList` там, где более уместен тип `List` — чрезмерное ограничение. Объявление всех переменных защищенными по умолчанию — недостаточное ограничение.

Принимая решение в своем коде, убедитесь в том, что вы действуете предельно точно и аккуратно. Знайте, почему принимается решение, и как вы собираетесь поступать с исключениями из правила. Не ленитесь обеспечивать точность своих решений. Если вы решили вызвать функцию, которая может вернуть `null` — проверьте возвращаемое значение. Если вы запрашиваете из базы данных запись, которая, по вашему мнению, является единственной — проверьте, не вернул ли запрос дополнительные записи. Если вам нужно работать с денежными суммами, используйте целые числа и округляйте результат по действующим правилам. Если в программе существует возможность одновременного объявления, реализуйте ту или иную разновидность блокировки. Неоднозначности и неточности в коде объясняются либо недопониманием, либо ленью. В любом случае от них следует избавиться.

G27: Структура важнее конвенций

Воплощайте архитектурные решения на уровне структуры кода; она важнее стандартов и конвенций. Содержательные имена полезны, но структура, заставляющая пользователя соблюдать установленные правила, важнее. Например, конструкции `switch/case` с хорошо выбранными именами элементов перечисления уступают базовым классам с абстрактными методами. Ничто не вынуждает пользователя применять одинаковую реализацию `switch/case` во всех случаях; с другой стороны, базовые классы заставляют его реализовать все абстрактные методы в конкретных классах.

G28: Инкапсулируйте условные конструкции

В булевской логике достаточно трудно разобраться и вне контекста команд `if` или `while`. Выделите в программе функции, объясняющие намерения условной конструкции. Например, команда

```
if (shouldBeDeleted(timer))
```

выразительнее команды

```
if (timer.hasExpired() && !timer.isRecurrent())
```

G29: Избегайте отрицательных условий

Отрицательные условия немного сложнее для понимания, чем положительные. Таким образом, по возможности старайтесь формулировать положительные условия. Например, запись

```
if (buffer.shouldCompact())
```

предпочтительнее записи

```
if (!buffer.shouldNotCompact())
```

G30: Функции должны выполнять одну операцию

Часто возникает искушение разделить свою функцию на несколько секций для выполнения разных операций. Такие функции выполняют несколько операций; их следует преобразовать в группу меньших функций, каждая из которых выполняет только одну операцию.

Пример:

```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
    }
  }
}
```

Эта функция выполняет сразу три операции: она перебирает всех работников; проверяет, начислены ли работнику какие-то выплаты; и наконец, производит оплату. Код лучше записать в следующем виде:

```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}
```

```
private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```

Каждая из этих функций выполняет только одну операцию (см. «Правило одной операции», с. 59).

G31: Скрытые временны́е привязки

Временны́е привязки часто необходимы, но они не должны скрываться. Структура аргументов функций должна быть такой, чтобы последовательность вызова была абсолютно очевидной. Рассмотрим следующий пример:

```
public class MoogDiver {
  Gradient gradient;
  List<Spline> splines;

  public void dive(String reason) {
    saturateGradient();
    reticulateSplines();
    diveForMoog(reason);
  }
  ...
}
```

Порядок вызова трех функций важен. Сначала вызывается `saturateGradient()`, затем `reticulateSplines()` и только после этого `diveForMoog()`. К сожалению, код не обеспечивает принудительного соблюдения временной привязки. Ничто не мешает другому программисту вызвать `reticulateSplines` до `saturateGradient`, и все кончится исключением `UnsaturatedGradientException`.

Более правильное решение выглядит так:

```
public class MoogDiver {
  Gradient gradient;
  List<Spline> splines;

  public void dive(String reason) {
    Gradient gradient = saturateGradient();
    List<Spline> splines = reticulateSplines(gradient);
    diveForMoog(splines, reason);
  }
  ...
}
```

Временная привязка реализуется посредством создания «эстафеты». Каждая функция выдает результат, необходимый для работы следующей функции, и вызвать эти функции с нарушением порядка сколько-нибудь разумным способом уже не удастся.

Пожалуй, кто-то сочтет, что это увеличивает сложность функций, и это действительно так. Однако дополнительные синтаксические сложности лишь выявляют

реальную сложность ситуации, обусловленную необходимостью согласования по времени.

Обратите внимание: переменные экземпляров остались на своих местах. Предполагается, что они используются приватными методами класса. Использование их в аргументах лишь явно выражает факт существования временной привязки.

G32: Структура кода должна быть обоснована

Структура кода должна выбираться не произвольно, а по строго определенным причинам. Позаботьтесь о том, чтобы эти причины были выражены в структуре кода. Если при чтении кода создается впечатление, что его структура выбрана произвольно, другим пользователям может показаться, что ее можно изменить. Если во всей системе последовательно используется единая структура кода, другие пользователи примут ее и сохранят действующие правила. Например, недавно я занимался объединением изменений в FitNesse и обнаружил, что один из наших авторов использовал следующую запись:

```
public class AliasLinkWidget extends ParentWidget
{
  public static class VariableExpandingWidgetRoot {
    ...
  ...
}
```

Проблема в том, что `VariableExpandingWidgetRoot` незачем находиться в облсти видимости `AliasLinkWidget`. Более того, класс `AliasLinkWidget.VariableExpanding-WidgetRoot` использовался посторонними классами, которые не имели никакого отношения к `AliasLinkWidget`.

Возможно, программист разместил `VariableExpandingWidgetRoot` в `AliasWidget` по соображениям удобства, а может, он действительно полагал, что область видимости этого класса должна находиться внутри области видимости `AliasWidget`. Какими бы причинами он ни руководствовался, результат выглядит необоснованным. Открытые классы, не являющиеся вспомогательными по отношению к другому классу (то есть используемыми только в его внутренних операциях), не должны размещаться внутри других классов. По стандартным правилам такие классы объявляются на верхнем уровне своих пакетов.

G33: Инкапсулируйте граничные условия

Отслеживать граничные условия нелегко. Разместите их обработку в одном месте. Не позволяйте им «растекаться» по всему коду. Не допускайте, чтобы в вашей программе кишели многочисленные `+1` и `-1`. Возьмем простой пример из FIT:

```
if(level + 1 < tags.length)
{
  parts = new Parse(body, tags, level + 1, offset + endTag);
```

```
  body = null;
}
```

Обратите внимание: `level+1` здесь встречается дважды. Это граничное условие, которое следует инкапсулировать в переменной — например, с именем `nextLevel`:

```
int nextLevel = level + 1;
if(nextLevel < tags.length)
{
  parts = new Parse(body, tags, nextLevel, offset + endTag);
  body = null;
}
```

G34: Функции должны быть написаны на одном уровне абстракции

Все команды функции должны быть сформулированы на одном уровне абстракции, который расположен одним уровнем ниже операции, описываемой именем функции. Возможно, это эвристическое правило сложнее всего правильно интерпретировать и соблюдать. Идея достаточно тривиальна, но люди слишком хорошо справляются со смешением разных уровней абстракции. Для примера возьмем следующий код из FitNesse:

```
public String render() throws Exception
{
  StringBuffer html = new StringBuffer("<hr");
  if(size > 0)
    html.append(" size=\"").append(size + 1).append("\"");
  html.append(">");

  return html.toString();
}
```

Разобраться в происходящем несложно. Функция конструирует тег HTML, который рисует на странице горизонтальную линию. Толщина линии задается переменной `size`.

А теперь взгляните еще раз. В этом методе смешиваются минимум два уровня абстракции. Первый уровень — наличие толщины у горизонтальной линии. Второй уровень — синтаксис тега HR. Код позаимствован из модуля `HruleWidget` проекта FitNesse. Модуль распознает строку из четырех и более дефисов и преобразует ее в соответствующий тег HR. Чем больше дефисов, тем больше толщина.

Я переработал этот фрагмент кода так, как показано ниже. Обратите внимание: имя поля `size` изменено в соответствии с его истинным назначением (в нем хранится количество дополнительных дефисов).

```
public String render() throws Exception
{
  HtmlTag hr = new HtmlTag("hr");
  if (extraDashes > 0)
```

```
    hr.addAttribute("size", hrSize(extraDashes));
  return hr.html();
}

private String hrSize(int height)
{
  int hrSize = height + 1;
  return String.format("%d", hrSize);
}
```

Изменение разделяет два уровня абстракции. Функция `render` просто конструирует тег HR, ничего не зная о синтаксисе HTML этого тега. Модуль `HtmlTag` берет на себя все хлопоты с синтаксисом.

Более того, при внесении этого изменения я обнаружил неприметную ошибку. Исходный код не закрывал тег HR косой чертой, как того требует стандарт XHTML (иначе говоря, он выдавал `<hr>` вместо `<hr/>`), хотя модуль `HtmlTag` был давно приведен в соответствие со стандартом XHTML.

Разделение уровней абстракции — одна из самых важных и одновременно самых сложных в реализации функций рефакторинга. В качестве примера возьмем следующий код — мою первую попытку разделения уровней абстракции в методе `HruleWidget.render`.

```
public String render() throws Exception
{
  HtmlTag hr = new HtmlTag("hr");
  if (size > 0) {
    hr.addAttribute("size", ""+(size+1));
  }
  return hr.html();
}
```

На этой стадии я стремился к тому, чтобы создать необходимое разделение, и обеспечить прохождение тестов. Мне удалось легко добиться этой цели, но в созданной функции по-прежнему смешивались разные уровни абстракции — на этот раз конструирование тега HR и интерпретация/форматирование переменной `size`. Таким образом, при разбиении функции по уровням абстракции иногда обнаруживаются новые уровни, скрытые прежней структурой.

G35: Храните конфигурационные данные на высоких уровнях

Если в программе имеется константа, определяющая значение по умолчанию или параметр конфигурации, и эта константа известна на высоких уровнях абстракции, — не прячьте ее в низкоуровневой функции. Передайте ее в аргументе низкоуровневой функции, вызываемой из функции высокого уровня. Рассмотрим пример из FitNesse.

```
  public static void main(String[] args) throws Exception
  {
    Arguments arguments = parseCommandLine(args);
    ...
  }
public class Arguments
{
  public static final String DEFAULT_PATH = ".";
  public static final String DEFAULT_ROOT = "FitNesseRoot";
  public static final int DEFAULT_PORT = 80;
  public static final int DEFAULT_VERSION_DAYS = 14;
  ...
}
```

Аргументы командной строки разбираются в самой первой исполняемой строке FitNesse. Значения аргументов по умолчанию задаются в начале класса `Argument`. Читателю не приходится спускаться на нижние уровни системы за командами следующего вида:

```
if (arguments.port == 0) // 80 по умолчанию
```

Конфигурационные константы находятся на очень высоком уровне. Если потребуется, их можно легко изменить. Их значения передаются на более низкие уровни иерархии другим компонентам приложения. Значения этих констант не принадлежат нижним уровням приложения.

G36: Избегайте транзитивных обращений

В общем случае модуль не должен обладать слишком полной информацией о тех компонентах, с которыми он взаимодействует. Точнее, если A взаимодействует с B, а B взаимодействует с C, то модули, использующие A, не должны знать о C (то есть нежелательны конструкции вида `a.getB().getC().doSomething();`). Иногда это называется «законом Деметры». Прагматичные программисты используют термин «умеренный код» [PRAG, с. 138].

В любом случае все сводится к тому, что модули должны обладать информацией только о тех модулях, с которыми они непосредственно взаимодействуют, а не располагать навигационной картой всей системы.

Если в нескольких модулях используется та или иная форма команды `a.getB().getC()`, то в дальнейшем вам будет трудно изменить архитектуру системы, вставив между B и C промежуточный компонент Q. Придется найти каждое вхождение `a.getB().getC()` и преобразовать его в `a.getB().getQ().getC()`. Так образуются жесткие, закостеневшие архитектуры. Слишком многие модули располагают слишком подробной информацией о системе.

Весь необходимый сервис должен предоставляться компонентами, с которыми напрямую взаимодействует модуль. Не заставляйте пользователя странствовать по графу объектов системы в поисках нужного метода. Проблема должна решаться простыми вызовами вида

```
myCollaborator.doSomething()
```

Java

J1: Используйте обобщенные директивы импорта

Если вы используете два и более класса из пакета, импортируйте весь пакет командой

```
import package.*;
```

Длинные списки импорта пугают читателя кода. Начало модуля не должно загромождаться 80-строчным списком директив импорта. Список импорта должен быть точной и лаконичной конструкцией, показывающей, с какими пакетами мы собираемся работать.

Конкретные директивы импорта определяют жесткие зависимости, обобщенные директивы импорта — нет. Если вы импортируете конкретный класс, то этот класс обязательно должен существовать. Но пакет, импортируемый обобщенной директивой, может не содержать ни одного класса. Директива импорта просто добавляет пакет в путь поиска имен. Таким образом, обобщенные директивы импорта не создают реальных зависимостей, а следовательно, способствуют смягчению логических привязок между модулями.

В некоторых ситуациях длинные списки конкретных директив импорта бывают полезными. Например, если вы работаете с унаследованным кодом и хотите узнать, для каких классов необходимо создать заглушки и имитации, можно пройтись по конкретным спискам импорта, узнать полные имена классов и написать для них соответствующие заглушки. Однако такое использование конкретных директив импорта встречается крайне редко. Более того, многие современные IDE позволяют преобразовать обобщенный список импорта в список конкретных директив одной командой. Таким образом, даже в унаследованном коде лучше применять обобщенный импорт.

Обобщенные директивы импорта иногда становятся причиной конфликтов имен и неоднозначностей. Два класса с одинаковыми именами, находящиеся в разных пакетах, должны импортироваться конкретными директивами (или по крайней мере их имена должны уточняться при использовании). Это создает определенные неудобства, однако ситуация встречается достаточно редко, так что в общем случае обобщенные директивы импорта все равно лучше конкретных.

J2: Не наследуйте от констант

Я уже неоднократно встречался с этим явлением, и каждый раз оно заставляло меня недовольно поморщиться. Программист размещает константы в интерфейсе, а затем наследует от этого интерфейса для получения доступа к константам. Взгляните на следующий код:

```
public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  private double hourlyRate;
```

```
  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
      hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  ...
}
```

Где определяются константы `TENTHS_PER_WEEK` и `OVERTIME_RATE`? Возможно, в классе `Employee`; давайте посмотрим:

```
public abstract class Employee implements PayrollConstants {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
```

Нет, не здесь. А где тогда? Присмотритесь повнимательнее к классу `Employee`. Он реализует интерфейс `PayrollConstants`.

```
public interface PayrollConstants {
  public static final int TENTHS_PER_WEEK = 400;
  public static final double OVERTIME_RATE = 1.5;
}
```

Совершенно отвратительная привычка! Константы скрыты на верхнем уровне иерархии наследования. Брр! Наследование не должно применяться для того, чтобы обойти языковые правила видимости. Используйте статическое импортирование.

```
import static PayrollConstants.*;

public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  private double hourlyRate;

  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
      hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  ...
}
```

J3: Константы против перечислений

В языке появились перечисления (Java 5) — пользуйтесь ими! Не используйте старый трюк с `public static final int`. Смысл `int` может потеряться; смысл перечислений потеряться не может, потому что они принадлежат указанному перечислению.

Тщательно изучите синтаксис перечислений. Не забудьте, что перечисления могут содержать методы и поля. Это очень мощные синтаксические инструменты, значительно превосходящие int по гибкости и выразительности. Рассмотрим следующую разновидность кода начисления зарплаты:

```
public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  HourlyPayGrade grade;

  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
      grade.rate() * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  ...
}

public enum HourlyPayGrade {
  APPRENTICE {
    public double rate() {
      return 1.0;
    }
  },
  LEUTENANT_JOURNEYMAN {
    public double rate() {
      return 1.2;
    }
  },
  JOURNEYMAN {
    public double rate() {
      return 1.5;
    }
  },
  MASTER {
    public double rate() {
      return 2.0;
    }
  };
  public abstract double rate();
}
```

Имена

N1: Используйте содержательные имена

Не торопитесь с выбором имен. Позаботьтесь о том, чтобы имена были содержательными. Помните, что смысл может изменяться в ходе развития про-

граммного продукта; почаще переосмысливайте уместность выбранных вами имен.

Не рассматривайте это как дополнительный «фактор комфортности». Имена в программных продуктах на 90% определяют удобочитаемость кода. Не жалейте времени на то, чтобы выбрать их осмысленно, и поддерживайте их актуальность. Имена слишком важны, чтобы относиться к ним легкомысленно.

Возьмем следующий код. Что он делает? Когда я представлю вам тот же код с нормально выбранными именами, вы моментально поймете его смысл, но в этом виде он представляет собой мешанину из символов и «волшебных чисел».

```
public int x() {
    int q = 0;
    int z = 0;
    for (int kk = 0; kk < 10; kk++) {
      if (l[z] == 10)
      {
        q += 10 + (l[z + 1] + l[z + 2]);
        z += 1;
      }
      else if (l[z] + l[z + 1] == 10)
      {
        q += 10 + l[z + 2];
        z += 2;
      } else {
        q += l[z] + l[z + 1];
        z += 2;
      }
    }
    return q;
  }
```

А вот как должен был выглядеть этот код. Вообще говоря, этот фрагмент чуть менее полон, чем приведенный выше. И все же вы сразу догадаетесь, что мы пытаемся сделать, и с большой вероятностью сможете написать отсутствующие функции, основываясь на своих предположениях. «Волшебные числа» перестали быть волшебными, а структура алгоритма радует своей очевидностью.

```
public int score() {
  int score = 0;
  int frame = 0;
  for (int frameNumber = 0; frameNumber < 10; frameNumber++) {
    if (isStrike(frame)) {
      score += 10 + nextTwoBallsForStrike(frame);
      frame += 1;
    } else if (isSpare(frame)) {
      score += 10 + nextBallForSpare(frame);
      frame += 2;
    } else {
      score += twoBallsInFrame(frame);
```

```
      frame += 2;
    }
  }
  return score;
}
```

Сила хорошо выбранных имен заключается в том, что они дополняют структуру кода описаниями. На основании этих описаний у читателя формируются определенные предположения по поводу того, что делают другие функции модуля. Взглянув на приведенный код, вы сможете представить себе примерную реализацию `isStrike()`. А при чтении метода `isStrike()` становится очевидно, что он делает «примерно то, что предполагалось»[1].

```
private boolean isStrike(int frame) {
  return rolls[frame] == 10;
}
```

N2: Выбирайте имена на подходящем уровне абстракции

Не используйте имена, передающие информацию о реализации. Имена должны отражать уровень абстракции, на котором работает класс или функция. Сделать это непросто — и снова потому, что люди слишком хорошо справляются со смешением разных уровней абстракции. При каждом просмотре кода вам с большой вероятностью попадется переменная, имя которой выбрано на слишком низком уровне. Воспользуйтесь случаем и измените его. Чтобы ваш код хорошо читался, вы должны серьезно относиться к его непрерывному совершенствованию. Возьмем следующий интерфейс `Modem`:

```
public interface Modem {
  boolean dial(String phoneNumber);
  boolean disconnect();
  boolean send(char c);
  char recv();
  String getConnectedPhoneNumber();
}
```

На первый взгляд все хорошо — имена функций выглядят разумно. В самом деле, во многих приложениях они точно соответствуют выполняемым операциям. А если для установления связи используется не коммутируемое подключение, а какой-то другой механизм? Например, модемы могут связываться на физическом уровне (как кабельные модемы, обеспечивающие доступ к Интернету во многих домах). А может быть, связь устанавливается посредством отправки номера порта коммутатору через интерфейс USB. Разумеется, концепция телефонных номеров в таких случаях относится к неверному уровню абстракции.

[1] См. цитату Уорда Каннингема на с. 34.

Более правильная стратегия выбора имен в таких сценариях может выглядеть так:

```
public interface Modem {
  boolean connect(String connectionLocator);
  boolean disconnect();
  boolean send(char c);
  char recv();
  String getConnectedLocator();
}
```

Теперь имена функций никак не ассоциируются с телефонными номерами. Они могут использоваться как для подключения по телефонной линии, так и для любой другой стратегии подключения.

N3: По возможности используйте стандартную номенклатуру

Имена проще понять, если они основаны на существующих конвенциях или стандартных обозначениях. Например, при использовании паттерна ДЕКОРАТОР можно включить в имена декорирующих классов слово `Decorator`. Например, имя `AutoHangupModemDecorator` может быть присвоено классу, который дополняет класс `Modem` возможностью автоматического разрыва связи в конце сеанса.

Паттерны составляют лишь одну разновидность стандартов. Например, в языке Java функции, преобразующие объекты в строковые представления, часто называются `toString`. Лучше следовать подобным стандартным конвенциям, чем изобретать их заново.

Группы часто разрабатывают собственные стандартные системы имен для конкретного проекта. Эрик Эванс (Eric Evans) называет их *всеобщим языком* проекта[1]. Широко используйте термины этого языка в своем коде. Чем больше вы используете имена, переопределенные специальным смыслом, относящимся к вашему конкретному проекту, тем проще читателю понять, о чем идет речь в вашем коде.

N4: Недвусмысленные имена

Выбирайте имена, которые максимально недвусмысленно передают назначение функции или переменной. Рассмотрим пример из FitNesse:

```
private String doRename() throws Exception
{
  if(refactorReferences)
    renameReferences();
  renamePage();
```

[1] [DDD].

```
  pathToRename.removeNameFromEnd();
  pathToRename.addNameToEnd(newName);
  return PathParser.render(pathToRename);
}
```

Имя функции получилось слишком общим и расплывчатым; оно ничего не говорит о том, что делает функция. Ситуацию усугубляет тот факт, что в функции с именем `doRename` находится функция `renamePage`! Что можно сказать о различиях между этими функциями по их именам? Ничего.

Функцию было бы правильнее назвать `renamePageAndOptionallyAllReferences`. На первый взгляд имя кажется слишком длинным, но функция вызывается только из одной точки модуля, поэтому ее документирующая ценность перевешивает длину.

N5: Используйте длинные имена для длинных областей видимости

Длина имени должна соответствовать длине его области видимости. Переменным с крошечной областью видимости можно присваивать очень короткие имена, но у переменных с большей областью видимости имена должны быть длинными.

Если область видимости переменной составляет всего пять строк, то переменной можно присвоить имя `i` или `j`. Возьмем следующий фрагмент из старой стандартной игры «Bowling»:

```
private void rollMany(int n, int pins)
{
  for (int i=0; i<n; i++)
    g.roll(pins);
}
```

Смысл переменной `i` абсолютно очевиден. Какое-нибудь раздражающее имя вида `rollCount` только затемнило бы смысл этого тривиального кода. С другой стороны, смысл коротких имен переменных и функций рассеивается на длинных дистанциях. Таким образом, чем длиннее область видимости имени, тем более длинным и точным должно быть ее имя.

N6: Избегайте кодирования

Информация о типе или области видимости не должна кодироваться в именах. Префиксы вида `m_` или `f` бессмысленны в современных средах. Кроме того, информация о проекте и/или подсистеме (например, префикс `vis_` для подсистемы визуализации) также отвлекает читателя и является избыточной. Современные среды разработки позволяют получить всю необходимую информацию без уродования имен. Поддерживайте чистоту в своих именах, не загрязняйте их венгерской записью.

N7: Имена должны описывать побочные эффекты

Имена должны описывать все, что делает функция, переменная или класс. Не скрывайте побочные эффекты за именами. Не используйте простые глаголы для описания функции, которая делает что-то помимо этой простой операции. Для примера возьмем следующий код из TestNG:

```
public ObjectOutputStream getOos() throws IOException {
  if (m_oos == null) {
    m_oos = new ObjectOutputStream(m_socket.getOutputStream());
  }
  return m_oos;
}
```

Функция не ограничивается простым получением `m_oos`; она создает объект `m_oos`, если он не был создан ранее. Таким образом, эту функцию было бы правильнее назвать `createOrReturnOos`.

Тесты

T1: Нехватка тестов

Сколько тестов должен включать тестовый пакет? К сожалению, многие программисты руководствуются принципом «Пожалуй, этого хватит». Тестовый пакет должен тестировать все, что может сломаться. Если в системе остались условия, не проверенные тестами, или вычисления, правильность которых не подтверждена, значит, количество тестов недостаточно.

T2: Используйте средства анализа покрытия кода

Средства анализа покрытия сообщают о пробелах в вашей стратегии тестирования. Они упрощают поиск модулей, классов и функций с недостаточно полным тестированием. Во многих IDE используются визуальные обозначения: строки, покрытые тестами, выделяются зеленым цветом, а непокрытые — красным. Это позволяет легко и быстро обнаружить команды `if` или `catch`, тело которых не проверяется тестами.

T3: Не пропускайте тривиальные тесты

Тривиальные тесты пишутся легко, а их информативная ценность превышает затраты времени на их создание.

T4: Отключенный тест как вопрос

Иногда мы не уверены в подробностях поведения системы, потому что неясны сами требования к программе. Вопрос о требованиях можно выразить в виде теста — закомментированного или помеченного аннотацией `@Ignore`. Выбор зависит от того, компилируется или нет код, к которому относится неопределенность.

T5: Тестируйте граничные условия

Особенно тщательно тестируйте граничные условия. Программисты часто правильно реализуют основную часть алгоритма, забывая о граничных ситуациях.

T6: Тщательно тестируйте код рядом с ошибками

Ошибки часто собираются группами. Если вы обнаружили ошибку в функции, особенно тщательно протестируйте эту функцию. Может оказаться, что ошибка была не одна.

T7: Закономерности сбоев часто несут полезную информацию

Иногда анализ закономерностей в сбоях тестовых сценариев помогает выявить причины возникших проблем. Это еще один аргумент в пользу максимальной полноты тестовых сценариев. Всесторонние наборы тестовых сценариев, упорядоченные логичным образом, выявляют закономерности.

Простой пример: вы заметили, что все тесты с входными данными, длина которых превышает пять символов, завершаются неудачей? Или что любой тест, который передает во втором аргументе функции отрицательное число, не проходит? Иногда простая закономерность чередования красного и зеленого в тестовом отчете заставляет нас воскликнуть «Ага!» на пути к правильному решению. Интересный пример такого рода приведен на с. 303.

T8: Закономерности покрытия кода часто несут полезную информацию

Анализ того, какой код выполняется или не выполняется в ходе тестирования, иногда подсказывает причины возможных сбоев в ходе тестирования.

T9: Тесты должны работать быстро

Медленный тест не выполняется за разумное время. Если время поджимает, из тестового пакета первыми будут удалены медленные тесты. Сделайте все необходимое для того, чтобы ваши тесты работали быстро.

Заключение

Приведенный список эвристических правил и «запахов» не претендует на полноту. Я вообще не уверен в том, можно ли составить полный список такого рода. Но вероятно, стопроцентная полнота здесь и не нужна, поскольку список всего лишь дает косвенное представление о системе ценностей.

Именно эта система ценностей является нашей целью — и основной темой книги. Невозможно написать чистый код, действуя по списку правил. Нельзя стать мастером, изучив набор эвристик. Профессионализм и мастерство формируются на основе ценностей, которыми вы руководствуетесь в обучении.

Литература

[Refactoring]: Refactoring: Improving the Design of Existing Code, Martin Fowler et al., Addison-Wesley, 1999.

[PRAG]: The Pragmatic Programmer, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

[GOF]: Design Patterns: Elements of Reusable Object Oriented Software, Gamma et al., Addison-Wesley, 1996.

[Beck97]: Smalltalk Best Practice Patterns, Kent Beck, Prentice Hall, 1997.

[Beck07]: Implementation Patterns, Kent Beck, Addison-Wesley, 2008.

[PPP]: Agile Software Development: Principles, Patterns, and Practices, Robert C. Martin, Prentice Hall, 2002.

[DDD]: Domain Driven Design, Eric Evans, Addison-Wesley, 2003.

A Многопоточность II

Бретт Л. Шухерт

Данное приложение дополняет и углубляет материал главы «Многопоточность», с. 207. Оно написано в виде набора независимых разделов, которые можно читать в произвольном порядке. Чтобы такое чтение было возможно, материал разделов частично перекрывается.

Пример приложения «клиент/сервер»

Представьте простое приложение «клиент/сервер». Сервер работает в режиме ожидания, прослушивая сокет на предмет клиентских подключений. Клиент подключается и отправляет запросы.

Сервер

Ниже приведена упрощенная версия серверного приложения. Полный исходный код примера приводится, начиная со с. 385.

```
ServerSocket serverSocket = new ServerSocket(8009);
while (keepProcessing) {
    try {
        Socket socket = serverSocket.accept();
        process(socket);
    } catch (Exception e) {
        handle(e);
    }
}
```

Приложение ожидает подключения, обрабатывает входящее сообщение, а затем снова ожидает следующего клиентского запроса. Код клиента для подключения к серверу выглядит так:

```
private void connectSendReceive(int i) {
    try {
        Socket socket = new Socket("localhost", PORT);
        MessageUtils.sendMessage(socket, Integer.toString(i));
        MessageUtils.getMessage(socket);
        socket.close();
```

```
    } catch (Exception e) {
        e.printStackTrace();

    }
}
```

Как работает пара «клиент/сервер»? Как формально описать ее производительность? Следующий тест проверяет, что производительность является «приемлемой»:

```
@Test(timeout = 10000)
public void shouldRunInUnder10Seconds() throws Exception {
    Thread[] threads = createThreads();
    startAllThreadsw(threads);
    waitForAllThreadsToFinish(threads);
}
```

Чтобы по возможности упростить пример, я исключил из него подготовительный код (см. «ClientTest.java», с. 387). Тест предполагает, что обработка должна быть завершена за 10 000 миллисекунд.

Перед нами классический пример оценки производительности системы. Система должна завершить обработку серии клиентских запросов за 10 секунд. Если сервер сможет обработать все клиентские запросы за положенное время, то тест пройдет.

А что делать, если тест завершится неудачей? В однопоточной модели практически невозможно как-то ускорить обработку запросов (если не считать реализации цикла опроса событий). Сможет ли многопоточная модель решить проблему? Может, но нам необходимо знать, в какой области расходуется основное время выполнения. Возможны два варианта:

- Ввод/вывод – использование сокета, подключение к базе данных, ожидание подгрузки из виртуальной памяти и т. д.
- Процессор – числовые вычисления, обработка регулярных выражений, уборка мусора и т. д.

В системах время обычно расходуется в обеих областях, но для конкретной операции одна из областей является доминирующей. Если код в основном ориентирован на обработку процессором, то повышение вычислительной мощности способно улучшить производительность и обеспечить прохождение теста. Однако количество процессорных тактов все же ограничено, так что реализация многопоточной модели в процессорно-ориентированных задачах не ускорит их выполнения.

С другой стороны, если значительное время в выполняемом процессе расходуется на операции ввода/вывода, то многопоточная модель способна повысить эффективность работы. Пока одна часть системы ожидает ввода/вывода, другая часть использует время ожидания для выполнения других действий, обеспечивая более эффективное использование процессорного времени.

Реализация многопоточности

Допустим, тест производительности не прошел. Как повысить производительность и обеспечить прохождение теста? Если серверный метод `process` ориентирован на ввод/вывод, одна из возможных реализаций многопоточной модели выглядит так:

```
void process(final Socket socket) {
    if (socket == null)
        return;

    Runnable clientHandler = new Runnable() {
        public void run() {
            try {
                String message = MessageUtils.getMessage(socket);
                MessageUtils.sendMessage(socket, "Processed: " + message);
                closeIgnoringException(socket);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };

    Thread clientConnection = new Thread(clientHandler);
    clientConnection.start();
}
```

Допустим, в результате внесенных изменений тест проходит[1]; задача решена, верно?

Анализ серверного кода

На обновленном сервере тест успешно завершается всего за одну с небольшим секунду. К сожалению, приведенное решение наивно, и оно создает ряд новых проблем.

Сколько потоков может создать наш сервер? Код не устанавливает ограничений, поэтому сервер вполне может столкнуться с ограничениями, установленными виртуальной машиной Java (JVM). Для многих простых систем этого достаточно. А если система обслуживает огромное количество пользователей в общедоступной сети? При одновременном подключении слишком многих пользователей система «заглохнет».

Но давайте ненадолго отложим проблемы с поведением. Чистота и структура представленного кода тоже оставляют желать лучшего. Какие ответственности возложены на серверный код?

- Управление подключением к сокетам.
- Обработка клиентских запросов.

[1] Вы можете убедиться в этом сами, тестируя код до и после внесения изменений. Однопоточный код приведен на с. 385, а многопоточный – на с. 389.

- Политика многопоточности.
- Политика завершения работы сервера.

К сожалению, все эти ответственности реализуются кодом функции `process`. Кроме того, код распространяется на несколько разных уровней абстракции. Следовательно, какой бы компактной ни была функция `process`, ее все равно необходимо разбить на несколько меньших функций.

У серверного кода несколько причин для изменения; следовательно, он нарушает принцип единой ответственности. Чтобы код многопоточной системы оставался чистым, управление потоками должно быть сосредоточено в нескольких хорошо контролируемых местах. Более того, код управления потоками не должен делать ничего другого. Почему? Да хотя бы потому, что отслеживать проблемы многопоточности достаточно сложно и без параллельного отслеживания других проблем, не имеющих ничего общего с многопоточностью.

Если создать отдельный класс для каждой из ответственностей, перечисленных выше (включая управление потоками), то последствия любых последующих изменений стратегии управления потоками затронут меньший объем кода и не будут загрязнять реализацию других обязанностей. Кроме того, такое разбиение упростит тестирование других модулей, так как вам не придется отвлекаться на многопоточные аспекты. Обновленная версия кода:

```
public void run() {
  while (keepProcessing) {
   try {
    ClientConnection clientConnection = connectionManager.awaitClient();
    ClientRequestProcessor requestProcessor
      = new ClientRequestProcessor(clientConnection);
    clientScheduler.schedule(requestProcessor);
    } catch (Exception e) {
      e.printStackTrace();
    }

  }
  connectionManager.shutdown();

}
```

Все аспекты, относящиеся к многопоточности, теперь собраны в объекте `clientScheduler`. Если в приложении возникнут многопоточные проблемы, искать придется только в одном месте:

```
public interface ClientScheduler {
    void schedule(ClientRequestProcessor requestProcessor);
}
```

Текущая политика реализуется легко:

```
public class ThreadPerRequestScheduler implements ClientScheduler {
    public void schedule(final ClientRequestProcessor requestProcessor) {
```

```
        Runnable runnable = new Runnable() {
            public void run() {
                requestProcessor.process();
            }
        };

       Thread thread = new Thread(runnable);
       thread.start();
    }
}
```

Изоляция всего многопоточного кода существенно упрощает внесение любых изменений в политику управления потоками. Например, чтобы перейти на инфраструктуру Java 5 Executor, достаточно написать новый класс и подключить его к существующему коду (листинг A.1).

Листинг A.1. ExecutorClientScheduler.java

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class ExecutorClientScheduler implements ClientScheduler {
    Executor executor;

    public ExecutorClientScheduler(int availableThreads) {
        executor = Executors.newFixedThreadPool(availableThreads);
    }

    public void schedule(final ClientRequestProcessor requestProcessor) {
        Runnable runnable = new Runnable() {
            public void run() {
                requestProcessor.process();
            }
        };
        executor.execute(runnable);
    }
}
```

Заключение

Применение многопоточной модели в этом конкретном примере показывает, как повысить производительность системы, а также демонстрирует методологию проверки изменившейся производительности посредством тестовой инфраструктуры. Сосредоточение всего многопоточного кода в небольшом количестве классов – пример практического применения принципа единой ответственности. В случае многопоточного программирования это особенно важно из-за его нетривиальности.

Возможные пути выполнения

Рассмотрим код `incrementValue` однострочного метода Java, не содержащего циклов или ветвления:

```
public class IdGenerator {
  int lastIdUsed;
  public int incrementValue() {
    return ++lastIdUsed;
  }
}
```

Забудем о возможности целочисленного переполнения. Будем считать, что только один программный поток имеет доступ к единственному экземпляру `IdGenerator`. В этом случае существует единственный путь выполнения с единственным гарантированным результатом:

- Возвращаемое значение равно значению `lastIdUsed`, и оба значения на одну единицу больше значения `lastIdUsed` непосредственно перед вызовом метода.

Что произойдет, если мы используем два программных потока, а метод останется неизменным? Какие возможны результаты, если каждый поток вызовет `incrementValue` по одному разу? Сколько существует возможных путей выполнения? Начнем с результатов (допустим, `lastIdUsed` начинается со значения 93):

- Поток 1 получает значение 94, поток 2 получает значение 95, значение lastIdUsed равно 95.
- Поток 1 получает значение 95, поток 2 получает значение 94, значение lastIdUsed равно 95.
- Поток 1 получает значение 94, поток 2 получает значение 94, значение lastIdUsed равно 94.

Последний результат выглядит удивительно, но он возможен. Чтобы понять, как образуются эти разные результаты, необходимо разобраться в количестве возможных путей выполнения и в том, как они исполняются виртуальной машиной Java.

Количество путей

Чтобы вычислить количество возможных путей выполнения, начнем со сгенерированного байт-кода. Одна строка кода Java (`return ++lastIdUsed;`) преобразуется в восемь инструкций байт-кода. Выполнение этих восьми инструкций двумя потоками может перемежаться подобно тому, как перемежаются карты в тасуемой колоде[1]. Хотя в каждой руке вы держите всего восемь карт, количество всевозможных перетасованных комбинаций чрезвычайно велико.

[1] Это несколько упрощенное объяснение. Впрочем, для целей нашего обсуждения мы воспользуемся этой упрощенной моделью.

В простом случае последовательности из N команд, без циклов и условных переходов, и T потоков, общее количество возможных путей выполнения будет равно

$$\frac{(NT)!}{N!^T}.$$

ВЫЧИСЛЕНИЕ ВОЗМОЖНЫХ ВАРИАНТОВ УПОРЯДОЧЕНИЯ

Из сообщения электронной почты, отправленного Дядюшкой Бобом Бретту:

При N шагов и T потоков общее количество в итоговой последовательности шагов равно T * N. Перед каждым шагом происходит переключение контекста, в ходе которого производится выбор между T потоками. Каждый путь может быть представлен в виде последовательности цифр, обозначающей переключение контекстов. Так, для шагов A и B с потоками 1 и 2 возможны шесть путей: 1122, 1212, 1221, 2112, 2121 и 2211. Если использовать в записи обозначения шагов, мы получаем A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1 и A2B2A1B1. Для трех потоков последовательность вариантов имеет вид 112233, 112323, 113223, 113232, 112233, 121233, 121323, 121332, 123132, 123123, ...

Одно из свойств этих строк заключается в том, что каждый поток должен присутствовать в строке в N экземплярах. Таким образом, строка 111111 невозможна, потому что она содержит шесть экземпляров 1 и нуль экземпляров 2 и 3.

Итак, нам нужно сгенерировать перестановки из N цифр 1, N цифр 2... и N цифр T. Искомое число равно числу перестановок из N * T объектов, то есть (N * T)!, но с удалением всех дубликатов. Таким образом, задача заключается в том, чтобы подсчитать дубликаты и вычесть их количество из (N * T)!.

Сколько дубликатов содержит серия для двух шагов и двух потоков? Каждая строка из четырех цифр состоит из двух 1 и двух 2. Цифры каждой пары можно поменять местами без изменения смысла строки. Мы можем переставить две цифры 1 и/или две цифры 2. Таким образом, каждая строка существует в четырех изоморфных версиях; это означает, что у каждой строки имеются три дубликата. Три варианта из четырех повторяются, то есть только одна перестановка из четырех НЕ ЯВЛЯЕТСЯ дубликатом. 4! * 0.25 = 6. Похоже, наша схема рассуждений работает.

Как вычислить количество дубликатов в общем случае? Для N = 2 и T = 2 можно переставить 1 и/или 2. Для N = 2 и T = 3 можно переставить 1, 2, 3, 1 и 2, 1 и 3 или 2 и 3. Количество вариантов равно количеству перестановок N. Допустим, существует P разных перестановок N. Количество разных вариантов размещения этих перестановок равно P**T.

Таким образом, количество возможных изоморфных версий равно N!**T. Соответственно, количество путей равно (T*N)!/(N!**T). Для исходного случая T = 2, N = 2 мы получаем 6 (24/4).

Для N = 2 и T = 3 количество путей равно 720/8 = 90.

Для N = 3 и T = 3 получается 9!/6^3 = 1680.

В простейшем случае с одной строкой кода Java, эквивалентной восьми инструкциям байт-кода, и двумя программными потоками общее количество возможных путей выполнения равно 12 870. Если переменная `lastIdUsed` будет относиться

к типу long, то каждая операция чтения/записи преобразуется в две инструкции вместо одной, а количество путей выполнения достигает 2,704,156.

Что произойдет, если внести в метод единственное одно изменение?

```
public synchronized void incrementValue() {
    ++lastIdUsed;
}
```

В этом случае количество возможных путей выполнения сократится до 2 для 2 потоков или до N! в общем случае.

Копаем глубже

А как же удивительный результат, когда два потока вызывают метод по одному разу (до добавления synchronized), получая одинаковое число? Как такое возможно? Начнем с начала.

Атомарной операцией называется операция, выполнение которой не может быть прервано. Например, в следующем коде строка 5, где переменной lastid присваивается значение 0, является атомарной операцией, поскольку в соответствии с моделью памяти Java присваивание 32-разрядного значения прерываться не может.

```
01: public class Example {
02:     int lastId;
03:
04:     public void resetId() {
05:         value = 0;
06:     }
07:
08:     public int getNextId() {
09:         ++value;
10:     }
11:}
```

Что произойдет, если изменить тип lastId с int на long? Останется ли строка 5 атомарной? В соответствии со спецификацией JVM – нет. Она может выполняться как атомарная операция на конкретном процессоре, но по спецификации JVM присваивание 64-разрядной величины требует двух 32-разрядных присваивания. Это означает, что между первым и вторым 32-разрядным присваиванием другой поток может вмешаться и изменить одно из значений.

А оператор префиксного увеличения ++ в строке 9? Выполнение этого оператора может быть прервано, поэтому данная операция не является атомарной. Чтобы понять, как это происходит, мы подробно проанализируем байт-код обоих методов.

Прежде чем двигаться дальше, необходимо усвоить ряд важных определений:

Кадр (frame) – для каждого вызова метода создается кадр с адресом возврата, значениями всех передаваемых параметров и локальных переменных, опреде-

ляемых в методе. Это стандартный способ реализации стека вызовов, используемого в современных языках для вызова функций/методов – как обычного, так и рекурсивного.

Локальная переменная – любая переменная, определяемая в области видимости метода. Все нестатические методы содержат как минимум одну переменную `this`, которая представляет текущий объект, то есть объект, получивший последнее сообщение (в текущем потоке), инициировавшее вызов метода.

Стек операндов – многим инструкциям JVM передаются параметры. Их значения размещаются в стеке операндов, реализованном в виде стандартной структуры данных LIFO (Last-In, First-Out, то есть «последним пришел, первым вышел»).

Байт-код, сгенерированный для `resetId()`, выглядит так.

Мнемоника	Описание	Состояние стека операндов после выполнения
ALOAD 0	Загрузка «нулевой» переменной в стек операндов. Что такое «нулевая» переменная? Это this, текущий объект. При вызове метода получатель сообщения, экземпляр Example, сохраняется в массиве локальных переменных кадра, созданного для вызова метода. Текущий объект всегда является первой сохраняемой переменной для каждого метода экземпляра	this
ICONST_0	Занесение константы 0 в стек операндов	this, 0
PUTFIELD lastId	Сохранение верхнего значения из стека (0) в поле объекта, который задается ссылкой, хранящейся на один элемент ниже вершины стека (this)	<пусто>

Эти три инструкции заведомо атомарны. Хотя программный поток, в котором они выполняются, может быть прерван после выполнения любой инструкции, данные инструкции PUTFIELD (константа 0 на вершине стека и ссылка на `this` в следующем элементе, вместе со значением поля `value`) не могут быть изменены другим потоком. Таким образом, при выполнении присваивания в поле `value` будет гарантированно сохранено значение 0. Операция является атомарной. Все операнды относятся к информации, локальной для данного метода, что исключает нежелательное вмешательство со стороны других потоков.

Итак, если эти три инструкции выполняются десятью потоками, существует 4.38679733629e+24 возможных путей выполнения. Так как в данном случае возможен только один результат, различия в порядке выполнения несущественны. Так уж вышло, что одинаковый результат гарантирован в этой ситуации и для `long`. Почему? Потому что все десять потоков присваивают одну и ту же константу. Даже если их выполнение будет чередоваться, результат не изменится.

Но с операцией ++ в методе `getNextId` возникают проблемы. Допустим, в начале метода поле `lastId` содержит значение 42. Байт-код нового метода выглядит так.

Мнемоника	Описание	Состояние стека операндов после выполнения
ALOAD 0	Загрузка this в стек операндов	this
DUP	Копирование вершины стека. Теперь в стеке операндов хранятся две копии this	this, this
GETFIELD lastID	Загрузка значения поля lastId объекта, ссылка на который хранится в вершине стека (this), и занесение загруженного значения в стек	this, 42
ICONST_1	Занесение константы 1 в стек	this, 42, 1
IADD	Целочисленное сложение двух верхних значений в стеке операндов. Результат сложения также сохраняется в стеке операндов	this, 43
DUP_X1	Копирование значения 43 и сохранение копии в стеке перед this	43, this, 43
PUTFIELD value	Сохранение верхнего значения из стека (43) в поле value текущего объекта, который задается ссылкой, хранящейся на один элемент ниже вершины стека (this)	43
IRETURN	Возвращение верхнего (и единственного) элемента стека	<пусто>

Представьте, что первый поток выполняет первые три инструкции (до `GETFIELD` включительно), а потом прерывается. Второй поток получает управление и выполняет весь метод, увеличивая `lastId` на 1; он получает значение 43. Затем первый поток продолжает работу с того места, на котором она была прервана; значение 42 все еще хранится в стеке операндов, потому что поле `lastId` в момент выполнения `GETFIELD` содержало именно это число. Поток увеличивает его на 1, снова получает 43 и сохраняет результат. В итоге первый поток также получает значение 43. В результате одно из двух увеличений теряется, так как первый поток «перекрыл» результат второго потока (после того как второй поток прервал выполнение первого потока).

Проблема решается объявлением метода `getNexId()` с ключевым словом `synchronized`.

Заключение

Чтобы понять, как потоки могут «перебегать дорогу» друг другу, не обязательно разбираться во всех тонкостях байт-кода. Приведенный пример наглядно показывает, что программные потоки могут вмешиваться в работу друг друга, и этого вполне достаточно.

Впрочем, даже этот тривиальный пример убеждает в необходимости хорошего понимания модели памяти, чтобы вы знали, какие операции безопасны, а какие – нет. Скажем, существует распространенное заблуждение по поводу атомарности оператора ++ (в префиксной или постфиксной форме), тогда как этот оператор атомарным не является. Следовательно, вы должны знать:

- где присутствуют общие объекты/значения;
- какой код может создать проблемы многопоточного чтения/обновления;
- как защититься от возможных проблем многопоточности.

Знайте свои библиотеки

Executor Framework

Как демонстрирует пример ExecutorClientScheduler.java на с. 361, представленная в Java 5 библиотека `Executor` предоставляет расширенные средства управления выполнением программ с использованием пулов программных потоков. Библиотека реализована в виде класса в пакете *java.util.concurrent*.

Если вы создаете потоки, не используя пулы, *или* используете ручную реализацию пулов, возможно, вам стоит воспользоваться `Executor`. От этого ваш код станет более чистым, понятным и компактным.

Инфраструктура `Executor` создает пул потоков с автоматическим изменением размера и повторным созданием потоков при необходимости. Также поддерживаются *фьючерсы* – стандартная конструкция многопоточного программирования. Библиотека `Executor` работает как с классами, реализующими интерфейс `Runnable`, так и с классами, реализующими интерфейс `Callable`. Интерфейс `Callable` похож на `Runnable`, но может возвращать результат, а это стандартная потребность в многопоточных решениях.

Фьючерсы удобны в тех ситуациях, когда код должен выполнить несколько независимых операций и дождаться их завершения:

```
public String processRequest(String message) throws Exception {
    Callable<String> makeExternalCall = new Callable<String>() {
        public String call() throws Exception {
            String result = "";
            // Внешний запрос
            return result;
        }
    };
    Future<String> result = executorService.submit(makeExternalCall);
    String partialResult = doSomeLocalProcessing();
    return result.get() + partialResult;
}
```

В этом примере метод запускает на выполнение объект `makeExternalCall`, после чего переходит к выполнению других действий. Последняя строка содержит

вызов `result.get()`, который блокирует выполнение вплоть до завершения фьючерса.

Неблокирующие решения

Виртуальная машина Java 5 пользуется особенностями архитектуры современных процессоров, поддерживающих надежное неблокирующее обновление. Для примера возьмем класс, использующий синхронизацию (а следовательно, блокировку) для реализации потоково-безопасного обновления `value`,

```
public class ObjectWithValue {
    private int value;
    public void synchronized incrementValue() { ++value; }
    public int getValue() { return value; }
}
```

В Java 5 для этой цели появился ряд новых классов. `AtomicBoolean`, `AtomicInteger` и `AtomicReference` – всего лишь три примера; есть и другие. Приведенный выше фрагмент можно переписать без использования блокировки в следующем виде:

```
public class ObjectWithValue {
    private AtomicInteger value = new AtomicInteger(0);

    public void incrementValue() {
        value.incrementAndGet();
    }

    public int getValue() {
        return value.get();
    }
}
```

Хотя эта реализация использует объект вместо примитива и отправляет сообщения (например, `incrementAndGet()`) вместо ++, по своей производительности этот класс почти всегда превосходит предыдущую версию. Иногда приращение скорости незначительно, но ситуации, в которых он бы работал медленнее, практически не встречаются.

Как такое возможно? Современные процессоры поддерживают операцию, которая обычно называется *CAS* (Compare and Swap). Эта операция является аналогом оптимистичной блокировки из теории баз данных, тогда как синхронизированная версия является аналогом пессимистичной блокировки.

Ключевое слово `synchronized` всегда устанавливает блокировку, даже если второй поток не пытается обновлять то же значение. Хотя производительность встроенных блокировок улучшается от версии к версии, они по-прежнему обходятся недешево. Неблокирующая версия изначально предполагает, что ситуация с обновлением одного значения множественными потоками обычно возникает недостаточно часто для возникновения проблем. Вместо этого она эффективно обнаруживает возникновение таких ситуаций и продолжает повторные попытки до тех пор,

пока обновление не пройдет успешно. Обнаружение конфликта почти всегда обходится дешевле установления блокировки, даже в ситуациях с умеренной и высокой конкуренцией.

Как VM решает эту задачу? CAS является атомарной операцией. На логическом уровне CAS выглядит примерно так:

```
int variableBeingSet;

void simulateNonBlockingSet(int newValue) {
    int currentValue;
    do {
        currentValue = variableBeingSet
    } while(currentValue != compareAndSwap(currentValue, newValue));
}

int synchronized compareAndSwap(int currentValue, int newValue) {
    if(variableBeingSet == currentValue) {
        variableBeingSet = newValue;
        return currentValue;
    }
    return variableBeingSet;
}
```

Когда метод пытается обновить общую переменную, операция CAS проверяет, что изменяемая переменная все еще имеет последнее известное значение. Если условие соблюдается, то переменная изменяется. Если нет, то обновление не выполняется, потому что другой поток успел ему «помешать». Метод, пытавшийся выполнить обновление (с использованием операции CAS), видит, что изменение не состоялось, и делает повторную попытку.

Потоково-небезопасные классы

Некоторые классы в принципе не обладают потоковой безопасностью. Несколько примеров:

- `SimpleDateFormat`
- Подключения к базам данных.
- Контейнеры из *java.util*.
- Сервлеты.

Некоторые классы коллекций содержат отдельные потоково-безопасные методы. Однако любая операция, связанная с вызовом более одного метода, потоково-безопасной не является. Например, если вы не хотите заменять уже существующий элемент `HashTable`, можно было бы написать следующий код:

```
if(!hashTable.containsKey(someKey)) {
    hashTable.put(someKey, new SomeValue());
}
```

По отдельности каждый метод потоково-безопасен, однако другой программный поток может добавить значение между вызовами `containsKey` и `put`. У проблемы есть несколько решений:

- Установите блокировку `HashTable` и проследите за тем, чтобы остальные пользователи `HashTable` делали то же самое (клиентская блокировка):

```
synchronized(map) {
if(!map.conainsKey(key))
    map.put(key,value);
```

- Инкапсулируйте `HashTable` в собственном объекте и используйте другой API (серверная блокировка с применением паттерна АДАПТЕР):

```
public class WrappedHashtable<K, V> {
    private Map<K, V> map = new Hashtable<K, V>();

    public synchronized void putIfAbsent(K key, V value) {
        if (map.containsKey(key))
            map.put(key, value);
    }
}
```

- Используйте потоково-безопасные коллекции:

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer,
String>();
map.putIfAbsent(key, value);
```

Для выполнения подобных операций в коллекциях пакета *java.util.concurrent* предусмотрены такие методы, как `putIfAbsent()`.

Зависимости между методами могут нарушить работу многопоточного кода

Тривиальный пример введения зависимостей между методами:

```
public class IntegerIterator implements Iterator<Integer>
    private Integer nextValue = 0;

    public synchronized boolean hasNext() {
        return nextValue < 100000;
    }
    public synchronized Integer next() {
        if (nextValue == 100000)
            throw new IteratorPastEndException();
        return nextValue++;
    }
    public synchronized Integer getNextValue() {
        return nextValue;
    }
}
```

Код, использующий IntegerIterator:

```
IntegerIterator iterator = new IntegerIterator();
while(iterator.hasNext()) {
    int nextValue = iterator.next();
    // Действия с nextValue
}
```

Если этот код выполняется одним потоком, проблем не будет. Но что произойдет, если два потока попытаются одновременно использовать общий экземпляр IngeterIterator в предположении, что каждый поток будет обрабатывать полученные значения, но каждый элемент списка обрабатывается только один раз? В большинстве случаев ничего плохого не произойдет; потоки будут совместно обращаться к списку, обрабатывая элементы, полученные от итератора, и завершат работу при завершении перебора. Но существует небольшая вероятность того, что в конце итерации два потока помешают работе друг друга, один поток выйдет за конечную позицию итератора, и произойдет исключение.

Проблема заключается в следующем: поток 1 проверяет наличие следующего элемента методом hasNext(), который возвращает true. Поток 1 вытесняется потоком 2; последний выдает тот же запрос, и получает тот же ответ true. Поток 2 вызывает метод next(), который возвращает значение, но с побочным эффектом: после него вызов hasNext() возвращает false. Поток 1 продолжает работу. Полагая, что hasNext() до сих пор возвращает true, он вызывает next(). Хотя каждый из отдельных методов синхронизирован, клиент использовал *два* метода.

Проблемы такого рода очень часто встречаются в многопоточном коде. В нашей конкретной ситуации проблема особенно нетривиальна, потому что она приводит к сбою только при завершающей итерации. Если передача управления между потоками произойдет в строго определенной последовательности, то один из потоков сможет выйти за конечную позицию итератора. Подобные ошибки часто проявляются уже после того, как система пойдет в эксплуатацию, и обнаружить их весьма нелегко.

У вас три варианта:

- Перенести сбои.
- Решить проблему, внося изменения на стороне клиента (клиентская блокировка).
- Решить проблему, внося изменения на стороне сервера, что приводит к дополнительному изменению клиента (серверная блокировка).

Перенесение сбоев

Иногда все удается устроить так, что сбой не приносит вреда. Например, клиент может перехватить исключение и выполнить необходимые действия для восстановления. Откровенно говоря, такое решение выглядит неуклюже. Оно напоминает полуночные перезагрузки, исправляющие последствия утечки памяти.

Клиентская блокировка

Чтобы класс IntegerIterator корректно работал в многопоточных условиях, измените приведенного выше клиента (а также всех остальных клиентов) следующим образом:

```
IntegerIterator iterator = new IntegerIterator();

    while (true) {
      int nextValue;
      synchronized (iterator) {
        if (!iterator.hasNext())
          break;
        nextValue = iterator.next();
      }
      doSometingWith(nextValue);
    }
```

Каждый клиент устанавливает блокировку при помощи ключевого слова synchronized. Дублирование нарушает принцип DRY, но оно может оказаться необходимым, если в коде используются инструменты сторонних разработчиков, не обладающие потоковой безопасностью.

Данная стратегия сопряжена с определенным риском. Все программисты, использующие сервер, должны помнить об установлении блокировки перед использованием и ее снятии после использования. Много (очень много!) лет назад я работал над системой, в которой использовалась клиентская блокировка общего ресурса. Ресурс использовался в сотне разных мест по всей кодовой базе. Один несчастный программист забыл установить блокировку в одном из таких мест.

Это была многотерминальная система с разделением времени, на которой выполнялись бухгалтерские программы профсоюза транспортных перевозок Local 705. Компьютер находился в зале с фальшполом и кондиционером за 50 миль к северу от управления Local 705. В управлении десятки операторов вводили данные на терминалах. Терминалы были подключены к компьютеру по выделенным телефонным линиям с полудуплексными модемами на скорости 600 бит/с (это было очень, очень давно).

Примерно раз в день один из терминалов «зависал». Никакие закономерности в сбоях не прослеживались. Зависания не были привязаны ни к конкретным терминалам, ни к конкретному времени. Все выглядело так, словно время зависания и терминал выбирались броском кубика. Иногда целые дни проходили без зависаний.

Поначалу проблема решалась только перезагрузкой, но перезагрузки было трудно координировать. Нам приходилось звонить в управление и просить всех операторов завершить текущую работу на всех терминалах. После этого мы могли отключить питание и перезагрузить систему. Если кто-то выполнял важную работу, занимавшую час или два, зависший терминал попросту простаивал.

Через несколько недель отладки мы обнаружили, что причиной зависания была десинхронизация между счетчиком кольцевого буфера и его указателем. Буфер управлял выводом на терминал. Значение указателя говорило о том, что буфер был пуст, а счетчик утверждал, что буфер полон. Так как буфер был пуст, выводить было нечего; но так как одновременно он был полон, в экранный буфер нельзя было ничего добавить для вывода на экран.

Итак, мы знали, почему зависали терминалы, но было неясно, из-за чего возникает десинхронизация кольцевого буфера. Поэтому мы реализовали обходное решение. Программный код мог прочитать состояние тумблеров на передней панели компьютера (это было очень, очень, очень давно). Мы написали небольшую функцию, которая обнаруживала переключение одного из тумблеров и искала кольцевой буфер, одновременно пустой и заполненный. Обнаружив такой буфер, функция сбрасывала его в пустое состояние. *Voila!* Зависший терминал снова начинал выводить информацию.

Теперь нам не приходилось перезагружать систему при зависании терминала. Когда нам звонили из управления и сообщали о зависании, мы шли в машинный зал и переключали тумблер.

Конечно, иногда управление работало по выходным, а мы – нет. Поэтому в планировщик была включена функция, которая раз в минуту проверяла состояние всех кольцевых буферов и сбрасывала те из них, которые были одновременно пустыми и заполненными. Зависший терминал начинал работать даже до того, как операторы успевали подойти к телефону.

Прошло несколько недель кропотливого просеивания монолитного ассемблерного кода, прежде чем была обнаружена причина. Мы занялись вычислениями и определили, что частота зависаний статистически соответствует одному незащищенному использованию кольцевого буфера. Оставалось только найти это одно использование. К сожалению, все это было очень давно. В те времена у нас не было функций поиска, перекрестных ссылок или других средств автоматизации. Нам просто приходилось просматривать листинги.

Тогда, холодной зимой 1971 года в Чикаго, я узнал важный урок. Клиентская блокировка — полный отстой.

Серверная блокировка

Дублирование можно устранить внесением следующих изменений в `Integer-Iterator`:

```
public class IntegerIteratorServerLocked {
    private Integer nextValue = 0;
    public synchronized Integer getNextOrNull() {
        if (nextValue < 100000)
            return nextValue++;
        else
            return null;
    }
}
```

В клиентском коде также вносятся изменения:

```
while (true) {
    Integer nextValue = iterator.getNextOrNull();
    if (next == null)
        break;
    // Действия с nextValue
}
```

В этом случае мы изменяем API своего класса, чтобы он обладал многопоточной поддержкой[1]. Вместо проверки `hasNext()` клиент должен выполнить проверку `null`.

В общем случае серверная блокировка предпочтительна по следующим причинам:

- Она сокращает дублирование кода – клиентская блокировка заставляет каждого клиента устанавливать соответствующую блокировку сервера. Если код блокировки размещается на сервере, клиенты могут использовать объект, не беспокоясь о написании дополнительного кода блокировки.
- Она обеспечивает более высокую производительность – в случае однопоточного развертывания потоково-безопасный сервер можно заменить потоково-небезопасным, устраняя все дополнительные затраты.
- Она снижает вероятность ошибок – в случае клиентской блокировки достаточно всего одному программисту забыть установить блокировку, и работа системы будет нарушена.
- Она определяет единую политику использования – политика сосредоточена в одном месте (на сервере), а не во множестве разных мест (то есть у каждого клиента).

Она сокращает область видимости общих переменных — клиент не знает ни о переменных, ни о том, как они блокируются. Все подробности скрыты на стороне сервера. Если что-то сломается, то количество мест, в которых следует искать причину, сокращается.

Что делать, если серверный код вам неподконтролен?

- Используйте паттерн АДАПТЕР, чтобы изменить API и добавить блокировку:

```
public class ThreadSafeIntegerIterator {
    private IntegerIterator iterator = new IntegerIterator();
    public synchronized Integer getNextOrNull() {
        if(iterator.hasNext())
            return iterator.next();
        return null;
    }
}
```

- ИЛИ еще лучше – используйте потоково-безопасные коллекции с расширенными интерфейсами.

[1] На самом деле интерфейс Iterator в принципе не обладает потоковой безопасностью. Он не проектировался с расчетом на многопоточное использование, так что этот факт не вызывает удивления.

Повышение производительности

Допустим, вы хотите выйти в сеть и прочитать содержимое группы страниц по списку URL-адресов. По мере чтения страницы обрабатываются для накопления некоторой статистики. После чтения всех страниц выводится сводный отчет.

Следующий класс возвращает содержимое одной страницы по URL-адресу.

```
public class PageReader {
  //...
  public String getPageFor(String url) {
    HttpMethod method = new GetMethod(url);

    try {
      httpClient.executeMethod(method);
      String response = method.getResponseBodyAsString();
      return response;
    } catch (Exception e) {
      handle(e);
    } finally {
      method.releaseConnection();
    }
  }
}
```

Следующий класс – итератор, предоставляющий содержимое страниц на основании итератора URL-адресов:

```
public class PageIterator {
  private PageReader reader;
  private URLIterator urls;

  public PageIterator(PageReader reader, URLIterator urls) {
    this.urls = urls;
    this.reader = reader;
  }

  public synchronized String getNextPageOrNull() {
    if (urls.hasNext())
      getPageFor(urls.next());
    else
      return null;
  }

  public String getPageFor(String url) {
    return reader.getPageFor(url);
  }
}
```

Экземпляр `PageIterator` может совместно использоваться разными потоками, каждый из которых использует собственный экземпляр `PageReader` для чтения и обработки страниц, полученных от итератора.

Обратите внимание: блок `synchronized` очень мал. Он содержит только критическую секцию, расположенную глубоко внутри `PageIterator`. Старайтесь синхронизировать как можно меньший объем кода.

Вычисление производительности в однопоточной модели

Выполним некоторые простые вычисления. Для наглядности возьмем следующие показатели:

- Время ввода/вывода для получения страницы (в среднем): 1 секунда.
- Время обработки страницы (в среднем): 0,5 секунды.
- Во время операций ввода/вывода процессор загружен на 0%, а во время обработки – на 100%.

При обработке N страниц в однопоточной модели общее время выполнения составляет 1,5 секунды * N. На рис. А.1 изображен график обработки 13 страниц примерно за 19,5 секунды.

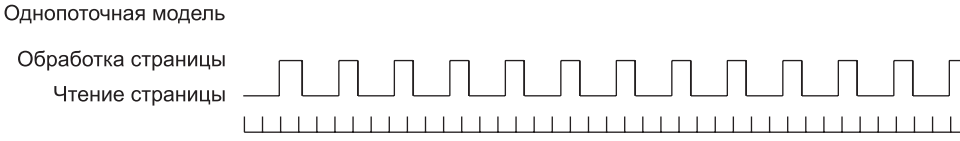


Рис. А.1. Обработка страниц в однопоточной модели

Вычисление производительности в многопоточной модели

Если страницы могут загружаться в произвольном порядке и обрабатываться независимо друг от друга, то для повышения производительности можно воспользоваться многопоточной моделью. Что произойдет, если обработка будет производиться тремя потоками? Сколько страниц удастся обработать за то же время?

Как видно из рис. А.2, многопоточное решение позволяет совмещать процессорно-ориентированную обработку страниц с операциями чтения страниц, ориентированными на ввод/вывод. В идеальном случае это обеспечивало бы полную загрузку процессора: каждое чтение страницы продолжительностью в одну секунду перекрывается с обработкой двух страниц. Таким образом, многопоточная модель обрабатывает две страницы в секунду, что втрое превышает производительность однопоточной модели.

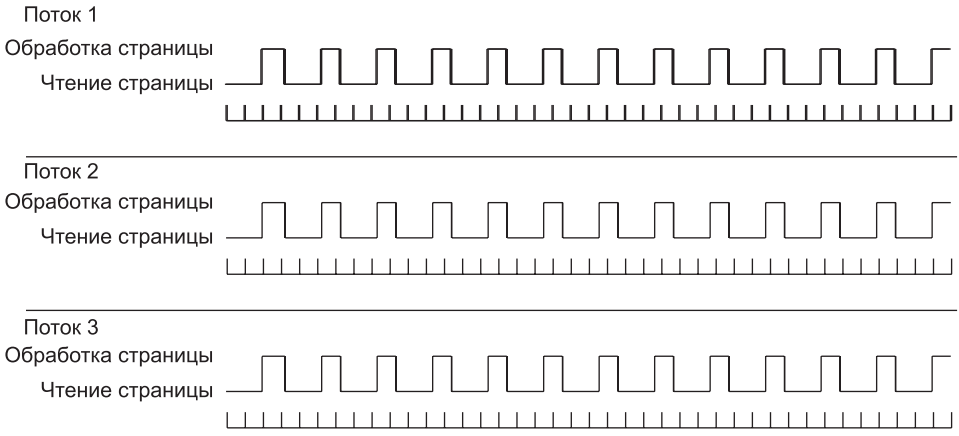


Рис. А.2. Обработка тремя параллельными потоками

Взаимная блокировка

Допустим, у нас имеется веб-приложение с двумя общими пулами ресурсов конечного размера:

- Пул подключений к базе данных для локальной обработки в памяти процесса.
- Пул подключений MQ к главному хранилищу.

В работе приложения используются две операции, создание и обновление:

- Создание – получение подключений к главному хранилищу и базе данных. Взаимодействие с главным хранилищем и локальное сохранение данных в базе данных процесса.
- Обновление – получение подключений к базе данных, а затем к главному хранилищу. Чтение данных из базы данных процесса и их последующая передача в главное хранилище.

Что произойдет, если количество пользователей превышает размеры пулов? Допустим, каждый пул содержит десять подключений.

- Десять пользователей пытаются использовать операцию создания. Они захватывают все десять подключений к базе данных. Выполнение каждого потока прерывается после захвата подключения к базе данных, но до захвата подключения к главному хранилищу.
- Десять пользователей пытаются использовать операцию обновления. Они захватывают все десять подключений к главному хранилищу. Выполнение каждого потока прерывается после захвата подключения к главному хранилищу, но до захвата подключения к базе данных.

- Десять потоков, выполняющих операцию создания, ожидают подключений к главному хранилищу, а десять потоков, выполняющих операцию обновления, ожидают подключений к базе данных.
- Возникает взаимная блокировка. Продолжение работы системы невозможно.

На первый взгляд такая ситуация выглядит маловероятной, но кому нужна система, которая гарантированно зависает каждую неделю? Кому захочется отлаживать систему с такими трудновоспроизводимыми симптомами? Когда такие проблемы возникают в эксплуатируемой системе, на их решение уходят целые недели.

Типичное «решение» основано на включении отладочных команд для получения дополнительной информации о происходящем. Конечно, отладочные команды достаточно сильно изменяют код, взаимная блокировка возникает в другой ситуации, и на повторение ошибки могут потребоваться целые месяцы[1].

Чтобы действительно решить проблему взаимной блокировки, необходимо понять, из-за чего она возникает. Для возникновения взаимной блокировки необходимы четыре условия:

- Взаимное исключение.
- Блокировка с ожиданием.
- Отсутствие вытеснения.
- Циклическое ожидание.

Взаимное исключение

Взаимное исключение возникает в том случае, когда несколько потоков должны использовать одни и те же ресурсы, и эти ресурсы:

- не могут использоваться несколькими потоками одновременно;
- существуют в ограниченном количестве.

Типичный пример ресурсов такого рода – подключения к базам данных, открытые для записи файлы, блокировки записей, семафоры.

Блокировка с ожиданием

Когда один поток захватывает ресурс, он не освобождает его до тех пор, пока не захватит все остальные необходимые ресурсы и не завершит свою работу.

Отсутствие вытеснения

Один поток не может отнимать ресурсы у другого потока. Если поток захватил ресурс, то другой поток сможет получить захваченный ресурс только в одном случае: если первый поток его освободит.

[1] Кто-то добавляет отладочный вывод, и проблема «исчезает». Отладочный код «решил» проблему, поэтому он остается в системе.

Циклическое ожидание

Допустим, имеются два потока T1 и T2 и два ресурса R1 и R2. Поток T1 захватил R1, поток T2 захватил R2. Потоку T1 также необходим ресурс R2, а потоку T2 также необходим ресурс R1. Ситуация выглядит так, как показано на рис. А.3.

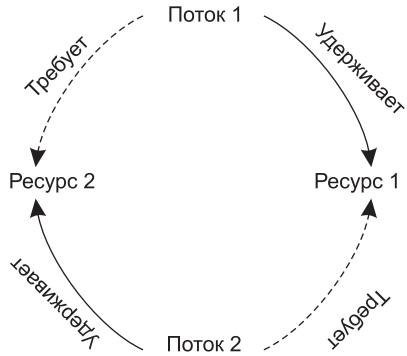


Рис. А.3. Циклическое ожидание

Взаимная блокировка возможна только при соблюдении всех четырех условий. Стоит хотя бы одному из них нарушиться, и взаимная блокировка исчезнет.

Нарушение взаимного исключения

Одна из стратегий предотвращения взаимной блокировки основана на предотвращении состояния взаимного исключения.

- Использование ресурсов, поддерживающих многопоточный доступ (например, `AtomicInteger`).
- Увеличение количества ресурсов, чтобы оно достигло или превосходило количество конкурирующих потоков.
- Проверка наличия всех свободных ресурсов перед попытками захвата.

К сожалению, большинство ресурсов существует в ограниченном количестве и не поддерживает многопоточное использование. Кроме того, второй ресурс нередко определяется только по результатам обработки первого ресурса. Но не огорчайтесь – остаются еще три условия.

Нарушение блокировки с ожиданием

Вы также можете нарушить взаимную блокировку, если откажетесь ждать. Проверяйте каждый ресурс, прежде чем захватывать его; если какой-либо ресурс занят, освободите все захваченные ресурсы и начните все заново.

При таком подходе возможны следующие потенциальные проблемы:

- *Истощение* – один поток стабильно не может захватить все необходимые ему ресурсы (уникальная комбинация, в которой все ресурсы одновременно оказываются свободными крайне редко).

- *Обратимая блокировка* – несколько потоков «входят в клинч»: все они захватывают один ресурс, затем освобождают один ресурс... снова и снова. Такая ситуация особенно вероятна при использовании тривиальных алгоритмов планирования процессорного времени (встроенные системы; простейшие, написанные вручную алгоритмы балансировки потоков).

Обе ситуации могут привести к снижению производительности. Первая ситуация оборачивается недостаточным, а вторая – завышенным и малоэффективным использованием процессора.

Какой бы неэффективной ни казалась эта стратегия, она все же лучше, чем ничего. По крайней мере, вы всегда можете реализовать ее, если все остальные меры не дают результата.

Нарушение отсутствия вытеснения

Другая стратегия предотвращения взаимных блокировок основана на том, чтобы потоки могли отнимать ресурсы у других потоков. Обычно вытеснение реализуется на основе простого механизма запросов. Когда поток обнаруживает, что ресурс занят, он обращается к владельцу с запросом на его освобождение. Если владелец также ожидает другого ресурса, он освобождает все удерживаемые ресурсы и начинает захватывать их заново.

Данное решение сходно с предыдущим, но имеет дополнительное преимущество: поток может ожидать освобождения ресурса. Это снижает количество освобождений и повторных захватов. Но учтите, что правильно реализовать управление запросами весьма непросто.

Нарушение циклического ожидания

Это самая распространенная стратегия предотвращения взаимных блокировок. В большинстве систем она не требует ничего, кроме системы простых правил, соблюдаемых всеми сторонами.

Вспомните приведенный ранее пример с потоком 1, которому необходимы ресурс 1 и ресурс 2, и потоком 2, которому необходимы ресурс 2 и ресурс 1. Заставьте поток 1 и поток 2 выделять ресурсы в постоянном порядке, при котором циклическое ожидание станет невозможным.

В более общем виде, если все потоки согласуют единый глобальный порядок и будут захватывать ресурсы только в указанном порядке, то взаимная блокировка станет невозможной. Эта стратегия, как и все остальные, может создавать проблемы:

- Порядок захвата может не соответствовать порядку использования; таким образом, ресурс, захваченный в начале, может не использоваться до самого конца. В результате ресурсы остаются заблокированными дольше необходимого.

- Соблюдение фиксированного порядка захвата ресурсов возможно не всегда. Если идентификатор второго ресурса определяется на основании операций, выполняемых с первым ресурсом, то эта стратегия становится невозможной.

Итак, существует много разных способов предотвращения взаимных блокировок. Одни приводят к истощению потоков, другие – к интенсивному использованию процессора и ухудшению времени отклика. Бесплатный сыр бывает только в мышеловке!

Изоляция потокового кода вашего решения упрощает настройку и эксперименты; к тому же это действенный способ сбора информации, необходимой для определения оптимальных стратегий.

Тестирование многопоточного кода

Как написать тест, демонстрирующий некорректность многопоточного кода?

```
01: public class ClassWithThreadingProblem {
02:     int nextId;
03:
04:     public int takeNextId() {
05:         return nextId++;
06:     }
07:}
```

Тест, доказывающий некорректность, может выглядеть так:

- Запомнить текущее значение `nextId`.
- Создать два потока, каждый из которых вызывает `takeNextId()` по одному разу.
- Убедиться в том, что значение `nextId` на 2 больше исходного.
- Выполнять тест до тех пор, пока в ходе очередного теста `nextId` не увеличится только на 1 вместо 2.

Код такого теста представлен в листинге А.2.

Листинг А.2. ClassWithThreadingProblemTest.java

```
01: package example;
02:
03: import static org.junit.Assert.fail;
04:
05: import org.junit.Test;
06:
07: public class ClassWithThreadingProblemTest {
08:     @Test
09:     public void twoThreadsShouldFailEventually() throws Exception {
10:         final ClassWithThreadingProblem classWithThreadingProblem
                = new ClassWithThreadingProblem();
11:
12:         Runnable runnable = new Runnable() {
```

продолжение ⇩

Листинг А.2 (продолжение)

```
13:             public void run() {
14:                 classWithThreadingProblem.takeNextId();
15:             }
16:         };
17:
18:         for (int i = 0; i < 50000; ++i) {
19:             int startingId = classWithThreadingProblem.lastId;
20:             int expectedResult = 2 + startingId;
21:
22:             Thread t1 = new Thread(runnable);
23:             Thread t2 = new Thread(runnable);
24:             t1.start();
25:             t2.start();
26:             t1.join();
27:             t2.join();
28:
29:             int endingId = classWithThreadingProblem.lastId;
30:
31:             if (endingId != expectedResult)
32:                 return;
33:         }
34:
35:         fail("Should have exposed a threading issue but it did not.");
36:     }
37: }
```

Строка	Описание
10	Создание экземпляра ClassWithThreadingProblem. Обратите внимание на необходимость использования ключевого слова final, так как ниже объект используется в анонимном внутреннем классе
12–16	Создание анонимного внутреннего класса, использующего экземпляр ClassWithThreadingProblem
18	Код выполняется количество раз, достаточное для демонстрации его некорректности, но так, чтобы он не выполнялся «слишком долго». Необходимо выдержать баланс между двумя целями; сбои должны выявляться за разумное время. Подобрать нужное число непросто, хотя, как мы вскоре увидим, его можно заметно сократить
19	Сохранение начального значения. Мы пытаемся доказать, что код ClassWithThreadingProblem некорректен. Если тест проходит, то он доказывает, что код некорректен. Если тест не проходит, то он не доказывает ничего
20	Итоговое значение должно быть на два больше текущего
22–23	Создание двух потоков, использующих объект, который был создан в строках 12–16. Два потока, пытающихся использовать один экземпляр ClassWithThreadingProblem, могут помешать друг другу; эту ситуацию мы и пытаемся воспроизвести.
24–25	Запуск двух потоков

Строка	Описание
26–27	Ожидание завершения обоих потоков с последующей проверкой результатов
29	Сохранение итогового значения
31–32	Отличается ли значение endingId от ожидаемого? Если отличается, вернуть признак завершения теста – доказано, что код работает некорректно. Если нет, попробовать еще раз
35	Если управление передано в эту точку, нашим тестам не удалось доказать некорректность кода за «разумное» время. Либо код работает корректно, либо количество итераций было недостаточным для возникновения сбойной комбинации

Бесспорно, этот тест создает условия для выявления проблем многопоточного обновления. Но проблема встречается настолько редко, что в подавляющем большинстве случаев тестирование ее попросту не выявит. В самом деле, для сколько-нибудь статистически значимого выявления проблемы количество итераций должно превышать миллион. Несмотря на это, за десять выполнений цикла из 1 000 000 итераций проблема была обнаружена всего один раз. Это означает, что для надежного выявления сбоев количество итераций должно составлять около 100 миллионов. Как долго вы готовы ждать?

Даже если тест будет надежно выявлять сбои на одном компьютере, вероятно, его придется заново настраивать с другими параметрами для выявления сбоев на другом компьютере, операционной системе или версии JVM.

А ведь мы взяли очень простую задачу. Если нам не удается легко продемонстрировать некорректность кода в тривиальной ситуации, как обнаружить по-настоящему сложную проблему?

Как ускорить выявление этого простейшего сбоя? И что еще важнее, как написать тесты, демонстрирующие сбои в более сложном коде? Как узнать, что код некорректен, если мы даже не знаем, где искать? Несколько возможных идей.

Тестирование методом Монте-Карло. Сделайте тесты достаточно гибкими, чтобы вы могли легко вносить изменения в их конфигурацию. Повторяйте тесты снова и снова (скажем, на тестовом сервере) со случайным изменением параметров. Если в ходе тестирования будет обнаружена ошибка, значит, код некорректен. Начните писать эти тесты на ранней стадии, чтобы как можно ранее начать их выполнение на сервере непрерывной интеграции. Не забудьте сохранить набор условий, при котором был выявлен сбой.

- Выполняйте тесты на каждой целевой платформе разработки. Многократно. Непрерывно. Чем продолжительнее тесты работают без сбоев, тем выше вероятность, что:
  - код продукта корректен, либо
  - тестирования недостаточно для выявления проблем.

Запускайте тесты на машинах с разной нагрузкой. Если вы сможете имитировать нагрузку, близкую к среде реальной эксплуатации, сделайте это.

Но даже после выполнения всех этих действий вероятность обнаружения многопоточных проблем в вашем коде оставляет желать лучшего. Самыми коваными оказываются проблемы, возникающие при определенных комбинациях условий, встречающихся один раз на миллиард. Такие проблемы – настоящий бич сложных систем.

Средства тестирования многопоточного кода

Компания IBM создала программу ConTest[1], которая особым образом готовит классы для повышения вероятности сбоев в потоково-небезопасном коде.

Мы не связаны ни с IBM, ни с группой разработки ConTest. Об этой программе нам рассказал один из коллег. Оказалось, что всего несколько минут использования ConTest радикально повышают вероятность выявления многопоточных сбоев.

Тестирование с использованием ConTest проходит по следующей схеме:

- Напишите тесты и код продукта. Проследите за тем, чтобы тесты были специально спроектированы для имитации обращений от многих пользователей при переменной нагрузке, как упоминалось ранее.
- Проведите инструментовку кода тестов и продукта при помощи ConTest.
- Выполните тесты.

Если вы помните, ранее сбой выявлялся примерно один раз за десять миллионов запусков. После инструментовки кода в ConTest сбои стали выявляться один раз за *тридцать* запусков. Таким образом, сбои в адаптированных классах стали выявляться намного быстрее и надежнее.

Заключение

В этой главе мы предприняли очень краткое путешествие по огромной, ненадежной территории многопоточного программирования. Наше знакомство с этой темой нельзя назвать даже поверхностным. Основное внимание уделялось методам поддержания чистоты многопоточного кода, но если вы собираетесь писать многопоточные системы, вам придется еще многому научиться. Мы рекомендуем начать с замечательной книги Дуга Ли «Concurrent Programming in Java: Design Principles and Patterns» [Lea99, p. 191].

В этой главе рассматривались опасности многопоточного обновления, а также методы чистой синхронизации и блокировки, которые могут их предотвратить. Мы

[1] http://www.haifa.ibm.com/projects/veri cation/contest/index.html

обсудили, как потоки могут повысить производительность систем с интенсивным вводом/выводом, а также конкретные приемы повышения производительности. Также была рассмотрена взаимная блокировка и чистые способы ее предотвращения. Приложение завершается описанием стратегий выявления многопоточных проблем посредством инструментовки кода.

Полные примеры кода

Однопоточная реализация архитектуры «клиент/сервер»

Листинг A.3. Server.java

```
package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
    ServerSocket serverSocket;
    volatile boolean keepProcessing = true;

    public Server(int port, int millisecondsTimeout) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(millisecondsTimeout);
    }

    public void run() {
        System.out.printf("Server Starting\n");

        while (keepProcessing) {
            try {
                System.out.printf("accepting client\n");
                Socket socket = serverSocket.accept();
                System.out.printf("got client\n");
                process(socket);
            } catch (Exception e) {
                handle(e);
            }
        }
    }
```

продолжение ⇩

Листинг А.3 (продолжение)

```
    private void handle(Exception e) {
        if (!(e instanceof SocketException)) {
            e.printStackTrace();
        }
    }

    public void stopProcessing() {
        keepProcessing = false;
        closeIgnoringException(serverSocket);
    }

    void process(Socket socket) {
        if (socket == null)
            return;

        try {
            System.out.printf("Server: getting message\n");
            String message = MessageUtils.getMessage(socket);
            System.out.printf("Server: got message: %s\n", message);
            Thread.sleep(1000);
            System.out.printf("Server: sending reply: %s\n", message);
            MessageUtils.sendMessage(socket, "Processed: " + message);
            System.out.printf("Server: sent\n");
            closeIgnoringException(socket);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    private void closeIgnoringException(Socket socket) {
        if (socket != null)
            try {
                socket.close();
            } catch (IOException ignore) {
            }
    }

    private void closeIgnoringException(ServerSocket serverSocket) {
        if (serverSocket != null)
            try {
                serverSocket.close();
            } catch (IOException ignore) {
            }
    }
}
```

Листинг A.4. ClientTest.java

```
package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
    ServerSocket serverSocket;
    volatile boolean keepProcessing = true;

    public Server(int port, int millisecondsTimeout) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(millisecondsTimeout);
    }

    public void run() {
        System.out.printf("Server Starting\n");

        while (keepProcessing) {
            try {
                System.out.printf("accepting client\n");
                Socket socket = serverSocket.accept();
                System.out.printf("got client\n");
                process(socket);
            } catch (Exception e) {
                handle(e);
            }
        }
    }

    private void handle(Exception e) {
        if (!(e instanceof SocketException)) {
            e.printStackTrace();
        }
    }

    public void stopProcessing() {
        keepProcessing = false;
        closeIgnoringException(serverSocket);
    }

    void process(Socket socket) {
        if (socket == null)
            return;
```

продолжение ⇩

Листинг А.4 (продолжение)

```
        try {
            System.out.printf("Server: getting message\n");
            String message = MessageUtils.getMessage(socket);
            System.out.printf("Server: got message: %s\n", message);
            Thread.sleep(1000);
            System.out.printf("Server: sending reply: %s\n", message);
            MessageUtils.sendMessage(socket, "Processed: " + message);
            System.out.printf("Server: sent\n");
            closeIgnoringException(socket);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    private void closeIgnoringException(Socket socket) {
        if (socket != null)
            try {
                socket.close();
            } catch (IOException ignore) {
            }
    }

    private void closeIgnoringException(ServerSocket serverSocket) {
        if (serverSocket != null)
            try {
                serverSocket.close();
            } catch (IOException ignore) {
            }
    }
}
```

Листинг А.5. MessageUtils.java

```
package common;

import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.net.Socket;

public class MessageUtils {
    public static void sendMessage(Socket socket, String message)
            throws IOException {
        OutputStream stream = socket.getOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(stream);
        oos.writeUTF(message);
        oos.flush();
    }
```

```
    public static String getMessage(Socket socket) throws IOException {
        InputStream stream = socket.getInputStream();
        ObjectInputStream ois = new ObjectInputStream(stream);
        return ois.readUTF();
    }
}
```

Архитектура «клиент/сервер» с использованием потоков

Перевод сервера на многопоточную архитектуру сводится к простому изменению функции process (новые строки выделены жирным шрифтом):

```
void process(final Socket socket) {
    if (socket == null)
        return;

    Runnable clientHandler = new Runnable() {
        public void run() {
            try {
                System.out.printf("Server: getting message\n");
                String message = MessageUtils.getMessage(socket);
                System.out.printf("Server: got message: %s\n", message);
                Thread.sleep(1000);
                System.out.printf("Server: sending reply: %s\n", message);
                MessageUtils.sendMessage(socket, "Processed: " + message);
                System.out.printf("Server: sent\n");
                closeIgnoringException(socket);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };

    Thread clientConnection = new Thread(clientHandler);
    clientConnection.start();
}
```

Б org.jfree.date.SerialDate

Листинг Б.1. SerialDate.Java

```
/* ===========================================================================
 * JCommon : библиотека классов общего назначения для платформы Java(tm)
 * ===========================================================================
 *
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 *
 * Информация о проекте:  http://www.jfree.org/jcommon/index.html
 *
 * Библиотека распространяется бесплатно; вы можете свободно распространять
 * и/или изменять ее на условиях лицензии Lesser General Public License
 * в формулировке Free Software Foundation; либо версии 2.1 лицензии, либо
 * (на ваше усмотрение) любой последующей версии.
 *
 * Библиотека распространяется в надежде, что она будет полезна, но
 * БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ, даже без подразумеваемой гарантии ПРИГОДНОСТИ
 * для КОНКРЕТНОЙ ЦЕЛИ. За подробностями обращайтесь к GNU Lesser General
 * Public License.
 *
 * Вы должны получить копию лицензии GNU Lesser General Public License
 * с этой библиотекой; если этого не произошло, обратитесь в Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 * USA.
 *
 * [Java является зарегистрированной торговой маркой Sun Microsystems, Inc.
 * в Соединенных Штатах и других странах].
 *
 * ---------------
 * SerialDate.java
 * ---------------
 * (C) Copyright 2001-2005, by Object Refinery Limited.
 *
 * Автор:  Дэвид Гилберт (для Object Refinery Limited);
 * Участники:   -;
 *
 * $Id: SerialDate.java,v 1.7 2005/11/03 09:25:17 mungady Exp $
```

```
36  *
37  * Изменения (начиная с 11 октября 2001)
38  * --------------------------
39  * 11.10.2001  : Реорганизация класса и его перемещение в новый пакет
40  *               com.jrefinery.date (DG);
41  * 05.12.2001  : Добавление метода getDescription(), исключение класса
42  *               NotableDate (DG);
43  * 12.12.2001  : После удаления класса NotableDate IBD требует наличия
44  *               метода setDescription() (DG); исправлены ошибки
45  *               в функциях getPreviousDayOfWeek(), getFollowingDayOfWeek()
46  *               и getNearestDayOfWeek() (DG);
47  * 05.12.2001  : Исправление ошибки в классе SpreadsheetDate (DG);
48  * 29.05.2002  : Перемещение констант месяцев в отдельный интерфейс
49  *               (MonthConstants) (DG);
50  * 27.08.2002  : Исправление ошибки в addMonths(), спасибо N???levka Petr (DG);
51  * 03.10.2002  : Исправление ошибок по информации Checkstyle (DG);
52  * 13.03.2003  : Реализация Serializable (DG);
53  * 29.05.2003  : Исправление ошибки в методе addMonths (DG);
54  * 04.09.2003  : Реализация Comparable. Обновление Javadoc для isInRange (DG);
55  * 05.01.2005  : Исправление ошибки в методе addYears() (1096282) (DG);
56  *
57  */
58
59 package org.jfree.date;
60
61 import java.io.Serializable;
62 import java.text.DateFormatSymbols;
63 import java.text.SimpleDateFormat;
64 import java.util.Calendar;
65 import java.util.GregorianCalendar;
66
67 /**
68  *  Абстрактный класс, определяющий требования для манипуляций с датами
69  *  без привязки к конкретной реализации.
70  *  <P>
71  *  Требование 1 : совпадение с представлением дат в формате Excel;
72  *  Требование 2 : класс должен быть неизменным;
73  *  <P>
74  *  Почему не использовать java.util.Date? Будем использовать, где это имеет смысл.
75  *  Класс java.util.Date бывмает *слишком* точным - он представляет момент
76  *  времени с точностью до 1/100 секунды (при этом сама дата зависит от часового
77  *  пояса). Иногда бывает нужно просто представить конкретный день (скажем,
78  *  21 января 2015), не заботясь о времени суток, часовом поясе и т.д.
79  *  Именно для таких ситуаций определяется класс SerialDate.
80  *  <P>
81  *  Вы можете вызвать getInstance() для получения конкретного субкласса
82  *  SerialDate, не беспокоясь о реализации.
83  *
84  * @author David Gilbert
85  */
```

продолжение ⇩

Листинг Б.1 (продолжение)

```
 86 public abstract class SerialDate implements Comparable,
 87                                            Serializable,
 88                                            MonthConstants {
 89
 90     /** Для сериализации. */
 91     private static final long serialVersionUID = -293716040467423637L;
 92
 93     /** Символические обозначения формата даты. */
 94     public static final DateFormatSymbols
 95         DATE_FORMAT_SYMBOLS = new SimpleDateFormat().getDateFormatSymbols();
 96
 97     /** Порядковый номер для 1 января 1900. */
 98     public static final int SERIAL_LOWER_BOUND = 2;
 99
100     /** Порядковый номер для 31 декабря 9999. */
101     public static final int SERIAL_UPPER_BOUND = 2958465;
102
103     /** Наименьшее значение года, поддерживаемое форматом даты. */
104     public static final int MINIMUM_YEAR_SUPPORTED = 1900;
105
106     /** Наибольшее значение года, поддерживаемое форматом даты. */
107     public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
108
109     /** Константа для понедельника, эквивалент java.util.Calendar.MONDAY. */
110     public static final int MONDAY = Calendar.MONDAY;
111
112     /**
113      * Константа для вторника, эквивалент java.util.Calendar.TUESDAY.
114      */
115     public static final int TUESDAY = Calendar.TUESDAY;
116
117     /**
118      * Константа для среды, эквивалент
119      * java.util.Calendar.WEDNESDAY.
120      */
121     public static final int WEDNESDAY = Calendar.WEDNESDAY;
122
123     /**
124      * Константа для четверга, эквивалент java.util.Calendar.THURSDAY.
125      */
126     public static final int THURSDAY = Calendar.THURSDAY;
127
128     /** Константа для пятницы, эквивалент java.util.Calendar.FRIDAY. */
129     public static final int FRIDAY = Calendar.FRIDAY;
130
131     /**
132      * Константа для субботы, эквивалент java.util.Calendar.SATURDAY.
133      */
134     public static final int SATURDAY = Calendar.SATURDAY;
135
```

```
136     /** Константа для воскресенья, эквивалент java.util.Calendar.SUNDAY. */
137     public static final int SUNDAY = Calendar.SUNDAY;
138
139     /** Количество дней в месяцах невисокосного года. */
140     static final int[] LAST_DAY_OF_MONTH =
141         {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
142
143     /** Количество дней от начала года до конца месяца в невисокосном годе. */
144     static final int[] AGGREGATE_DAYS_TO_END_OF_MONTH =
145         {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
146
147     /** Количество дней до конца предыдущего месяца. */
148     static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
149         {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
150
151     /** Количество дней от начала года до конца месяца в високосном годе. */
152     static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH =
153         {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
154
155     /**
156      * Количество дней до конца предыдущего месяца в високосном годе.
157      */
158     static final int[]
159         LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
160             {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
161
162     /** Константа для обозначения первой недели месяца. */
163     public static final int FIRST_WEEK_IN_MONTH = 1;
164
165     /** Константа для обозначения второй недели месяца. */
166     public static final int SECOND_WEEK_IN_MONTH = 2;
167
168     /** Константа для обозначения третьей недели месяца. */
169     public static final int THIRD_WEEK_IN_MONTH = 3;
170
171     /** Константа для обозначения четвертой недели месяца. */
172     public static final int FOURTH_WEEK_IN_MONTH = 4;
173
174     /** Константа для обозначения последней недели месяца. */
175     public static final int LAST_WEEK_IN_MONTH = 0;
176
177     /** Константа для обозначения типа диапазона. */
178     public static final int INCLUDE_NONE = 0;
179
180     /** Константа для обозначения типа диапазона. */
181     public static final int INCLUDE_FIRST = 1;
182
183     /** Константа для обозначения типа диапазона. */
184     public static final int INCLUDE_SECOND = 2;
185
186     /** Константа для обозначения типа диапазона. */
```

продолжение ⇩

Листинг Б.1 (продолжение)

```
187     public static final int INCLUDE_BOTH = 3;
188
189     /**
190      * Useful constant for specifying a day of the week relative to a fixed
191      * date.
192      */
193     public static final int PRECEDING = -1;
194
195     /**
196      * Константа для определения дня недели относительно
197      * фиксированной даты.
198      */
199     public static final int NEAREST = 0;
200
201     /**
202      * Константа для определения дня недели относительно
203      * фиксированной даты.
204      */
205     public static final int FOLLOWING = 1;
206
207     /** Описание даты. */
208     private String description;
209
210     /**
211      * Конструктор по умолчанию.
212      */
213     protected SerialDate() {
214     }
215
216     /**
217      * Возвращает <code>true</code>, если целое число code соответствует
218      * действительному дню недели, или <code>false</code> в противном случае.
219      *
220      * @param code  код, проверяемый на допустимость.
221      *
222      * @return <code>true</code>, если целое число code соответствует
223      *  действительному дню недели, <code>false</code> в противном случае.
224      */
225     public static boolean isValidWeekdayCode(final int code) {
226
227         switch(code) {
228             case SUNDAY:
229             case MONDAY:
230             case TUESDAY:
231             case WEDNESDAY:
232             case THURSDAY:
233             case FRIDAY:
234             case SATURDAY:
235                 return true;
236             default:
```

```
237                 return false;
238             }
239
240         }
241
242         /**
243          * Преобразует переданную строку в день недели.
244          *
245          * @param s  строка, представляющая день недели.
246          *
247          * @return <code>-1</code>, если строка не преобразуется, день недели
248          *         в противном случае.
249          */
250         public static int stringToWeekdayCode(String s) {
251
252             final String[] shortWeekdayNames
253                 = DATE_FORMAT_SYMBOLS.getShortWeekdays();
254             final String[] weekDayNames = DATE_FORMAT_SYMBOLS.getWeekdays();
255
256             int result = -1;
257             s = s.trim();
258             for (int i = 0; i < weekDayNames.length; i++) {
259                 if (s.equals(shortWeekdayNames[i])) {
260                     result = i;
261                     break;
262                 }
263                 if (s.equals(weekDayNames[i])) {
264                     result = i;
265                     break;
266                 }
267             }
268             return result;
269
270         }
271
272         /**
273          * Возвращает строку, представляющую заданный день недели.
274          * <P>
275          * Необходимо поискать более элегантное решение.
276          *
277          * @param weekday  день недели.
278          *
279          * @return строка, представляющая заданный день недели.
280          */
281         public static String weekdayCodeToString(final int weekday) {
282
283             final String[] weekdays = DATE_FORMAT_SYMBOLS.getWeekdays();
284             return weekdays[weekday];
285
286         }
```

продолжение ⇘

Листинг Б.1 (продолжение)

```
287
288     /**
289      * Возвращает массив названий месяцев.
290      *
291      * @return массив названий месяцев.
292      */
293     public static String[] getMonths() {
294
295         return getMonths(false);
296
297     }
298
299     /**
300      * Возвращает массив названий месяцев.
301      *
302      * @param shortened  флаг, указывающий на необходимость возврата
303      *                   сокращенных названий месяцев.
304      *
305      * @return массив названий месяцев.
306      */
307     public static String[] getMonths(final boolean shortened) {
308
309         if (shortened) {
310             return DATE_FORMAT_SYMBOLS.getShortMonths();
311         }
312         else {
313             return DATE_FORMAT_SYMBOLS.getMonths();
314         }
315
316     }
317
318     /**
319      * Возвращает true, если целое число code соответствует действительному месяцу.
320      *
321      * @param code  Код, проверяемый на действительность.
322      *
323      * @return <code>true</code>, если целое число code соответствует
324      *         действительному месяцу.
325      */
326     public static boolean isValidMonthCode(final int code) {
327
328         switch(code) {
329             case JANUARY:
330             case FEBRUARY:
331             case MARCH:
332             case APRIL:
333             case MAY:
334             case JUNE:
335             case JULY:
336             case AUGUST:
```

```
337             case SEPTEMBER:
338             case OCTOBER:
339             case NOVEMBER:
340             case DECEMBER:
341                 return true;
342             default:
343                 return false;
344         }
345
346     }
347
348     /**
349      * Возвращает квартал для заданного месяца.
350      *
351      * @param code  код месяца (1-12).
352      *
353      * @return квартал, к которому относится месяц.
354      * @throws java.lang.IllegalArgumentException
355      */
356     public static int monthCodeToQuarter(final int code) {
357
358         switch(code) {
359             case JANUARY:
360             case FEBRUARY:
361             case MARCH: return 1;
362             case APRIL:
363             case MAY:
364             case JUNE: return 2;
365             case JULY:
366             case AUGUST:
367             case SEPTEMBER: return 3;
368             case OCTOBER:
369             case NOVEMBER:
370             case DECEMBER: return 4;
371             default: throw new IllegalArgumentException(
372                 "SerialDate.monthCodeToQuarter: invalid month code.");
373         }
374
375     }
376
377     /**
378      * Возвращает строку, представляющую заданный месяц.
379      * <P>
380      * Строка возвращается в форме длинного названия месяца
381      * из локального контекста по умолчанию.
382      *
383      * @param month  месяц.
384      *
385      * @return строка, представляющая заданный месяц.
386      */
```

продолжение ⇩

Листинг Б.1 (продолжение)

```
387     public static String monthCodeToString(final int month) {
388
389         return monthCodeToString(month, false);
390
391     }
392
393     /**
394      * Возвращает строку, представляющую заданный месяц.
395      * <P>
396      * Строка возвращается в форме длинного или короткого названия месяца
397      * из локального контекста по умолчанию.
398      *
399      * @param month  месяц.
400      * @param shortened  если <code>true</code> возвращает сокращенное
401      *                   название месяца.
402      *
403      * @return строка, представляющая заданный месяц.
404      * @throws java.lang.IllegalArgumentException
405      */
406     public static String monthCodeToString(final int month,
407                                            final boolean shortened) {
408
409         // Проверка аргументов...
410         if (!isValidMonthCode(month)) {
411             throw new IllegalArgumentException(
412                 "SerialDate.monthCodeToString: month outside valid range.");
413         }
414
415         final String[] months;
416
417         if (shortened) {
418             months = DATE_FORMAT_SYMBOLS.getShortMonths();
419         }
420         else {
421             months = DATE_FORMAT_SYMBOLS.getMonths();
422         }
423
424         return months[month - 1];
425
426     }
427
428     /**
429      * Преобразует строку в код месяца.
430      * <P>
431      * Метод возвращает одну из констант JANUARY, FEBRUARY, ...,
432      * DECEMBER, соответствующую заданной строке. Если строка не распознается,
433      * метод возвращает -1.
434      *
435      * @param s  строка для обработки.
436      *
```

```
437      * @return <code>-1</code>, если строка не разбирается, месяц года
438      *         в противном случае.
439      */
440     public static int stringToMonthCode(String s) {
441
442         final String[] shortMonthNames = DATE_FORMAT_SYMBOLS.getShortMonths();
443         final String[] monthNames = DATE_FORMAT_SYMBOLS.getMonths();
444
445         int result = -1;
446         s = s.trim();
447
448         // Сначала пытаемся разобрать строку как целое число (1-12)...
449         try {
450             result = Integer.parseInt(s);
451         }
452         catch (NumberFormatException e) {
453             // Подавление
454         }
455
456         // Теперь ищем по названиям месяцев...
457         if ((result < 1) || (result > 12)) {
458             for (int i = 0; i < monthNames.length; i++) {
459                 if (s.equals(shortMonthNames[i])) {
460                     result = i + 1;
461                     break;
462                 }
463                 if (s.equals(monthNames[i])) {
464                     result = i + 1;
465                     break;
466                 }
467             }
468         }
469
470         return result;
471
472     }
473
474     /**
475      * Возвращает true, если целое число code представляет действительную
476      * неделю месяца, или false в противном случае.
477      *
478      * @param code  код, проверяемый на действительность.
479      * @return <code>true</code>, если целое число code представляет
480      *         действительную неделю месяца.
481      */
482     public static boolean isValidWeekInMonthCode(final int code) {
483
484         switch(code) {
485             case FIRST_WEEK_IN_MONTH:
486             case SECOND_WEEK_IN_MONTH:
487             case THIRD_WEEK_IN_MONTH:
```

продолжение ⇩

Листинг Б.1 (продолжение)

```
488             case FOURTH_WEEK_IN_MONTH:
489             case LAST_WEEK_IN_MONTH: return true;
490             default: return false;
491         }
492
493     }
494
495     /**
496      * Определяет, является ли заданный год високосным.
497      *
498      * @param yyyy  год (в диапазоне от 1900 до 9999).
499      *
500      * @return <code>true</code>, если заданный код является високосным.
501      */
502     public static boolean isLeapYear(final int yyyy) {
503
504         if ((yyyy % 4) != 0) {
505             return false;
506         }
507         else if ((yyyy % 400) == 0) {
508             return true;
509         }
510         else if ((yyyy % 100) == 0) {
511             return false;
512         }
513         else {
514             return true;
515         }
516
517     }
518
519     /**
520      * Возвращает количество високосных годов от 1900 до заданного года
521      * ВКЛЮЧИТЕЛЬНО.
522      * <P>
523      * Учтите, что 1900 год високосным не является.
524      *
525      * @param yyyy  год (в диапазоне от 1900 до 9999).
526      *
527      * @return количество високосных годов от 1900 до заданного года.
528      */
529     public static int leapYearCount(final int yyyy) {
530
531         final int leap4 = (yyyy - 1896) / 4;
532         final int leap100 = (yyyy - 1800) / 100;
533         final int leap400 = (yyyy - 1600) / 400;
534         return leap4 - leap100 + leap400;
535
536     }
537
```

```
538     /**
539      * Возвращает номер последнего дня месяца с учетом
540      * високосных годов.
541      *
542      * @param month  месяц.
543      * @param yyyy  год (в диапазоне от 1900 до 9999).
544      *
545      * @return номер последнего дня месяца.
546      */
547     public static int lastDayOfMonth(final int month, final int yyyy) {
548
549         final int result = LAST_DAY_OF_MONTH[month];
550         if (month != FEBRUARY) {
551             return result;
552         }
553         else if (isLeapYear(yyyy)) {
554             return result + 1;
555         }
556         else {
557             return result;
558         }
559
560     }
561
562     /**
563      * Создает новую дату, прибавляя заданное количество дней
564      *  к базовой дате.
565      *
566      * @param days  количество прибавляемых дней (может быть отрицательным).
567      * @param base  базовая дата.
568      *
569      * @return новая дата.
570      */
571     public static SerialDate addDays(final int days, final SerialDate base) {
572
573         final int serialDayNumber = base.toSerial() + days;
574         return SerialDate.createInstance(serialDayNumber);
575
576     }
577
578     /**
579      * Создает новую дату, прибавляя заданное количество месяцев
580      * к базовой дате.
581      * <P>
582      * Если базовая дата близка к концу месяца, результат может слегка
583      * смещаться:  31 мая + 1 месяц = 30 июня
584      *
585      * @param months  количество прибавляемых месяцев (может быть отрицательным).
586      * @param base  базовая дата.
587      *
```

продолжение ⇩

Листинг Б.1 (продолжение)

```
588      * @return новая дата.
589      */
590     public static SerialDate addMonths(final int months,
591                                        final SerialDate base) {
592
593         final int yy = (12 * base.getYYYY() + base.getMonth() + months - 1)
594                        / 12;
595         final int mm = (12 * base.getYYYY() + base.getMonth() + months - 1)
596                        % 12 + 1;
597         final int dd = Math.min(
598             base.getDayOfMonth(), SerialDate.lastDayOfMonth(mm, yy)
599         );
600         return SerialDate.createInstance(dd, mm, yy);
601
602     }
603
604     /**
605      * Создает новую дату, прибавляя заданное количество лет
606      * к базовой дате.
607      *
608      * @param years  количество прибавляемых лет (может быть отрицательным).
609      * @param base  базовая дата.
610      *
611      * @return новая дата.
612      */
613     public static SerialDate addYears(final int years, final SerialDate base) {
614
615         final int baseY = base.getYYYY();
616         final int baseM = base.getMonth();
617         final int baseD = base.getDayOfMonth();
618
619         final int targetY = baseY + years;
620         final int targetD = Math.min(
621             baseD, SerialDate.lastDayOfMonth(baseM, targetY)
622         );
623
624         return SerialDate.createInstance(targetD, baseM, targetY);
625
626     }
627
628     /**
629      * Возвращает последнюю дату, приходящуюся  на заданный день недели,
630      * ПРЕДШЕСТВУЮЩУЮ базовой дате.
631      *
632      * @param targetWeekday  код дня недели.
633      * @param base  базовая дата.
634      *
635      * @return последняя дата, приходящаяся на заданный день недели,
636      *         ПРЕДШЕСТВУЮЩАЯ базовой дате.
637      */
```

```
638     public static SerialDate getPreviousDayOfWeek(final int targetWeekday,
639                                                    final SerialDate base) {
640
641         // Проверить аргументы...
642         if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
643             throw new IllegalArgumentException(
644                 "Invalid day-of-the-week code."
645             );
646         }
647
648         // Определить дату...
649         final int adjust;
650         final int baseDOW = base.getDayOfWeek();
651         if (baseDOW > targetWeekday) {
652             adjust = Math.min(0, targetWeekday - baseDOW);
653         }
654         else {
655             adjust = -7 + Math.max(0, targetWeekday - baseDOW);
656         }
657
658         return SerialDate.addDays(adjust, base);
659
660     }
661
662     /**
663      * Возвращает самую раннюю дату, приходящуюся  на заданный день недели
664      * ПОСЛЕ базовой даты.
665      *
666      * @param targetWeekday  код дня недели.
667      * @param base  базовая дата.
668      *
669      * @return самая ранняя дата, приходящаяся на заданный день недели
670      *         ПОСЛЕ базовой даты.
671      */
672     public static SerialDate getFollowingDayOfWeek(final int targetWeekday,
673                                                    final SerialDate base) {
674
675         // Проверить аргументы...
676         if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
677             throw new IllegalArgumentException(
678                 "Invalid day-of-the-week code."
679             );
680         }
681
682         // Определить дату...
683         final int adjust;
684         final int baseDOW = base.getDayOfWeek();
685         if (baseDOW > targetWeekday) {
686             adjust = 7 + Math.min(0, targetWeekday - baseDOW);
687         }
688         else {
```

продолжение ⇩

Листинг Б.1 (продолжение)

```
689             adjust = Math.max(0, targetWeekday - baseDOW);
690         }
691
692         return SerialDate.addDays(adjust, base);
693     }
694
695     /**
696      * Возвращает дату, приходящуюся  на заданный день недели,
697      * САМУЮ БЛИЗКУЮ к базовой дате.
698      *
699      * @param targetDOW  код дня недели.
700      * @param base  базовая дата.
701      *
702      * @return дата, приходящаяся  на заданный день недели,
703      *         САМАЯ БЛИЗКАЯ к базовой дате.
704      */
705     public static SerialDate getNearestDayOfWeek(final int targetDOW,
706                                                   final SerialDate base) {
707
708         // Проверить аргументы...
709         if (!SerialDate.isValidWeekdayCode(targetDOW)) {
710             throw new IllegalArgumentException(
711                 "Invalid day-of-the-week code."
712             );
713         }
714
715         // Определить дату...
716         final int baseDOW = base.getDayOfWeek();
717         int adjust = -Math.abs(targetDOW - baseDOW);
718         if (adjust >= 4) {
719             adjust = 7 - adjust;
720         }
721         if (adjust <= -4) {
722             adjust = 7 + adjust;
723         }
724         return SerialDate.addDays(adjust, base);
725
726     }
727
728     /**
729      * Перемещает дату к последнему дню месяца.
730      *
731      * @param base  базовая дата.
732      *
733      * @return новая дата.
734      */
735     public SerialDate getEndOfCurrentMonth(final SerialDate base) {
736         final int last = SerialDate.lastDayOfMonth(
737             base.getMonth(), base.getYYYY()
738         );
```

```
739         return SerialDate.createInstance(last, base.getMonth(), base.getYYYY());
740     }
741
742     /**
743      * Возвращает строку, соответствующую коду недели в месяце.
744      * <P>
745      * Необходимо поискать более элегантное решение.
746      *
747      * @param count  целочисленный код недели в месяце.
748      *
749      * @return строка, соответствующая коду недели в месяце.
750      */
751     public static String weekInMonthToString(final int count) {
752
753         switch (count) {
754             case SerialDate.FIRST_WEEK_IN_MONTH : return "First";
755             case SerialDate.SECOND_WEEK_IN_MONTH : return "Second";
756             case SerialDate.THIRD_WEEK_IN_MONTH : return "Third";
757             case SerialDate.FOURTH_WEEK_IN_MONTH : return "Fourth";
758             case SerialDate.LAST_WEEK_IN_MONTH : return "Last";
759             default :
760                 return "SerialDate.weekInMonthToString(): invalid code.";
761         }
762
763     }
764
765     /**
766      * Возвращает строку, представляющую переданное значение relative.
767      * <P>
768      * Необходимо поискать более элегантное решение.
769      *
770      * @param relative  константа, представляющая значение'relative'.
771      *
772      * @return строка, представляющая переданное значение 'relative'.
773      */
774     public static String relativeToString(final int relative) {
775
776         switch (relative) {
777             case SerialDate.PRECEDING : return "Preceding";
778             case SerialDate.NEAREST : return "Nearest";
779             case SerialDate.FOLLOWING : return "Following";
780             default : return "ERROR : Relative To String";
781         }
782
783     }
784
785     /**
786      * Метод-фабрика, возвращающий экземпляр конкретного субкласса
787      * {@link SerialDate}.
788      *
```

продолжение ⇩

Листинг Б.1 (продолжение)

```
789      * @param day  день (1-31).
790      * @param month  месяц (1-12).
791      * @param yyyy  год (в диапазоне от 1900 до 9999).
792      *
793      * @return Экземпляр {@link SerialDate}.
794      */
795     public static SerialDate createInstance(final int day, final int month,
796                                              final int yyyy) {
797         return new SpreadsheetDate(day, month, yyyy);
798     }
799
800     /**
801      * Метод-фабрика, возвращающий экземпляр конкретного субкласса
802      * {@link SerialDate}.
803      *
804      * @param serial  порядковый номер дня (1 января 1900 = 2).
805      *
806      * @return экземпляр SerialDate.
807      */
808     public static SerialDate createInstance(final int serial) {
809         return new SpreadsheetDate(serial);
810     }
811
812     /**
813      * Метод-фабрика, возвращающий экземпляр субкласса SerialDate.
814      *
815      * @param date  объект даты Java.
816      *
817      * @return экземпляр SerialDate.
818      */
819     public static SerialDate createInstance(final java.util.Date date) {
820
821         final GregorianCalendar calendar = new GregorianCalendar();
822         calendar.setTime(date);
823         return new SpreadsheetDate(calendar.get(Calendar.DATE),
824                                    calendar.get(Calendar.MONTH) + 1,
825                                    calendar.get(Calendar.YEAR));
826
827     }
828
829     /**
830      * Возвращает порядковый номер для даты, где 1 January 1900 = 2 (что почти
831      * соответствует системе нумерации, используемой в Microsoft Excel for
832      * Windows и Lotus 1-2-3).
833      *
834      * @return порядковый номер даты.
835      */
836     public abstract int toSerial();
837
838     /**
```

```
839      * Возвращает java.util.Date. Поскольку java.util.Date превосходит SerialDate
840      * по точности, необходимо определить схему выбора 'времени суток'.
841      *
842      * @return текущий объект в виде <code>java.util.Date</code>.
843      */
844     public abstract java.util.Date toDate();
845
846     /**
847      * Возвращает описание даты.
848      *
849      * @return описание даты.
850      */
851     public String getDescription() {
852         return this.description;
853     }
854
855     /**
856      * Задает описание даты.
857      *
858      * @param description  новое описание даты.
859      */
860     public void setDescription(final String description) {
861         this.description = description;
862     }
863
864     /**
865      * Преобразует дату в строку.
866      *
867      * @return  строковое представление даты.
868      */
869     public String toString() {
870         return getDayOfMonth() + "-" + SerialDate.monthCodeToString(getMonth())
871                                + "-" + getYYYY();
872     }
873
874     /**
875      * Возвращает год (в действительном диапазоне от 1900 до 9999).
876      *
877      * @return год.
878      */
879     public abstract int getYYYY();
880
881     /**
882      * Возвращает месяц (январь = 1, февраль = 2, март = 3).
883      *
884      * @return месяц.
885      */
886     public abstract int getMonth();
887
888     /**
```

продолжение ⇩

Листинг Б.1 (продолжение)

```
889      * Возвращает день месяца.
890      *
891      * @return день месяца.
892      */
893     public abstract int getDayOfMonth();
894
895     /**
896      * Возвращает день недели.
897      *
898      * @return день недели.
899      */
900     public abstract int getDayOfWeek();
901
902     /**
903      * Возвращает разность (в днях) между текущей и заданной
904      * 'другой' датой.
905      * <P>
906      * Результат положительный, если текущая дата следует после 'другой',
907      * или отрицателен, если текущая дата предшествует 'другой'.
908      *
909      * @param other  дата для сравнения.
910      *
911      * @return разность между текущей и другой датой.
912      */
913     public abstract int compare(SerialDate other);
914
915     /**
916      * Возвращает true, если текущий объект SerialDate представляет ту же дату,
917      * что и заданный объект SerialDate.
918      *
919      * @param other  дата для сравнения.
920      *
921      * @return <code>true</code>, если текущий объект SerialDate представляет
922      *         ту же дату, что и заданный объект SerialDate.
923      */
924     public abstract boolean isOn(SerialDate other);
925
926     /**
927      * Возвращает true, если текущий объект SerialDate представляет более раннюю
928      * дату по сравнению с заданным объектом SerialDate.
929      *
930      * @param other  дата для сравнения.
931      *
932      * @return <code>true</code>, если текущий объект SerialDate представляет
933      *         более раннюю дату по сравнению с заданным объектом SerialDate.
934      */
935     public abstract boolean isBefore(SerialDate other);
936
937     /**
938      * Возвращает true, если текущий объект SerialDate представляет ту же
```

```
939      * дату, что и заданный объект SerialDate.
940      *
941      * @param other  дата для сравнения.
942      *
943      * @return <code>true<code>, если текущий объект SerialDate представляет
944      *          ту же дату, что и заданный объект SerialDate.
945      */
946     public abstract boolean isOnOrBefore(SerialDate other);
947
948     /**
949      * Возвращает true, если текущий объект SerialDate представляет ту же
950      * дату, что и заданный объект SerialDate.
951      *
952      * @param other  дата для сравнения.
953      *
954      * @return <code>true</code>, если текущий объект SerialDate представляет
955      *         ту же дату, что и заданный объект SerialDate.
956      */
957     public abstract boolean isAfter(SerialDate other);
958
959     /**
960      * Возвращает true, если текущий объект SerialDate представляет ту же
961      * дату, что и заданный объект SerialDate.
962      *
963      * @param other  дата для сравнения.
964      *
965      * @return <code>true</code>, если текущий объект SerialDate представляет
966      *         ту же дату, что и заданный объект SerialDate.
967      */
968     public abstract boolean isOnOrAfter(SerialDate other);
969
970     /**
971      * Возвращает <code>true</code>, если текущий {@link SerialDate} принадлежит
972      * заданному диапазону (режим INCLUSIVE).  Порядок дат d1 и d2
973      * не важен.
974      *
975      * @param d1  граничная дата диапазона.
976      * @param d2  другая граничная дата диапазона.
977      *
978      * @return Логический признак.
979      */
980     public abstract boolean isInRange(SerialDate d1, SerialDate d2);
981
982     /**
983      * Возвращает <code>true</code> если текущий {@link SerialDate} принадлежит
984      * заданному диапазону (включение границ указывается при вызове). Порядок
985      * дат d1 и d2 не важен.
986      *
987      * @param d1  граничная дата диапазона.
988      * @param d2  другая граничная дата диапазона.
```

продолжение ⇩

Листинг Б.1 (продолжение)

```
     * @param include  код, управляющий включением начальной и конечной дат
     *                 в диапазон.
     *
     * @return Логический признак.
     */
    public abstract boolean isInRange(SerialDate d1, SerialDate d2,
                                      int include);

    /**
     * Возвращает последнюю дату, приходящуюся  на заданный день недели,
     * ПРЕДШЕСТВУЮЩУЮ текущей дате.
     *
     * @param targetDOW  код дня недели.
     *
     * @return последняя дата, приходящаяся  на заданный день недели,
     *         ПРЕДШЕСТВУЮЩАЯ текущей дате.
     */
    public SerialDate getPreviousDayOfWeek(final int targetDOW) {
        return getPreviousDayOfWeek(targetDOW, this);
    }

    /**
     * Возвращает самую раннюю дату, приходящуюся  на заданный день недели,
     * ПОСЛЕ текущей даты.
     *
     * @param targetDOW  код дня недели.
     *
     * @return самая ранняя дата, приходящаяся на заданный день недели
     *         ПОСЛЕ текущей даты.
     */
    public SerialDate getFollowingDayOfWeek(final int targetDOW) {
        return getFollowingDayOfWeek(targetDOW, this);
    }

    /**
     * Возвращает ближайшую дату, приходящуюся  на заданный день недели,
     *
     * @param targetDOW  код дня недели.
     *
     * @return ближайшая дата, приходящаяся  на заданный день недели,
     */
    public SerialDate getNearestDayOfWeek(final int targetDOW) {
        return getNearestDayOfWeek(targetDOW, this);
    }

}
```

Листинг Б.2. SerialDateTest.java

```
/* ===========================================================================
 * JCommon : библиотека классов общего назначения для платформы Java(tm)
 * ===========================================================================
 *
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 *
 * Информация о проекте:  http://www.jfree.org/jcommon/index.html
 *
 * Библиотека распространяется бесплатно; вы можете свободно распространять
 * и/или изменять ее на условиях лицензии Lesser General Public License
 * в формулировке Free Software Foundation; либо версии 2.1 лицензии, либо
 * (на ваше усмотрение) любой последующей версии.
 *
 * Библиотека распространяется в надежде, что она будет полезна, но
 * БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ, даже без подразумеваемой гарантии ПРИГОДНОСТИ
 * для КОНКРЕТНОЙ ЦЕЛИ. За подробностями обращайтесь к GNU Lesser General
 * Public License.
 *
 * Вы должны получить копию лицензии GNU Lesser General Public License
 * с этой библиотекой; если этого не произошло, обратитесь в Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 * USA.
 *
 * [Java является зарегистрированной торговой маркой Sun Microsystems, Inc.
 * в Соединенных Штатах и других странах].
 *
 * --------------------
 * SerialDateTests.java
 * --------------------
 * (C) Copyright 2001-2005, by Object Refinery Limited.
 *
 * Автор:  Дэвид Гилберт (для Object Refinery Limited);
 * Участники:   -;
 *
 * $Id: SerialDateTests.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
 *
 * Изменения
 * -------------------------
 * 15.12.2001  : Версия 1 (DG);
 * 25.06.2002  : Удаление лишнего импорта (DG);
 * 24.10.2002  : Исправление ошибок по информации Checkstyle (DG);
 * 13.03.2003  : Добавление теста сериализации (DG);
 * 05.01.2005  : Добавление теста для ошибки по отчету 1096282 (DG);
 *
 */

package org.jfree.date.junit;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
```

продолжение ⇩

Листинг Б.2 (продолжение)

```
 51 import java.io.ObjectInput;
 52 import java.io.ObjectInputStream;
 53 import java.io.ObjectOutput;
 54 import java.io.ObjectOutputStream;
 55
 56 import junit.framework.Test;
 57 import junit.framework.TestCase;
 58 import junit.framework.TestSuite;
 59
 60 import org.jfree.date.MonthConstants;
 61 import org.jfree.date.SerialDate;
 62
 63 /**
 64  * Тесты JUnit для класса {@link SerialDate}.
 65  */
 66 public class SerialDateTests extends TestCase {
 67
 68     /** Дата, представляющая 9 ноября. */
 69     private SerialDate nov9Y2001;
 70
 71     /**
 72      * Создает новый тестовый сценарий.
 73      *
 74      * @param name  the name.
 75      */
 76     public SerialDateTests(final String name) {
 77         super(name);
 78     }
 79
 80     /**
 81      * Возвращает пакет тестов для системы запуска тестов JUnit.
 82      *
 83      * @return тестовый пакет.
 84      */
 85     public static Test suite() {
 86         return new TestSuite(SerialDateTests.class);
 87     }
 88
 89     /**
 90      * Подготовка задачи.
 91      */
 92     protected void setUp() {
 93       this.nov9Y2001 = SerialDate.createInstance(9, MonthConstants.NOVEMBER, 2001);
 94     }
 95
 96     /**
 97      * 9 ноября 2001 + 2 месяца = должно быть 9 января 2002.
 98      */
 99     public void testAddMonthsTo9Nov2001() {
100         final SerialDate jan9Y2002 = SerialDate.addMonths(2, this.nov9Y2001);
101         final SerialDate answer = SerialDate.createInstance(9, 1, 2002);
```

```
102         assertEquals(answer, jan9Y2002);
103     }
104
105     /**
106      * Тестовый сценарий для известной ошибки (исправлено).
107      */
108     public void testAddMonthsTo5Oct2003() {
109         final SerialDate d1 = SerialDate.createInstance(5, MonthConstants.OCTOBER,
            2003);
110         final SerialDate d2 = SerialDate.addMonths(2, d1);
111         assertEquals(d2, SerialDate.createInstance(5, MonthConstants.DECEMBER,
            2003));
112     }
113
114     /**
115      * Тестовый сценарий для известной ошибки (исправлено).
116      */
117     public void testAddMonthsTo1Jan2003() {
118         final SerialDate d1 = SerialDate.createInstance(1, MonthConstants.JANUARY,
            2003);
119         final SerialDate d2 = SerialDate.addMonths(0, d1);
120         assertEquals(d2, d1);
121     }
122
123     /**
124      * Понедельник, предшествующий 9 ноября 2001, - должно быть 5 ноября.
125      */
126     public void testMondayPrecedingFriday9Nov2001() {
127         SerialDate mondayBefore = SerialDate.getPreviousDayOfWeek(
128             SerialDate.MONDAY, this.nov9Y2001
129         );
130         assertEquals(5, mondayBefore.getDayOfMonth());
131     }
132
133     /**
134      * Понедельник, следующий за 9 ноября 2001, - должно быть 12 ноября.
135      */
136     public void testMondayFollowingFriday9Nov2001() {
137         SerialDate mondayAfter = SerialDate.getFollowingDayOfWeek(
138             SerialDate.MONDAY, this.nov9Y2001
139         );
140         assertEquals(12, mondayAfter.getDayOfMonth());
141     }
142
143     /**
144      * Понедельник, ближайший к 9 ноября 2001, - должно быть 12 ноября.
145      */
146     public void testMondayNearestFriday9Nov2001() {
147         SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(
148             SerialDate.MONDAY, this.nov9Y2001
149         );
150         assertEquals(12, mondayNearest.getDayOfMonth());
```

продолжение ⇩

Листинг Б.2 (продолжение)x

```
151     }
152
153     /**
154      * Понедельник, ближайший к 22 января 1970, - должно быть 19-е января.
155      */
156     public void testMondayNearest22Jan1970() {
157         SerialDate jan22Y1970 = SerialDate.createInstance(22, MonthConstants.JANUARY,
            1970);
158         SerialDate mondayNearest=SerialDate.getNearestDayOfWeek(SerialDate.MONDAY,
            jan22Y1970);
159         assertEquals(19, mondayNearest.getDayOfMonth());
160     }
161
162     /**
163      * Проверяет преобразование дней в строки. На самом деле результат
164      * зависит от локального контекста, так что тест следует изменить.
165      */
166     public void testWeekdayCodeToString() {
167
168         final String test = SerialDate.weekdayCodeToString(SerialDate.SATURDAY);
169         assertEquals("Saturday", test);
170
171     }
172
173     /**
174      * Проверяет преобразование строки в день недели. Если в локальном контексте
175      * не используются английские названия дней недели, тест не пройдет (улучшить)!
176      */
177     public void testStringToWeekday() {
178
179         int weekday = SerialDate.stringToWeekdayCode("Wednesday");
180         assertEquals(SerialDate.WEDNESDAY, weekday);
181
182         weekday = SerialDate.stringToWeekdayCode(" Wednesday ");
183         assertEquals(SerialDate.WEDNESDAY, weekday);
184
185         weekday = SerialDate.stringToWeekdayCode("Wed");
186         assertEquals(SerialDate.WEDNESDAY, weekday);
187
188     }
189
190     /**
191      * Проверяет преобразование строки в месяц. Если в локальном контексте
192      * не используются английские названия месяцев, тест не пройдет (улучшить)!
193      */
194     public void testStringToMonthCode() {
195
196         int m = SerialDate.stringToMonthCode("January");
197         assertEquals(MonthConstants.JANUARY, m);
198
199         m = SerialDate.stringToMonthCode(" January ");
```

```
200         assertEquals(MonthConstants.JANUARY, m);
201
202         m = SerialDate.stringToMonthCode("Jan");
203         assertEquals(MonthConstants.JANUARY, m);
204
205     }
206
207     /**
208      * Проверяет преобразование кода месяца в строку.
209      */
210     public void testMonthCodeToStringCode() {
211
212         final String test = SerialDate.monthCodeToString(MonthConstants.DECEMBER);
213         assertEquals("December", test);
214
215     }
216
217     /**
218      * Год 1900 не является високосным.
219      */
220     public void testIsNotLeapYear1900() {
221         assertTrue(!SerialDate.isLeapYear(1900));
222     }
223
224     /**
225      * Год 2000 - високосный.
226      */
227     public void testIsLeapYear2000() {
228         assertTrue(SerialDate.isLeapYear(2000));
229     }
230
231     /**
232      * Количество високосных годов с 1900 до 1899 включительно равно 0.
233      */
234     public void testLeapYearCount1899() {
235         assertEquals(SerialDate.leapYearCount(1899), 0);
236     }
237
238     /**
239      * Количество високосных годов с 1900 до 1903 включительно равно 0.
240      */
241     public void testLeapYearCount1903() {
242         assertEquals(SerialDate.leapYearCount(1903), 0);
243     }
244
245     /**
246      * Количество високосных годов с 1900 до 1904 включительно равно 1.
247      */
248     public void testLeapYearCount1904() {
249         assertEquals(SerialDate.leapYearCount(1904), 1);
250     }
251
252     /**
```

продолжение ⇩

Листинг Б.2 (продолжение)

```
253      * Количество високосных годов с 1900 до 1999 включительно равно 24.
254      */
255     public void testLeapYearCount1999() {
256         assertEquals(SerialDate.leapYearCount(1999), 24);
257     }
258
259     /**
260      * Количество високосных годов с 1900 до 2000 включительно равно 25.
261      */
262     public void testLeapYearCount2000() {
263         assertEquals(SerialDate.leapYearCount(2000), 25);
264     }
265
266     /**
267      * Сериализовать экземпляр, восстановить и проверить на равенство.
268      */
269     public void testSerialization() {
270
271         SerialDate d1 = SerialDate.createInstance(15, 4, 2000);
272         SerialDate d2 = null;
273
274         try {
275             ByteArrayOutputStream buffer = new ByteArrayOutputStream();
276             ObjectOutput out = new ObjectOutputStream(buffer);
277             out.writeObject(d1);
278             out.close();
279
280             ObjectInput in = new ObjectInputStream(
                  new ByteArrayInputStream(buffer.toByteArray()));
281             d2 = (SerialDate) in.readObject();
282             in.close();
283         }
284         catch (Exception e) {
285             System.out.println(e.toString());
286         }
287         assertEquals(d1, d2);
288
289     }
290
291     /**
292      * Тест для ошибки по отчету 1096282 (исправлено).
293      */
294     public void test1096282() {
295         SerialDate d = SerialDate.createInstance(29, 2, 2004);
296         d = SerialDate.addYears(1, d);
297         SerialDate expected = SerialDate.createInstance(28, 2, 2005);
298         assertTrue(d.isOn(expected));
299     }
300
301     /**
```

```
302      * Различные тесты для метода addMonths().
303      */
304     public void testAddMonths() {
305         SerialDate d1 = SerialDate.createInstance(31, 5, 2004);
306
307         SerialDate d2 = SerialDate.addMonths(1, d1);
308         assertEquals(30, d2.getDayOfMonth());
309         assertEquals(6, d2.getMonth());
310         assertEquals(2004, d2.getYYYY());
311
312         SerialDate d3 = SerialDate.addMonths(2, d1);
313         assertEquals(31, d3.getDayOfMonth());
314         assertEquals(7, d3.getMonth());
315         assertEquals(2004, d3.getYYYY());
316
317         SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
318         assertEquals(30, d4.getDayOfMonth());
319         assertEquals(7, d4.getMonth());
320         assertEquals(2004, d4.getYYYY());
321     }
322 }
```

Листинг Б.3. MonthConstants.java

```
  1 /* ===========================================================================
  2  * JCommon : библиотека классов общего назначения для платформы Java(tm)
  3  * ===========================================================================
  4  *
  5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
  6  *
  7  * Информация о проекте:  http://www.jfree.org/jcommon/index.html
  8  *
  9  * Библиотека распространяется бесплатно; вы можете свободно распространять
 10  * и/или изменять ее на условиях лицензии Lesser General Public License
 11  * в формулировке Free Software Foundation; либо версии 2.1 лицензии, либо
 12  * (на ваше усмотрение) любой последующей версии.
 13  *
 14  * Библиотека распространяется в надежде, что она будет полезна, но
 15  * БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ, даже без подразумеваемой гарантии ПРИГОДНОСТИ
 16  * для КОНКРЕТНОЙ ЦЕЛИ. За подробностями обращайтесь к GNU Lesser General
 17  * Public License.
 18  *
 19  * Вы должны получить копию лицензии GNU Lesser General Public License
 20  * с этой библиотекой; если этого не произошло, обратитесь в Free Software
 21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 22  * USA.
 23  *
 24  * [Java является зарегистрированной торговой маркой Sun Microsystems, Inc.
 25  * в Соединенных Штатах и других странах].
 26  *
 27  * ------------------
 28  * MonthConstants.java
```

продолжение ⇩

Листинг Б.3 (продолжение)

```
 * -------------------
 * (C) Copyright 2002, 2003, by Object Refinery Limited.
 *
 * Автор:  Дэвид Гилберт (для Object Refinery Limited);
 * Участники:   -;
 *
 * $Id: MonthConstants.java,v 1.4 2005/11/16 15:58:40 taqua Exp $
 *
 * Изменения
 * -------
 * 29.05.2002 : Версия 1 (код перемещен из класса SerialDate) (DG);
 *
 */

package org.jfree.date;

/**
 * Константы месяцев. Обратите внимание: константы НЕ ЭКИВАЛЕНТНЫ определяемым
 * в java.util.Calendar (где JANUARY=0, а DECEMBER=11).
 * <P>
 * Используются классами SerialDate и RegularTimePeriod.
 *
 * @author Дэвид Гилберт
 */
public interface MonthConstants {

    /** Константа для января. */
    public static final int JANUARY = 1;

    /** Константа для февраля. */
    public static final int FEBRUARY = 2;

    /** Константа для мая. */
    public static final int MARCH = 3;

    /** Константа для апреля. */
    public static final int APRIL = 4;

    /** Константа для мая. */
    public static final int MAY = 5;

    /** Константа для июня. */
    public static final int JUNE = 6;

    /** Константа для июля. */
    public static final int JULY = 7;

    /** Константа для августа. */
    public static final int AUGUST = 8;

```

```
79     /** Константа для сентября. */
80     public static final int SEPTEMBER = 9;
81
82     /** Константа для октября. */
83     public static final int OCTOBER = 10;
84
85     /** Константа для ноября. */
86     public static final int NOVEMBER = 11;
87
88     /** Константа для декабря. */
89     public static final int DECEMBER = 12;
90
91 }
```

Листинг Б.4. BobsSerialDateTest.java

```
 1 package org.jfree.date.junit;
 2
 3 import junit.framework.TestCase;
 4 import org.jfree.date.*;
 5 import static org.jfree.date.SerialDate.*;
 6
 7 import java.util.*;
 8
 9 public class BobsSerialDateTest extends TestCase {
10
11   public void testIsValidWeekdayCode() throws Exception {
12     for (int day = 1; day <= 7; day++)
13       assertTrue(isValidWeekdayCode(day));
14     assertFalse(isValidWeekdayCode(0));
15     assertFalse(isValidWeekdayCode(8));
16   }
17
18   public void testStringToWeekdayCode() throws Exception {
19
20     assertEquals(-1, stringToWeekdayCode("Hello"));
21     assertEquals(MONDAY, stringToWeekdayCode("Monday"));
22     assertEquals(MONDAY, stringToWeekdayCode("Mon"));
23 //todo    assertEquals(MONDAY,stringToWeekdayCode("monday"));
24 //    assertEquals(MONDAY,stringToWeekdayCode("MONDAY"));
25 //    assertEquals(MONDAY, stringToWeekdayCode("mon"));
26
27     assertEquals(TUESDAY, stringToWeekdayCode("Tuesday"));
28     assertEquals(TUESDAY, stringToWeekdayCode("Tue"));
29 //    assertEquals(TUESDAY,stringToWeekdayCode("tuesday"));
30 //    assertEquals(TUESDAY,stringToWeekdayCode("TUESDAY"));
31 //    assertEquals(TUESDAY, stringToWeekdayCode("tue"));
32 //    assertEquals(TUESDAY, stringToWeekdayCode("tues"));
33
34     assertEquals(WEDNESDAY, stringToWeekdayCode("Wednesday"));
35     assertEquals(WEDNESDAY, stringToWeekdayCode("Wed"));
36 //    assertEquals(WEDNESDAY,stringToWeekdayCode("wednesday"));
```

продолжение ⇩

Листинг Б.4 (продолжение)

```
37 //    assertEquals(WEDNESDAY,stringToWeekdayCode("WEDNESDAY"));
38 //    assertEquals(WEDNESDAY, stringToWeekdayCode("wed"));
39
40     assertEquals(THURSDAY, stringToWeekdayCode("Thursday"));
41     assertEquals(THURSDAY, stringToWeekdayCode("Thu"));
42 //    assertEquals(THURSDAY,stringToWeekdayCode("thursday"));
43 //    assertEquals(THURSDAY,stringToWeekdayCode("THURSDAY"));
44 //    assertEquals(THURSDAY, stringToWeekdayCode("thu"));
45 //    assertEquals(THURSDAY, stringToWeekdayCode("thurs"));
46
47     assertEquals(FRIDAY, stringToWeekdayCode("Friday"));
48     assertEquals(FRIDAY, stringToWeekdayCode("Fri"));
49 //    assertEquals(FRIDAY,stringToWeekdayCode("friday"));
50 //    assertEquals(FRIDAY,stringToWeekdayCode("FRIDAY"));
51 //    assertEquals(FRIDAY, stringToWeekdayCode("fri"));
52
53     assertEquals(SATURDAY, stringToWeekdayCode("Saturday"));
54     assertEquals(SATURDAY, stringToWeekdayCode("Sat"));
55 //    assertEquals(SATURDAY,stringToWeekdayCode("saturday"));
56 //    assertEquals(SATURDAY,stringToWeekdayCode("SATURDAY"));
57 //    assertEquals(SATURDAY, stringToWeekdayCode("sat"));
58
59     assertEquals(SUNDAY, stringToWeekdayCode("Sunday"));
60     assertEquals(SUNDAY, stringToWeekdayCode("Sun"));
61 //    assertEquals(SUNDAY,stringToWeekdayCode("sunday"));
62 //    assertEquals(SUNDAY,stringToWeekdayCode("SUNDAY"));
63 //    assertEquals(SUNDAY, stringToWeekdayCode("sun"));
64   }
65
66   public void testWeekdayCodeToString() throws Exception {
67     assertEquals("Sunday", weekdayCodeToString(SUNDAY));
68     assertEquals("Monday", weekdayCodeToString(MONDAY));
69     assertEquals("Tuesday", weekdayCodeToString(TUESDAY));
70     assertEquals("Wednesday", weekdayCodeToString(WEDNESDAY));
71     assertEquals("Thursday", weekdayCodeToString(THURSDAY));
72     assertEquals("Friday", weekdayCodeToString(FRIDAY));
73     assertEquals("Saturday", weekdayCodeToString(SATURDAY));
74   }
75
76   public void testIsValidMonthCode() throws Exception {
77     for (int i = 1; i <= 12; i++)
78       assertTrue(isValidMonthCode(i));
79     assertFalse(isValidMonthCode(0));
80     assertFalse(isValidMonthCode(13));
81   }
82
83   public void testMonthToQuarter() throws Exception {
84     assertEquals(1, monthCodeToQuarter(JANUARY));
85     assertEquals(1, monthCodeToQuarter(FEBRUARY));
86     assertEquals(1, monthCodeToQuarter(MARCH));
```

```
 87     assertEquals(2, monthCodeToQuarter(APRIL));
 88     assertEquals(2, monthCodeToQuarter(MAY));
 89     assertEquals(2, monthCodeToQuarter(JUNE));
 90     assertEquals(3, monthCodeToQuarter(JULY));
 91     assertEquals(3, monthCodeToQuarter(AUGUST));
 92     assertEquals(3, monthCodeToQuarter(SEPTEMBER));
 93     assertEquals(4, monthCodeToQuarter(OCTOBER));
 94     assertEquals(4, monthCodeToQuarter(NOVEMBER));
 95     assertEquals(4, monthCodeToQuarter(DECEMBER));
 96
 97     try {
 98       monthCodeToQuarter(-1);
 99       fail("Invalid Month Code should throw exception");
100     } catch (IllegalArgumentException e) {
101     }
102   }
103
104   public void testMonthCodeToString() throws Exception {
105     assertEquals("January", monthCodeToString(JANUARY));
106     assertEquals("February", monthCodeToString(FEBRUARY));
107     assertEquals("March", monthCodeToString(MARCH));
108     assertEquals("April", monthCodeToString(APRIL));
109     assertEquals("May", monthCodeToString(MAY));
110     assertEquals("June", monthCodeToString(JUNE));
111     assertEquals("July", monthCodeToString(JULY));
112     assertEquals("August", monthCodeToString(AUGUST));
113     assertEquals("September", monthCodeToString(SEPTEMBER));
114     assertEquals("October", monthCodeToString(OCTOBER));
115     assertEquals("November", monthCodeToString(NOVEMBER));
116     assertEquals("December", monthCodeToString(DECEMBER));
117
118     assertEquals("Jan", monthCodeToString(JANUARY, true));
119     assertEquals("Feb", monthCodeToString(FEBRUARY, true));
120     assertEquals("Mar", monthCodeToString(MARCH, true));
121     assertEquals("Apr", monthCodeToString(APRIL, true));
122     assertEquals("May", monthCodeToString(MAY, true));
123     assertEquals("Jun", monthCodeToString(JUNE, true));
124     assertEquals("Jul", monthCodeToString(JULY, true));
125     assertEquals("Aug", monthCodeToString(AUGUST, true));
126     assertEquals("Sep", monthCodeToString(SEPTEMBER, true));
127     assertEquals("Oct", monthCodeToString(OCTOBER, true));
128     assertEquals("Nov", monthCodeToString(NOVEMBER, true));
129     assertEquals("Dec", monthCodeToString(DECEMBER, true));
130
131     try {
132       monthCodeToString(-1);
133       fail("Invalid month code should throw exception");
134     } catch (IllegalArgumentException e) {
135     }
136
137   }
```

продолжение ⇩

Листинг Б.4 (продолжение)

```
138
139   public void testStringToMonthCode() throws Exception {
140     assertEquals(JANUARY,stringToMonthCode("1"));
141     assertEquals(FEBRUARY,stringToMonthCode("2"));
142     assertEquals(MARCH,stringToMonthCode("3"));
143     assertEquals(APRIL,stringToMonthCode("4"));
144     assertEquals(MAY,stringToMonthCode("5"));
145     assertEquals(JUNE,stringToMonthCode("6"));
146     assertEquals(JULY,stringToMonthCode("7"));
147     assertEquals(AUGUST,stringToMonthCode("8"));
148     assertEquals(SEPTEMBER,stringToMonthCode("9"));
149     assertEquals(OCTOBER,stringToMonthCode("10"));
150     assertEquals(NOVEMBER, stringToMonthCode("11"));
151     assertEquals(DECEMBER,stringToMonthCode("12"));
152
153 //todo    assertEquals(-1, stringToMonthCode("0"));
154 //    assertEquals(-1, stringToMonthCode("13"));
155
156     assertEquals(-1,stringToMonthCode("Hello"));
157
158     for (int m = 1; m <= 12; m++) {
159       assertEquals(m, stringToMonthCode(monthCodeToString(m, false)));
160       assertEquals(m, stringToMonthCode(monthCodeToString(m, true)));
161     }
162
163 //    assertEquals(1,stringToMonthCode("jan"));
164 //    assertEquals(2,stringToMonthCode("feb"));
165 //    assertEquals(3,stringToMonthCode("mar"));
166 //    assertEquals(4,stringToMonthCode("apr"));
167 //    assertEquals(5,stringToMonthCode("may"));
168 //    assertEquals(6,stringToMonthCode("jun"));
169 //    assertEquals(7,stringToMonthCode("jul"));
170 //    assertEquals(8,stringToMonthCode("aug"));
171 //    assertEquals(9,stringToMonthCode("sep"));
172 //    assertEquals(10,stringToMonthCode("oct"));
173 //    assertEquals(11,stringToMonthCode("nov"));
174 //    assertEquals(12,stringToMonthCode("dec"));
175
176 //    assertEquals(1,stringToMonthCode("JAN"));
177 //    assertEquals(2,stringToMonthCode("FEB"));
178 //    assertEquals(3,stringToMonthCode("MAR"));
179 //    assertEquals(4,stringToMonthCode("APR"));
180 //    assertEquals(5,stringToMonthCode("MAY"));
181 //    assertEquals(6,stringToMonthCode("JUN"));
182 //    assertEquals(7,stringToMonthCode("JUL"));
183 //    assertEquals(8,stringToMonthCode("AUG"));
184 //    assertEquals(9,stringToMonthCode("SEP"));
185 //    assertEquals(10,stringToMonthCode("OCT"));
186 //    assertEquals(11,stringToMonthCode("NOV"));
187 //    assertEquals(12,stringToMonthCode("DEC"));
```

```
188
189 //    assertEquals(1,stringToMonthCode("january"));
190 //    assertEquals(2,stringToMonthCode("february"));
191 //    assertEquals(3,stringToMonthCode("march"));
192 //    assertEquals(4,stringToMonthCode("april"));
193 //    assertEquals(5,stringToMonthCode("may"));
194 //    assertEquals(6,stringToMonthCode("june"));
195 //    assertEquals(7,stringToMonthCode("july"));
196 //    assertEquals(8,stringToMonthCode("august"));
197 //    assertEquals(9,stringToMonthCode("september"));
198 //    assertEquals(10,stringToMonthCode("october"));
199 //    assertEquals(11,stringToMonthCode("november"));
200 //    assertEquals(12,stringToMonthCode("december"));
201
202 //    assertEquals(1,stringToMonthCode("JANUARY"));
203 //    assertEquals(2,stringToMonthCode("FEBRUARY"));
204 //    assertEquals(3,stringToMonthCode("MAR"));
205 //    assertEquals(4,stringToMonthCode("APRIL"));
206 //    assertEquals(5,stringToMonthCode("MAY"));
207 //    assertEquals(6,stringToMonthCode("JUNE"));
208 //    assertEquals(7,stringToMonthCode("JULY"));
209 //    assertEquals(8,stringToMonthCode("AUGUST"));
210 //    assertEquals(9,stringToMonthCode("SEPTEMBER"));
211 //    assertEquals(10,stringToMonthCode("OCTOBER"));
212 //    assertEquals(11,stringToMonthCode("NOVEMBER"));
213 //    assertEquals(12,stringToMonthCode("DECEMBER"));
214   }
215
216   public void testIsValidWeekInMonthCode() throws Exception {
217     for (int w = 0; w <= 4; w++) {
218       assertTrue(isValidWeekInMonthCode(w));
219     }
220     assertFalse(isValidWeekInMonthCode(5));
221   }
222
223   public void testIsLeapYear() throws Exception {
224     assertFalse(isLeapYear(1900));
225     assertFalse(isLeapYear(1901));
226     assertFalse(isLeapYear(1902));
227     assertFalse(isLeapYear(1903));
228     assertTrue(isLeapYear(1904));
229     assertTrue(isLeapYear(1908));
230     assertFalse(isLeapYear(1955));
231     assertTrue(isLeapYear(1964));
232     assertTrue(isLeapYear(1980));
233     assertTrue(isLeapYear(2000));
234     assertFalse(isLeapYear(2001));
235     assertFalse(isLeapYear(2100));
236   }
237
238   public void testLeapYearCount() throws Exception {
```

продолжение ⇩

Листинг Б.4 (продолжение)

```
239     assertEquals(0, leapYearCount(1900));
240     assertEquals(0, leapYearCount(1901));
241     assertEquals(0, leapYearCount(1902));
242     assertEquals(0, leapYearCount(1903));
243     assertEquals(1, leapYearCount(1904));
244     assertEquals(1, leapYearCount(1905));
245     assertEquals(1, leapYearCount(1906));
246     assertEquals(1, leapYearCount(1907));
247     assertEquals(2, leapYearCount(1908));
248     assertEquals(24, leapYearCount(1999));
249     assertEquals(25, leapYearCount(2001));
250     assertEquals(49, leapYearCount(2101));
251     assertEquals(73, leapYearCount(2201));
252     assertEquals(97, leapYearCount(2301));
253     assertEquals(122, leapYearCount(2401));
254   }
255
256   public void testLastDayOfMonth() throws Exception {
257     assertEquals(31, lastDayOfMonth(JANUARY, 1901));
258     assertEquals(28, lastDayOfMonth(FEBRUARY, 1901));
259     assertEquals(31, lastDayOfMonth(MARCH, 1901));
260     assertEquals(30, lastDayOfMonth(APRIL, 1901));
261     assertEquals(31, lastDayOfMonth(MAY, 1901));
262     assertEquals(30, lastDayOfMonth(JUNE, 1901));
263     assertEquals(31, lastDayOfMonth(JULY, 1901));
264     assertEquals(31, lastDayOfMonth(AUGUST, 1901));
265     assertEquals(30, lastDayOfMonth(SEPTEMBER, 1901));
266     assertEquals(31, lastDayOfMonth(OCTOBER, 1901));
267     assertEquals(30, lastDayOfMonth(NOVEMBER, 1901));
268     assertEquals(31, lastDayOfMonth(DECEMBER, 1901));
269     assertEquals(29, lastDayOfMonth(FEBRUARY, 1904));
270   }
271
272   public void testAddDays() throws Exception {
273     SerialDate newYears = d(1, JANUARY, 1900);
274     assertEquals(d(2, JANUARY, 1900), addDays(1, newYears));
275     assertEquals(d(1, FEBRUARY, 1900), addDays(31, newYears));
276     assertEquals(d(1, JANUARY, 1901), addDays(365, newYears));
277     assertEquals(d(31, DECEMBER, 1904), addDays(5 * 365, newYears));
278   }
279
280   private static SpreadsheetDate d(int day, int month, int year) {return new
                                                    SpreadsheetDate(day, month, year);}
281
282   public void testAddMonths() throws Exception {
283     assertEquals(d(1, FEBRUARY, 1900), addMonths(1, d(1, JANUARY, 1900)));
284     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(31, JANUARY, 1900)));
285     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(30, JANUARY, 1900)));
286     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(29, JANUARY, 1900)));
287     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(28, JANUARY, 1900)));
288     assertEquals(d(27, FEBRUARY, 1900), addMonths(1, d(27, JANUARY, 1900)));
289
290     assertEquals(d(30, JUNE, 1900), addMonths(5, d(31, JANUARY, 1900)));
```

```
291     assertEquals(d(30, JUNE, 1901), addMonths(17, d(31, JANUARY, 1900)));
292
293     assertEquals(d(29, FEBRUARY, 1904), addMonths(49, d(31, JANUARY, 1900)));
294
295   }
296
297   public void testAddYears() throws Exception {
298     assertEquals(d(1, JANUARY, 1901), addYears(1, d(1, JANUARY, 1900)));
299     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(29, FEBRUARY, 1904)));
300     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(28, FEBRUARY, 1904)));
301     assertEquals(d(28, FEBRUARY, 1904), addYears(1, d(28, FEBRUARY, 1903)));
302   }
303
304   public void testGetPreviousDayOfWeek() throws Exception {
305     assertEquals(d(24, FEBRUARY, 2006), getPreviousDayOfWeek(FRIDAY, d(1, MARCH, 2006)));
306     assertEquals(d(22, FEBRUARY, 2006), getPreviousDayOfWeek(WEDNESDAY, d(1, MARCH,
        2006)));
307     assertEquals(d(29, FEBRUARY, 2004), getPreviousDayOfWeek(SUNDAY, d(3, MARCH, 2004)));
308     assertEquals(d(29, DECEMBER, 2004), getPreviousDayOfWeek(WEDNESDAY, d(5, JANUARY,
        2005)));
309
310     try {
311       getPreviousDayOfWeek(-1, d(1, JANUARY, 2006));
312       fail("Invalid day of week code should throw exception");
313     } catch (IllegalArgumentException e) {
314     }
315   }
316
317   public void testGetFollowingDayOfWeek() throws Exception {
318 //    assertEquals(d(1, JANUARY, 2005),getFollowingDayOfWeek(SATURDAY, d(25,
    DECEMBER, 2004)));
319     assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(26, DECEMBER,
        2004)));
320     assertEquals(d(3, MARCH, 2004), getFollowingDayOfWeek(WEDNESDAY, d(28, FEBRUARY,
        2004)));
321
322     try {
323       getFollowingDayOfWeek(-1, d(1, JANUARY, 2006));
324       fail("Invalid day of week code should throw exception");
325     } catch (IllegalArgumentException e) {
326     }
327   }
328
329   public void testGetNearestDayOfWeek() throws Exception {
330     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(16, APRIL, 2006)));
331     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(17, APRIL, 2006)));
332     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(18, APRIL, 2006)));
333     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(19, APRIL, 2006)));
334     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(20, APRIL, 2006)));
335     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(21, APRIL, 2006)));
336     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(22, APRIL, 2006)));
337
338 //todo    assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(16, APRIL,
    2006)));
```

продолжение ⇩

Листинг Б.4. (продолжение)

```
339     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(17, APRIL, 2006)));
340     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(18, APRIL, 2006)));
341     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(19, APRIL, 2006)));
342     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(20, APRIL, 2006)));
343     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(21, APRIL, 2006)));
344     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(22, APRIL, 2006)));
345
346 //    assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(16, APRIL, 2006)));
347 //    assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(17, APRIL, 2006)));
348     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(18, APRIL, 2006)));
349     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(19, APRIL, 2006)));
350     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(20, APRIL, 2006)));
351     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(21, APRIL, 2006)));
352     assertEquals(d(25, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(22, APRIL, 2006)));
353
354 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(16, APRIL, 2006)));
355 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(17, APRIL, 2006)));
356 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(18, APRIL, 2006)));
357     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(19, APRIL, 02006)));
358     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(20, APRIL, 2006)));
359     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(21, APRIL, 2006)));
360     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(22, APRIL, 2006)));
361
362 //    assertEquals(d(13, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(16, APRIL, 2006)));
363 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(17, APRIL, 2006)));
364 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(18, APRIL, 2006)));
365 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(19, APRIL, 2006)));
366     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(20, APRIL, 2006)));
367     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(21, APRIL, 2006)));
368     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(22, APRIL, 2006)));
369
370 //    assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(16, APRIL, 2006)));
371 //    assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(17, APRIL, 2006)));
372 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(18, APRIL, 2006)));
373 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(19, APRIL, 2006)));
374 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(20, APRIL, 2006)));
375     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(21, APRIL, 2006)));
376     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(22, APRIL, 2006)));
377
378 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(16, APRIL, 2006)));
379 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(17, APRIL, 2006)));
380 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(18, APRIL, 2006)));
381 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(19, APRIL, 2006)));
382 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(20, APRIL, 2006)));
383 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(21, APRIL, 2006)));
384     assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(22, APRIL, 2006)));
385
386     try {
387       getNearestDayOfWeek(-1, d(1, JANUARY, 2006));
388       fail("Invalid day of week code should throw exception");
389     } catch (IllegalArgumentException e) {
390     }
391   }
```

```
392
393   public void testEndOfCurrentMonth() throws Exception {
394     SerialDate d = SerialDate.createInstance(2);
395     assertEquals(d(31, JANUARY, 2006), d.getEndOfCurrentMonth(d(1, JANUARY, 2006)));
396     assertEquals(d(28, FEBRUARY, 2006), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2006)));
397     assertEquals(d(31, MARCH, 2006), d.getEndOfCurrentMonth(d(1, MARCH, 2006)));
398     assertEquals(d(30, APRIL, 2006), d.getEndOfCurrentMonth(d(1, APRIL, 2006)));
399     assertEquals(d(31, MAY, 2006), d.getEndOfCurrentMonth(d(1, MAY, 2006)));
400     assertEquals(d(30, JUNE, 2006), d.getEndOfCurrentMonth(d(1, JUNE, 2006)));
401     assertEquals(d(31, JULY, 2006), d.getEndOfCurrentMonth(d(1, JULY, 2006)));
402     assertEquals(d(31, AUGUST, 2006), d.getEndOfCurrentMonth(d(1, AUGUST, 2006)));
403     assertEquals(d(30, SEPTEMBER, 2006), d.getEndOfCurrentMonth(d(1, SEPTEMBER, 2006)));
404     assertEquals(d(31, OCTOBER, 2006), d.getEndOfCurrentMonth(d(1, OCTOBER, 2006)));
405     assertEquals(d(30, NOVEMBER, 2006), d.getEndOfCurrentMonth(d(1, NOVEMBER, 2006)));
406     assertEquals(d(31, DECEMBER, 2006), d.getEndOfCurrentMonth(d(1, DECEMBER, 2006)));
407     assertEquals(d(29, FEBRUARY, 2008), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2008)));
408   }
409
410   public void testWeekInMonthToString() throws Exception {
411     assertEquals("First",weekInMonthToString(FIRST_WEEK_IN_MONTH));
412     assertEquals("Second",weekInMonthToString(SECOND_WEEK_IN_MONTH));
413     assertEquals("Third",weekInMonthToString(THIRD_WEEK_IN_MONTH));
414     assertEquals("Fourth",weekInMonthToString(FOURTH_WEEK_IN_MONTH));
415     assertEquals("Last",weekInMonthToString(LAST_WEEK_IN_MONTH));
416
417 //todo    try {
418 //      weekInMonthToString(-1);
419 //      fail("Invalid week code should throw exception");
420 //    } catch (IllegalArgumentException e) {
421 //    }
422   }
423
424   public void testRelativeToString() throws Exception {
425     assertEquals("Preceding",relativeToString(PRECEDING));
426     assertEquals("Nearest",relativeToString(NEAREST));
427     assertEquals("Following",relativeToString(FOLLOWING));
428
429 //todo    try {
430 //      relativeToString(-1000);
431 //      fail("Invalid relative code should throw exception");
432 //    } catch (IllegalArgumentException e) {
433 //    }
434   }
435
436   public void testCreateInstanceFromDDMMYYY() throws Exception {
437     SerialDate date = createInstance(1, JANUARY, 1900);
438     assertEquals(1,date.getDayOfMonth());
439     assertEquals(JANUARY,date.getMonth());
440     assertEquals(1900,date.getYYYY());
441     assertEquals(2,date.toSerial());
442   }
443
444   public void testCreateInstanceFromSerial() throws Exception {
445     assertEquals(d(1, JANUARY, 1900),createInstance(2));
```

продолжение ⇩

Листинг Б.4. (продолжение)

```
446     assertEquals(d(1, JANUARY, 1901), createInstance(367));
447   }
448
449   public void testCreateInstanceFromJavaDate() throws Exception {
450     assertEquals(d(1, JANUARY, 1900),
                    createInstance(new GregorianCalendar(1900,0,1).getTime()));
451     assertEquals(d(1, JANUARY, 2006),
                    createInstance(new GregorianCalendar(2006,0,1).getTime()));
452   }
453
454   public static void main(String[] args) {
455     junit.textui.TestRunner.run(BobsSerialDateTest.class);
456   }
457 }
```

Листинг Б.5. SpreadsheetDate.java

```
 1 /* ===========================================================================
 2  * JCommon : библиотека классов общего назначения для платформы Java(tm)
 3  * ===========================================================================
 4  *
 5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 6  *
 7  * Информация о проекте:  http://www.jfree.org/jcommon/index.html
 8  *
 9  * Библиотека распространяется бесплатно; вы можете свободно распространять
10  * и/или изменять ее на условиях лицензии Lesser General Public License
11  * в формулировке Free Software Foundation; либо версии 2.1 лицензии, либо
12  * (на ваше усмотрение) любой последующей версии.
13  *
14  * Библиотека распространяется в надежде, что она будет полезна, но
15  * БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ, даже без подразумеваемой гарантии ПРИГОДНОСТИ
16  * для КОНКРЕТНОЙ ЦЕЛИ. За подробностями обращайтесь к GNU Lesser General
17  * Public License.
18  *
19  * Вы должны получить копию лицензии GNU Lesser General Public License
20  * с этой библиотекой; если этого не произошло, обратитесь в Free Software
21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
22  * USA.
23  *
24  * [Java является зарегистрированной торговой маркой Sun Microsystems, Inc.
25  * в Соединенных Штатах и других странах].
26  *
27  * --------------------
28  * SpreadsheetDate.java
29  * --------------------
30  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
31  *
32  * Автор:  Дэвид Гилберт (для Object Refinery Limited);
33  * Участники:   -;
34  *
```

```
35  * $Id: SpreadsheetDate.java,v 1.8 2005/11/03 09:25:39 mungady Exp $
36  *
37  * Изменения
38  * -------
39  * 11.10.2001 : Версия 1 (DG);
40  * 05.11.2001 : Добавлены методы getDescription() и setDescription() (DG);
41  * 12.11.2001 : Переименование ExcelDate.java в SpreadsheetDate.java (DG);
42  *              Исправлена ошибка в вычислении дня, месяца и года
43  *              по порядковому номеру (DG);
44  * 24.01.2002 : Исправлена ошибка в вычислении порядкового номера по дню,
45  *              месяцу и году. Спасибо Тревору Хиллзу за сообщение(DG);
46  * 29.05.2002 : Добавлен метод equals(Object) (SourceForge ID 558850) (DG);
47  * 03.10.2002 : Исправлены ошибки по информации Checkstyle (DG);
48  * 13.03.2003 : Реализован интерфейс Serializable (DG);
49  * 04.09.2003 : Завершены методы isInRange() (DG);
50  * 05.09.2003 : Реализован интерфейс Comparable (DG);
51  * 21.10.2003 : Добавлен метод hashCode() (DG);
52  *
53  */
54
55 package org.jfree.date;
56
57 import java.util.Calendar;
58 import java.util.Date;
59
60 /**
61  * Представляет дату с использованием целого числа, по аналогии с реализацией
62  * в Microsoft Excel.  Поддерживаемый диапазон дат:
63  * с 1 января 1900 по 31 декабря 9999.
64  * <P>
65  * Учтите, что в Excel существует намеренная ошибка, вследствие которой год
66  * 1900 считается високосным, тогда как в действительности он таковым не является.
67  * Дополнительная информация приведена на сайте Microsoft в статье Q181370:
68  * <P>
69  * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
70  * <P>
71  * Excel считает, что 1 января 1900 = 1.  Этот класс считает, что
72  * 1 января 1900 = 2.
73  * В результате номер дня этого класса будет отличаться от номера Excel
74  * в январе и феврале 1900...но затем Excel прибавляет лишний день
75  * (29 февраля 1900, который в действительности не существует!), и с этого
76  *  момента нумерация дней совпадает.
77  *
78  * @author Дэвид Гилберт
79  */
80 public class SpreadsheetDate extends SerialDate {
81
82     /** Для сериализации. */
83     private static final long serialVersionUID = -2039586705374454461L;
84
85     /**
```

продолжение ⇘

Листинг Б.5 (продолжение)

```
 86      * Номер дня (1.01.1900 = 2, 2.01.1900 = 3, ..., 31.12.9999 =
 87      * 2958465).
 88      */
 89     private int serial;
 90
 91     /** День месяца (от 1 до 28, 29, 30 или 31 в зависимости от месяца). */
 92     private int day;
 93
 94     /** Месяц года (от 1 по 12). */
 95     private int month;
 96
 97     /** Год (от 1900 до 9999). */
 98     private int year;
 99
100     /** Необязательное описание даты. */
101     private String description;
102
103     /**
104      * Создает новый экземпляр даты.
105      *
106      * @param day  день (в диапазоне от 1 до 28/29/30/31).
107      * @param month  месяц (в диапазоне от 1 до 12).
108      * @param year  год (в диапазоне от 1900 до 9999).
109      */
110     public SpreadsheetDate(final int day, final int month, final int year) {
111
112         if ((year >= 1900) && (year <= 9999)) {
113             this.year = year;
114         }
115         else {
116             throw new IllegalArgumentException(
117                 "The 'year' argument must be in range 1900 to 9999."
118             );
119         }
120
121         if ((month >= MonthConstants.JANUARY)
122                 && (month <= MonthConstants.DECEMBER)) {
123             this.month = month;
124         }
125         else {
126             throw new IllegalArgumentException(
127                 "The 'month' argument must be in the range 1 to 12."
128             );
129         }
130
131         if ((day >= 1) && (day <= SerialDate.lastDayOfMonth(month, year))) {
132             this.day = day;
133         }
134         else {
135             throw new IllegalArgumentException("Invalid 'day' argument.");
```

```
136         }
137
138         // Порядковый номер должен синхронизироваться с днем-месяцем-годом...
139         this.serial = calcSerial(day, month, year);
140
141         this.description = null;
142
143     }
144
145     /**
146      * Стандартный конструктор - создает новый объект даты, представляющий
147      * день с заданным номером (в диапазоне от 2 до 2958465).
148      *
149      * @param serial  порядковый номер дня (диапазон: от 2 до 2958465).
150      */
151     public SpreadsheetDate(final int serial) {
152
153         if ((serial >= SERIAL_LOWER_BOUND) && (serial <= SERIAL_UPPER_BOUND)) {
154             this.serial = serial;
155         }
156         else {
157             throw new IllegalArgumentException(
158                 "SpreadsheetDate: Serial must be in range 2 to 2958465.");
159         }
160
161         // День-месяц-год должен синхронизироваться с порядковым номером...
162         calcDayMonthYear();
163
164     }
165
166     /**
167      * Возвращает описание, присоединенное к дате.
168      * Дата не обязана иметь описание, но в некоторых приложениях
169      * оно может оказаться полезным.
170      *
171      * @return описание, присоединенное к дате.
172      */
173     public String getDescription() {
174         return this.description;
175     }
176
177     /**
178      * Задает описание для даты.
179      *
180      * @param description  описание даты (разрешается
181      *                     <code>null</code>).
182      */
183     public void setDescription(final String description) {
184         this.description = description;
185     }
186
```

продолжение ⇩

Листинг Б.5 (продолжение)

```
187     /**
188      * Возвращает порядковый номер даты, где 1 января 1900 = 2
189      * (что почти соответствует системе нумерации, используемой в Microsoft
190      * Excel for Windows и Lotus 1-2-3).
191      *
192      * @return порядковый номер даты.
193      */
194     public int toSerial() {
195         return this.serial;
196     }
197
198     /**
199      * Возвращает объект <code>java.util.Date</code>, эквивалентный текущей дате.
200      *
201      * @return объект даты.
202      */
203     public Date toDate() {
204         final Calendar calendar = Calendar.getInstance();
205         calendar.set(getYYYY(), getMonth() - 1, getDayOfMonth(), 0, 0, 0);
206         return calendar.getTime();
207     }
208
209     /**
210      * Возвращает год (из действительного диапазона от 1900 до 9999).
211      *
212      * @return год.
213      */
214     public int getYYYY() {
215         return this.year;
216     }
217
218     /**
219      * Возвращает месяц (январь = 1, февраль = 2, март = 3).
220      *
221      * @return месяц года.
222      */
223     public int getMonth() {
224         return this.month;
225     }
226
227     /**
228      * Возвращает день месяца.
229      *
230      * @return день месяца.
231      */
232     public int getDayOfMonth() {
233         return this.day;
234     }
235
236     /**
```

```
237      * Возвращает код, представляющий день недели.
238      * <P>
239      * Коды определяются в классе {@link SerialDate} следующим образом:
240      * <code>SUNDAY</code>, <code>MONDAY</code>, <code>TUESDAY</code>,
241      * <code>WEDNESDAY</code>, <code>THURSDAY</code>, <code>FRIDAY</code> и
242      * <code>SATURDAY</code>.
243      *
244      * @return Код, представляющий день недели.
245      */
246     public int getDayOfWeek() {
247         return (this.serial + 6) % 7 + 1;
248     }
249
250     /**
251      * Проверяет равенство текущей даты с другим произвольным объектом.
252      * <P>
253      * Метод возвращает true ТОЛЬКО в том случае, если объект является
254      * экземпляром базового класса {@link SerialDate} и представляет тот же
255      * день, что и {@link SpreadsheetDate}.
256      *
257      * @param object  объект для сравнения (допускается <code>null</code>).
258      *
259      * @return Логический признак.
260      */
261     public boolean equals(final Object object) {
262
263         if (object instanceof SerialDate) {
264             final SerialDate s = (SerialDate) object;
265             return (s.toSerial() == this.toSerial());
266         }
267         else {
268             return false;
269         }
270
271     }
272
273     /**
274      * Возвращает хеш-код для экземпляра класса.
275      *
276      * @return хеш-код.
277      */
278     public int hashCode() {
279         return toSerial();
280     }
281
282     /**
283      * Возвращает разность (в днях) между текущей и заданной
284      * 'другой' датой.
285      *
286      * @param other  дата для сравнения.
287      *
```

продолжение ⇩

Листинг Б.5 (продолжение)

```
288      * @return разность (в днях) между текущий и заданной
289      *         'другой' датой.
290      */
291     public int compare(final SerialDate other) {
292         return this.serial - other.toSerial();
293     }
294
295     /**
296      * Реализует метод, необходимый для интерфейса Comparable.
297      *
298      * @param other  другой объект (обычно другой объект SerialDate).
299      *
300      * @return отрицательное целое, нуль или положительное целое число,
301      *         если объект меньше, равен или больше заданного объекта.
302      */
303     public int compareTo(final Object other) {
304         return compare((SerialDate) other);
305     }
306
307     /**
308      * Возвращает true, если текущий объект SerialDate представляет ту же дату,
309      * что и заданный объект SerialDate.
310      *
311      * @param other  дата для сравнения.
312      *
313      * @return <code>true</code>, если текущий объект SerialDate представляет
314      *         ту же дату, что и заданный объект SerialDate.
315      */
316     public boolean isOn(final SerialDate other) {
317         return (this.serial == other.toSerial());
318     }
319
320     /**
321      * Возвращает true, если текущий объект SerialDate представляет более раннюю
322      * дату по сравнению с заданным объектом SerialDate.
323      *
324      * @param other  дата для сравнения.
325      *
326      * @return <code>true</code>, если текущий объект SerialDate представляет
327      *         более раннюю дату по сравнению с заданным объектом SerialDate.
328      */
329     public boolean isBefore(final SerialDate other) {
330         return (this.serial < other.toSerial());
331     }
332
333     /**
334      * Возвращает true, если текущий объект SerialDate представляет ту же дату,
335      * что и заданный объект SerialDate.
336      *
337      * @param other  дата для сравнения.
```

```
338      *
339      * @return <code>true</code>, если текущий объект SerialDate представляет
340      *         ту же дату, что и заданный объект SerialDate.
341      */
342     public boolean isOnOrBefore(final SerialDate other) {
343         return (this.serial <= other.toSerial());
344     }
345
346     /**
347      * Возвращает true, если текущий объект SerialDate представляет ту же дату,
348      * что и заданный объект SerialDate.
349      *
350      * @param other  дата для сравнения.
351      *
352      * @return <code>true</code>, если текущий объект SerialDate представляет
353      *         ту же дату, что и заданный объект SerialDate.
354      */
355     public boolean isAfter(final SerialDate other) {
356         return (this.serial > other.toSerial());
357     }
358
359     /**
360      * Возвращает true, если текущий объект SerialDate представляет ту же дату,
361      * что и заданный объект SerialDate.
362      *
363      * @param other  дата для сравнения.
364      *
365      * @return <code>true</code>, если текущий объект SerialDate представляет
366      *          ту же дату, что и заданный объект SerialDate.
367      */
368     public boolean isOnOrAfter(final SerialDate other) {
369         return (this.serial >= other.toSerial());
370     }
371
372     /**
373      * Возвращает <code>true</code>, если текущий объект {@link SerialDate}
       принадлежит
374      * заданному диапазону (режим INCLUSIVE).  Порядок дат d1 и d2
375      * не важен.
376      *
377      * @param d1  граничная дата диапазона.
378      * @param d2  другая граничная дата диапазона.
379      *
380      * @return логический признак.
381      */
382     public boolean isInRange(final SerialDate d1, final SerialDate d2) {
383         return isInRange(d1, d2, SerialDate.INCLUDE_BOTH);
384     }
385
386     /**
387      * Возвращает <code>true</code>, если текущий объект SerialDate принадлежит
```

продолжение ⇩

Листинг Б.5 (продолжение)

```
388      * заданному диапазону (включение границ указывается при вызове). Порядок
389      * d1 и d2 не важен.
390      *
391      * @param d1  граничная дата диапазона.
392      * @param d2  другая граничная дата диапазона.
393      * @param include  код, управляющий включением начальной и конечной дат
394      *                 в диапазон.
395      *
396      * @return <code>true</code>, если текущий объект SerialDate принадлежит
397      *         заданному диапазону.
398      */
399     public boolean isInRange(final SerialDate d1, final SerialDate d2,
400                              final int include) {
401         final int s1 = d1.toSerial();
402         final int s2 = d2.toSerial();
403         final int start = Math.min(s1, s2);
404         final int end = Math.max(s1, s2);
405
406         final int s = toSerial();
407         if (include == SerialDate.INCLUDE_BOTH) {
408             return (s >= start && s <= end);
409         }
410         else if (include == SerialDate.INCLUDE_FIRST) {
411             return (s >= start && s < end);
412         }
413         else if (include == SerialDate.INCLUDE_SECOND) {
414             return (s > start && s <= end);
415         }
416         else {
417             return (s > start && s < end);
418         }
419     }
420
421     /**
422      * Вычисляет порядковый номер по дню, месяцу и году.
423      * <P>
424      * 1 января 1900 = 2.
425      *
426      * @param d  день.
427      * @param m  месяц.
428      * @param y  год.
429      *
430      * @return порядковый номер для заданного дня, месяца и года.
431      */
432     private int calcSerial(final int d, final int m, final int y) {
433         final int yy = ((y - 1900) * 365) + SerialDate.leapYearCount(y - 1);
434         int mm = SerialDate.AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[m];
435         if (m > MonthConstants.FEBRUARY) {
436             if (SerialDate.isLeapYear(y)) {
437                 mm = mm + 1;
```

```
438             }
439         }
440         final int dd = d;
441         return yy + mm + dd + 1;
442     }
443
444     /**
445      * Вычисляет день, месяц и год по порядковому номеру.
446      */
447     private void calcDayMonthYear() {
448
449         // Вычислить год по порядковому номеру
450         final int days = this.serial - SERIAL_LOWER_BOUND;
451         // Переоценка из-за проигнорированных високосных дней.
452         final int overestimatedYYYY = 1900 + (days / 365);
453         final int leaps = SerialDate.leapYearCount(overestimatedYYYY);
454         final int nonleapdays = days - leaps;
455         // Недооценка из-за переоцененных лет.
456         int underestimatedYYYY = 1900 + (nonleapdays / 365);
457
458         if (underestimatedYYYY == overestimatedYYYY) {
459             this.year = underestimatedYYYY;
460         }
461         else {
462             int ss1 = calcSerial(1, 1, underestimatedYYYY);
463             while (ss1 <= this.serial) {
464                 underestimatedYYYY = underestimatedYYYY + 1;
465                 ss1 = calcSerial(1, 1, underestimatedYYYY);
466             }
467             this.year = underestimatedYYYY - 1;
468         }
469
470         final int ss2 = calcSerial(1, 1, this.year);
471
472         int[] daysToEndOfPrecedingMonth
473             = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
474
475         if (isLeapYear(this.year)) {
476             daysToEndOfPrecedingMonth
477                 = LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
478         }
479
480         // Получение месяца по порядковому номеру
481         int mm = 1;
482         int sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
483         while (sss < this.serial) {
484             mm = mm + 1;
485             sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
486         }
487         this.month = mm - 1;
488
```

продолжение ⇩

Листинг Б.5 (продолжение)

```
489         // Остается d(+1);
490         this.day = this.serial - ss2
491                     - daysToEndOfPrecedingMonth[this.month] + 1;
492
493     }
494
495 }
```

Листинг Б.6. RelativeDayOfWeekRule.java

```
  1 /* ===========================================================================
  2  * JCommon : библиотека классов общего назначения для платформы Java(tm)
  3  * ===========================================================================
  4  *
  5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
  6  *
  7  * Информация о проекте:  http://www.jfree.org/jcommon/index.html
  8  *
  9  * Библиотека распространяется бесплатно; вы можете свободно распространять
 10  * и/или изменять ее на условиях лицензии Lesser General Public License
 11  * в формулировке Free Software Foundation; либо версии 2.1 лицензии, либо
 12  * (на ваше усмотрение) любой последующей версии.
 13  *
 14  * Библиотека распространяется в надежде, что она будет полезна, но
 15  * БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ, даже без подразумеваемой гарантии ПРИГОДНОСТИ
 16  * для КОНКРЕТНОЙ ЦЕЛИ. За подробностями обращайтесь к GNU Lesser General
 17  * Public License.
 18  *
 19  * Вы должны получить копию лицензии GNU Lesser General Public License
 20  * с этой библиотекой; если этого не произошло, обратитесь в Free Software
 21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 22  * USA.
 23  *
 24  * [Java является зарегистрированной торговой маркой Sun Microsystems, Inc.
 25  * в Соединенных Штатах и других странах].
 26  *
 27  * --------------------------
 28  * RelativeDayOfWeekRule.java
 29  * --------------------------
 30  * (C) Copyright 2000-2003, by Object Refinery Limited and Contributors.
 31  *
 32  * Автор:  Дэвид Гилберт (для Object Refinery Limited);
 33  * Участники:   -;
 34  *
 35  * $Id: RelativeDayOfWeekRule.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
 36  *
 37  * Изменения (начиная с 26 октября 2001)
 38  * --------------------------
 39  * 26-Oct-2001 : Пакет изменен на com.jrefinery.date.*;
 40  * 03-Oct-2002 : Исправлены ошибки по информации Checkstyle (DG);
 41  *
```

```
 */

package org.jfree.date;

/**
 * Правило ежегодной даты, возвращающее дату для каждого года на основании
 *  (а) эталонного правила; (б) дня недели; и (в) параметра выбора.
 * (SerialDate.PRECEDING, SerialDate.NEAREST, SerialDate.FOLLOWING).
 * <P>
 * Например, Страстная пятница может задаваться как 'пятница, ПРЕДШЕСТВУЮЩАЯ
 * Пасхе'.
 *
 * @author Дэвид Гилберт
 */
public class RelativeDayOfWeekRule extends AnnualDateRule {

    /** Ссылка на правило ежегодной даты, на котором основано данное правило. */
    private AnnualDateRule subrule;

    /**
     * День недели (SerialDate.MONDAY, SerialDate.TUESDAY и т.д.).
     */
    private int dayOfWeek;

    /** Признак выбора дня недели (PRECEDING, NEAREST или FOLLOWING). */
    private int relative;

    /**
     * Конструктор по умолчанию - строит правило для понедельника после
     1 января.
     */
    public RelativeDayOfWeekRule() {
        this(new DayAndMonthRule(), SerialDate.MONDAY, SerialDate.FOLLOWING);
    }

    /**
     * Стандартный конструктор - строит правило на основании субправила.
     *
     * @param subrule  правило, определяющее эталонную дату.
     * @param dayOfWeek  день недели по отношению к эталонной дате.
     * @param relative  признак выбора дня недели (preceding, nearest
     *                  или following).
     */
    public RelativeDayOfWeekRule(final AnnualDateRule subrule,
            final int dayOfWeek, final int relative) {
        this.subrule = subrule;
        this.dayOfWeek = dayOfWeek;
        this.relative = relative;
    }

    /**
```

продолжение ⇩

Листинг Б.6 (продолжение)

```
92      * Возвращает субправило (также называемое эталонным правилом).
93      *
94      * @return Правило ежегодной даты, определяющее эталонную дату
95      *         для текущего правила.
96      */
97     public AnnualDateRule getSubrule() {
98         return this.subrule;
99     }
100
101    /**
102     * Назначает субправило.
103     *
104     * @param subrule  Правило ежегодной даты, определяющее эталонную дату
105     *                 для текущего правила.
106     */
107    public void setSubrule(final AnnualDateRule subrule) {
108        this.subrule = subrule;
109    }
110
111    /**
112     * Возвращает день недели для текущего правила.
113     *
114     * @return день недели для текущего правила.
115     */
116    public int getDayOfWeek() {
117        return this.dayOfWeek;
118    }
119
120    /**
121     * Назначает день недели для текущего правила.
122     *
123     * @param dayOfWeek  день недели (SerialDate.MONDAY,
124     *                   SerialDate.TUESDAY и т.д.).
125     */
126    public void setDayOfWeek(final int dayOfWeek) {
127        this.dayOfWeek = dayOfWeek;
128    }
129
130    /**
131     * Возвращает атрибут 'relative', который определяет,
132     *  *какой* день недели нас интересует (SerialDate.PRECEDING,
133     * SerialDate.NEAREST или SerialDate.FOLLOWING).
134     *
135     * @return атрибут 'relative'.
136     */
137    public int getRelative() {
138        return this.relative;
139    }
140
141    /**
```

```
142      * Задает атрибут 'relative' (SerialDate.PRECEDING, SerialDate.NEAREST,
143      * SerialDate.FOLLOWING).
144      *
145      * @param relative  определяет, *какой* день недели выбирается
146      *                  текущим правилом.
147      */
148     public void setRelative(final int relative) {
149         this.relative = relative;
150     }
151
152     /**
153      * Создает копию текущего правила.
154      *
155      * @return копия текущего правила.
156      *
157      * @throws CloneNotSupportedException this should never happen.
158      */
159     public Object clone() throws CloneNotSupportedException {
160         final RelativeDayOfWeekRule duplicate
161             = (RelativeDayOfWeekRule) super.clone();
162         duplicate.subrule = (AnnualDateRule) duplicate.getSubrule().clone();
163         return duplicate;
164     }
165
166     /**
167      * Возвращает дату, сгенерированную текущим правилом для заданного года.
168      *
169      * @param year  год (1900 &lt;= год &lt;= 9999).
170      *
171      * @return дата, сгенерированная правилом для заданного года
172      *         (допускается <code>null</code>).
173      */
174     public SerialDate getDate(final int year) {
175
176         // Проверить аргумент...
177         if ((year < SerialDate.MINIMUM_YEAR_SUPPORTED)
178             || (year > SerialDate.MAXIMUM_YEAR_SUPPORTED)) {
179             throw new IllegalArgumentException(
180                 "RelativeDayOfWeekRule.getDate(): year outside valid range.");
181         }
182
183         // Вычислить дату...
184         SerialDate result = null;
185         final SerialDate base = this.subrule.getDate(year);
186
187         if (base != null) {
188             switch (this.relative) {
189                 case(SerialDate.PRECEDING):
190                     result = SerialDate.getPreviousDayOfWeek(this.dayOfWeek,
191                             base);
192                     break;
```

продолжение ⇩

Листинг Б.6 (продолжение)

```
193                 case(SerialDate.NEAREST):
194                     result = SerialDate.getNearestDayOfWeek(this.dayOfWeek,
195                             base);
196                     break;
197                 case(SerialDate.FOLLOWING):
198                     result = SerialDate.getFollowingDayOfWeek(this.dayOfWeek,
199                             base);
200                     break;
201                 default:
202                     break;
203             }
204         }
205         return result;
206
207     }
208
209 }
```

Листинг Б.7. DayDate.java (окончательная версия)

```
 1 /* ===========================================================================
 2  * JCommon : библиотека классов общего назначения для платформы Java(tm)
 3  * ===========================================================================
 4  *
 5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 ...
36  */
37 package org.jfree.date;
38
39 import java.io.Serializable;
40 import java.util.*;
41
42 /**
43  * Абстрактный класс, представляющий неизменяемые даты с точностью
44  * до одного дня. Реализация отображает дату на целое число, представляющее
45  * смещение в днях от фиксированной точки отсчета.
46  *
47  *  Почему не использовать java.util.Date? Будем использовать, где это имеет смысл.
48  *  Класс java.util.Date бывает *слишком* точным - он представляет момент
49  *  времени с точностью до 1/100 секунды (при этом сама дата зависит от часового
50  *  пояса). Иногда бывает нужно просто представить конкретный день (скажем,
51  *  21 января 2015), не заботясь о времени суток, часовом поясе и т.д.
52  *  Именно для таких ситуаций определяется класс DayDate.
53  *
54  * Для создания экземпляра используется DayDateFactory.makeDate.
55  *
56  * @author Дэвид Гилберт
57  * @author Роберт С. Мартин провел значительную переработку.
58  */
59
60 public abstract class DayDate implements Comparable, Serializable {
61   public abstract int getOrdinalDay();
62   public abstract int getYear();
```

```
63  public abstract Month getMonth();
64  public abstract int getDayOfMonth();
65
66  protected abstract Day getDayOfWeekForOrdinalZero();
67
68  public DayDate plusDays(int days) {
69    return DayDateFactory.makeDate(getOrdinalDay() + days);
70  }
71
72  public DayDate plusMonths(int months) {
73    int thisMonthAsOrdinal = getMonth().toInt() - Month.JANUARY.toInt();
74    int thisMonthAndYearAsOrdinal = 12 * getYear() + thisMonthAsOrdinal;
75    int resultMonthAndYearAsOrdinal = thisMonthAndYearAsOrdinal + months;
76    int resultYear = resultMonthAndYearAsOrdinal / 12;
77    int resultMonthAsOrdinal = resultMonthAndYearAsOrdinal % 12 + Month.JANUARY.
      toInt();
78    Month resultMonth = Month.fromInt(resultMonthAsOrdinal);
79    int resultDay = correctLastDayOfMonth(getDayOfMonth(), resultMonth, resultYear);
80    return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
81  }
82
83  public DayDate plusYears(int years) {
84    int resultYear = getYear() + years;
85    int resultDay = correctLastDayOfMonth(getDayOfMonth(), getMonth(), resultYear);
86    return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
87  }
88
89  private int correctLastDayOfMonth(int day, Month month, int year) {
90    int lastDayOfMonth = DateUtil.lastDayOfMonth(month, year);
91    if (day > lastDayOfMonth)
92        day = lastDayOfMonth;
93    return day;
94  }
95
96  public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
97    int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
98    if (offsetToTarget >= 0)
99      offsetToTarget -= 7;
100   return plusDays(offsetToTarget);
101 }
102
103 public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
104   int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
105   if (offsetToTarget <= 0)
106     offsetToTarget += 7;
107   return plusDays(offsetToTarget);
108 }
109
110 public DayDate getNearestDayOfWeek(Day targetDayOfWeek) {
111   int offsetToThisWeeksTarget = targetDayOfWeek.toInt() - getDayOfWeek().
      toInt();
112   int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
113   int offsetToPreviousTarget = offsetToFutureTarget - 7;
114
```

продолжение ⇩

Листинг Б.7 (продолжение)

```
115     if (offsetToFutureTarget > 3)
116       return plusDays(offsetToPreviousTarget);
117     else
118       return plusDays(offsetToFutureTarget);
119   }
120
121   public DayDate getEndOfMonth() {
122     Month month = getMonth();
123     int year = getYear();
124     int lastDay = DateUtil.lastDayOfMonth(month, year);
125     return DayDateFactory.makeDate(lastDay, month, year);
126   }
127
128   public Date toDate() {
129     final Calendar calendar = Calendar.getInstance();
130     int ordinalMonth = getMonth().toInt() - Month.JANUARY.toInt();
131     calendar.set(getYear(), ordinalMonth, getDayOfMonth(), 0, 0, 0);
132     return calendar.getTime();
133   }
134
135   public String toString() {
136     return String.format(«%02d-%s-%d», getDayOfMonth(), getMonth(), getYear());
137   }
138
139   public Day getDayOfWeek() {
140     Day startingDay = getDayOfWeekForOrdinalZero();
141     int startingOffset = startingDay.toInt() - Day.SUNDAY.toInt();
142     int ordinalOfDayOfWeek = (getOrdinalDay() + startingOffset) % 7;
143     return Day.fromInt(ordinalOfDayOfWeek + Day.SUNDAY.toInt());
144   }
145
146   public int daysSince(DayDate date) {
147     return getOrdinalDay() - date.getOrdinalDay();
148   }
149
150   public boolean isOn(DayDate other) {
151     return getOrdinalDay() == other.getOrdinalDay();
152   }
153
154   public boolean isBefore(DayDate other) {
155     return getOrdinalDay() < other.getOrdinalDay();
156   }
157
158   public boolean isOnOrBefore(DayDate other) {
159     return getOrdinalDay() <= other.getOrdinalDay();
160   }
161
162   public boolean isAfter(DayDate other) {
163     return getOrdinalDay() > other.getOrdinalDay();
164   }
```

```
165
166   public boolean isOnOrAfter(DayDate other) {
167     return getOrdinalDay() >= other.getOrdinalDay();
168   }
169
170   public boolean isInRange(DayDate d1, DayDate d2) {
171     return isInRange(d1, d2, DateInterval.CLOSED);
172   }
173
174   public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
175     int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
176     int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
177     return interval.isIn(getOrdinalDay(), left, right);
178   }
179 }
```

Листинг Б.8. Month.java (окончательная версия)

```
 1 package org.jfree.date;
 2
 3 import java.text.DateFormatSymbols;
 4
 5 public enum Month {
 6   JANUARY(1), FEBRUARY(2), MARCH(3),
 7   APRIL(4),   MAY(5),      JUNE(6),
 8   JULY(7),    AUGUST(8),   SEPTEMBER(9),
 9   OCTOBER(10),NOVEMBER(11),DECEMBER(12);
10   private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
11   private static final int[] LAST_DAY_OF_MONTH =
12     {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
13
14   private int index;
15
16   Month(int index) {
17     this.index = index;
18   }
19
20   public static Month fromInt(int monthIndex) {
21     for (Month m : Month.values()) {
22       if (m.index == monthIndex)
23         return m;
24     }
25     throw new IllegalArgumentException("Invalid month index " + monthIndex);
26   }
27
28   public int lastDay() {
29     return LAST_DAY_OF_MONTH[index];
30   }
31
32   public int quarter() {
33     return 1 + (index - 1) / 3;
34   }
```

продолжение ⇩

Листинг Б.8 (продолжение)

```
35
36   public String toString() {
37     return dateFormatSymbols.getMonths()[index - 1];
38   }
39
40   public String toShortString() {
41     return dateFormatSymbols.getShortMonths()[index - 1];
42   }
43
44   public static Month parse(String s) {
45     s = s.trim();
46     for (Month m : Month.values())
47       if (m.matches(s))
48         return m;
49
50     try {
51       return fromInt(Integer.parseInt(s));
52     }
53     catch (NumberFormatException e) {}
54     throw new IllegalArgumentException(«Invalid month « + s);
55   }
56
57   private boolean matches(String s) {
58     return s.equalsIgnoreCase(toString()) ||
59            s.equalsIgnoreCase(toShortString());
60   }
61
62   public int toInt() {
63     return index;
64   }
65 }
```

Листинг Б.9. Day.java (окончательная версия)

```
 1 package org.jfree.date;
 2
 3 import java.util.Calendar;
 4 import java.text.DateFormatSymbols;
 5
 6 public enum Day {
 7   MONDAY(Calendar.MONDAY),
 8   TUESDAY(Calendar.TUESDAY),
 9   WEDNESDAY(Calendar.WEDNESDAY),
10   THURSDAY(Calendar.THURSDAY),
11   FRIDAY(Calendar.FRIDAY),
12   SATURDAY(Calendar.SATURDAY),
13   SUNDAY(Calendar.SUNDAY);
14
15   private final int index;
16   private static DateFormatSymbols dateSymbols = new DateFormatSymbols();
17
```

```
18   Day(int day) {
19     index = day;
20   }
21
22   public static Day fromInt(int index) throws IllegalArgumentException {
23     for (Day d : Day.values())
24       if (d.index == index)
25         return d;
26     throw new IllegalArgumentException(
27       String.format(«Illegal day index: %d.», index));
28   }
29
30   public static Day parse(String s) throws IllegalArgumentException {
31     String[] shortWeekdayNames =
32       dateSymbols.getShortWeekdays();
33     String[] weekDayNames =
34       dateSymbols.getWeekdays();
35
36     s = s.trim();
37     for (Day day : Day.values()) {
38       if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
39           s.equalsIgnoreCase(weekDayNames[day.index])) {
40         return day;
41       }
42     }
43     throw new IllegalArgumentException(
44       String.format("%s is not a valid weekday string", s));
45   }
46
47   public String toString() {
48     return dateSymbols.getWeekdays()[index];
49   }
50
51   public int toInt() {
52     return index;
53   }
54 }
```

Листинг Б.10. DateInterval.java (окончательная версия)

```
 1 package org.jfree.date;
 2
 3 public enum DateInterval {
 4   OPEN {
 5     public boolean isIn(int d, int left, int right) {
 6       return d > left && d < right;
 7     }
 8   },
 9   CLOSED_LEFT {
10     public boolean isIn(int d, int left, int right) {
11       return d >= left && d < right;
12     }
```

продолжение ⇩

```
  },
  CLOSED_RIGHT {
    public boolean isIn(int d, int left, int right) {
      return d > left && d <= right;
    }
  },
  CLOSED {
    public boolean isIn(int d, int left, int right) {
      return d >= left && d <= right;
    }
  };

  public abstract boolean isIn(int d, int left, int right);
}
```

Листинг Б.11. WeekInMonth.java (окончательная версия)

```
package org.jfree.date;

public enum WeekInMonth {
  FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
  private final int index;

  WeekInMonth(int index) {
    this.index = index;
  }

  public int toInt() {
    return index;
  }
}
```

Листинг Б.12. WeekdayRange.java (окончательная версия)

```
package org.jfree.date;

public enum WeekdayRange {
  LAST, NEAREST, NEXT
}
```

Листинг Б.13. DateUtil.java (окончательная версия)

```
package org.jfree.date;

import java.text.DateFormatSymbols;

public class DateUtil {
  private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();

  public static String[] getMonthNames() {
    return dateFormatSymbols.getMonths();
  }

```

```
12   public static boolean isLeapYear(int year) {
13     boolean fourth = year % 4 == 0;
14     boolean hundredth = year % 100 == 0;
15     boolean fourHundredth = year % 400 == 0;
16     return fourth && (!hundredth || fourHundredth);
17   }
18
19   public static int lastDayOfMonth(Month month, int year) {
20     if (month == Month.FEBRUARY && isLeapYear(year))
21       return month.lastDay() + 1;
22     else
23       return month.lastDay();
24   }
25
26   public static int leapYearCount(int year) {
27     int leap4 = (year - 1896) / 4;
28     int leap100 = (year - 1800) / 100;
29     int leap400 = (year - 1600) / 400;
30     return leap4 - leap100 + leap400;
31   }
32 }
```

Листинг Б.14. DayDateFactory.java (окончательная версия)

```
 1 package org.jfree.date;
 2
 3 public abstract class DayDateFactory {
 4   private static DayDateFactory factory = new SpreadsheetDateFactory();
 5   public static void setInstance(DayDateFactory factory) {
 6     DayDateFactory.factory = factory;
 7   }
 8
 9   protected abstract DayDate _makeDate(int ordinal);
10   protected abstract DayDate _makeDate(int day, Month month, int year);
11   protected abstract DayDate _makeDate(int day, int month, int year);
12   protected abstract DayDate _makeDate(java.util.Date date);
13   protected abstract int _getMinimumYear();
14   protected abstract int _getMaximumYear();
15
16   public static DayDate makeDate(int ordinal) {
17     return factory._makeDate(ordinal);
18   }
19
20   public static DayDate makeDate(int day, Month month, int year) {
21     return factory._makeDate(day, month, year);
22   }
23
24   public static DayDate makeDate(int day, int month, int year) {
25     return factory._makeDate(day, month, year);
26   }
27
28   public static DayDate makeDate(java.util.Date date) {
```

продолжение ⇩

Листинг Б.14 (продолжение)

```
    return factory._makeDate(date);
  }

  public static int getMinimumYear() {
    return factory._getMinimumYear();
  }

  public static int getMaximumYear() {
    return factory._getMaximumYear();
  }
}
```

Листинг Б.15. SpreadsheetDateFactory.java (окончательная версия)

```
package org.jfree.date;

import java.util.*;

public class SpreadsheetDateFactory extends DayDateFactory {
  public DayDate _makeDate(int ordinal) {
    return new SpreadsheetDate(ordinal);
  }

  public DayDate _makeDate(int day, Month month, int year) {
    return new SpreadsheetDate(day, month, year);
  }

  public DayDate _makeDate(int day, int month, int year) {
    return new SpreadsheetDate(day, month, year);
  }

  public DayDate _makeDate(Date date) {
    final GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(date);
    return new SpreadsheetDate(
      calendar.get(Calendar.DATE),
      Month.fromInt(calendar.get(Calendar.MONTH) + 1),
      calendar.get(Calendar.YEAR));
  }

  protected int _getMinimumYear() {
    return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
  }

  protected int _getMaximumYear() {
    return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
  }
}
```

Листинг Б.16. SpreadsheetDate.java (окончательная версия)

```
 1 /* ===========================================================================
 2  * JCommon : библиотека классов общего назначения для платформы Java(tm)
 3  * ===========================================================================
 4  *
 5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 6  *
...
52  *
53  */
54
55 package org.jfree.date;
56
57 import static org.jfree.date.Month.FEBRUARY;
58
59 import java.util.*;
60
61 /**
62  * Представляет дату с использованием целого числа, по аналогии с реализацией
63  * в Microsoft Excel.  Поддерживаемый диапазон дат:
64  * с 1 января 1900 по 31 декабря 9999.
65  * <p/>
66  * Учтите, что в Excel существует намеренная ошибка, вследствие которой год
67  * 1900 считается високосным, тогда как в действительности он таковым не является.
68  * Дополнительная информация приведена на сайте Microsoft в статье Q181370:
69  * <p/>
70  * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
71  * <p/>
72  * По правилам Excel 1 января 1900 = 1.  По правилам этого класса
73  * 1 января 1900 = 2.
74  * В результате номер дня этого класса будет отличаться от номера Excel
75  * в январе и феврале 1900...но затем Excel прибавляет лишний день
76  * (29 февраля 1900, который в действительности не существует!), и с этого
77  *  момента нумерация дней совпадает.
78  *
79  * @author David Gilbert
80  */
81 public class SpreadsheetDate extends DayDate {
82   public static final int EARLIEST_DATE_ORDINAL = 2;     // 1/1/1900
83   public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
84   public static final int MINIMUM_YEAR_SUPPORTED = 1900;
85   public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
86   static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
87     {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
88   static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
89     {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
90
91   private int ordinalDay;
92   private int day;
93   private Month month;
94   private int year;
```

продолжение ⇗

Листинг Б.16 (продолжение)

```
 95
 96   public SpreadsheetDate(int day, Month month, int year) {
 97     if (year < MINIMUM_YEAR_SUPPORTED || year > MAXIMUM_YEAR_SUPPORTED)
 98       throw new IllegalArgumentException(
 99         "The 'year' argument must be in range " +
100         MINIMUM_YEAR_SUPPORTED + " to " + MAXIMUM_YEAR_SUPPORTED + ".");
101     if (day < 1 || day > DateUtil.lastDayOfMonth(month, year))
102       throw new IllegalArgumentException("Invalid 'day' argument.");
103
104     this.year = year;
105     this.month = month;
106     this.day = day;
107     ordinalDay = calcOrdinal(day, month, year);
108   }
109
110   public SpreadsheetDate(int day, int month, int year) {
111     this(day, Month.fromInt(month), year);
112   }
113
114   public SpreadsheetDate(int serial) {
115     if (serial < EARLIEST_DATE_ORDINAL || serial > LATEST_DATE_ORDINAL)
116       throw new IllegalArgumentException(
117         "SpreadsheetDate: Serial must be in range 2 to 2958465.");
118
119     ordinalDay = serial;
120     calcDayMonthYear();
121   }
122
123   public int getOrdinalDay() {
124     return ordinalDay;
125   }
126
127   public int getYear() {
128     return year;
129   }
130
131   public Month getMonth() {
132     return month;
133   }
134
135   public int getDayOfMonth() {
136     return day;
137   }
138
139   protected Day getDayOfWeekForOrdinalZero() {return Day.SATURDAY;}
140
141   public boolean equals(Object object) {
142     if (!(object instanceof DayDate))
143       return false;
144
```

```
145     DayDate date = (DayDate) object;
146     return date.getOrdinalDay() == getOrdinalDay();
147   }
148
149   public int hashCode() {
150     return getOrdinalDay();
151   }
152
153   public int compareTo(Object other) {
154     return daysSince((DayDate) other);
155   }
156
157   private int calcOrdinal(int day, Month month, int year) {
158     int leapDaysForYear = DateUtil.leapYearCount(year - 1);
159     int daysUpToYear = (year - MINIMUM_YEAR_SUPPORTED) * 365 + leapDaysForYear;
160     int daysUpToMonth = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[month.toInt()];
161     if (DateUtil.isLeapYear(year) && month.toInt() > FEBRUARY.toInt())
162       daysUpToMonth++;
163     int daysInMonth = day - 1;
164     return daysUpToYear + daysUpToMonth + daysInMonth + EARLIEST_DATE_ORDINAL;
165   }
166
167   private void calcDayMonthYear() {
168     int days = ordinalDay - EARLIEST_DATE_ORDINAL;
169     int overestimatedYear = MINIMUM_YEAR_SUPPORTED + days / 365;
170     int nonleapdays = days - DateUtil.leapYearCount(overestimatedYear);
171     int underestimatedYear = MINIMUM_YEAR_SUPPORTED + nonleapdays / 365;
172
173     year = huntForYearContaining(ordinalDay, underestimatedYear);
174     int firstOrdinalOfYear = firstOrdinalOfYear(year);
175     month = huntForMonthContaining(ordinalDay, firstOrdinalOfYear);
176     day = ordinalDay - firstOrdinalOfYear - daysBeforeThisMonth(month.toInt());
177   }
178
179   private Month huntForMonthContaining(int anOrdinal, int firstOrdinalOfYear) {
180     int daysIntoThisYear = anOrdinal - firstOrdinalOfYear;
181     int aMonth = 1;
182     while (daysBeforeThisMonth(aMonth) < daysIntoThisYear)
183       aMonth++;
184
185     return Month.fromInt(aMonth - 1);
186   }
187
188   private int daysBeforeThisMonth(int aMonth) {
189     if (DateUtil.isLeapYear(year))
190       return LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
191     else
192       return AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
193   }
194
195   private int huntForYearContaining(int anOrdinalDay, int startingYear) {
```

продолжение ⇩

Листинг Б.16 (продолжение)

```
196     int aYear = startingYear;
197     while (firstOrdinalOfYear(aYear) <= anOrdinalDay)
198       aYear++;
199
200     return aYear - 1;
201   }
202
203   private int firstOrdinalOfYear(int year) {
204     return calcOrdinal(1, Month.JANUARY, year);
205   }
206
207   public static DayDate createInstance(Date date) {
208     GregorianCalendar calendar = new GregorianCalendar();
209     calendar.setTime(date);
210     return new SpreadsheetDate(calendar.get(Calendar.DATE),
211                                Month.fromInt(calendar.get(Calendar.MONTH) + 1),
212                                calendar.get(Calendar.YEAR));
213
214   }
215 }
```

B Перекрестные ссылки

Перекрестные ссылки для «запахов кода» и эвристических правил из приложения А организованы по принципу Номер_главы-Номер_страницы.

C1	16-306, 16-309, 17-323
C2	16-309, 16-312, 16-320, 17-323
C3	16-311, 16-312, 16-314, 17-323
C4	17-323
C5	17-324
E1	17-324
E2	17-324
F1	14-266, 17-325
F2	17-325
F3	17-325
F4	16-304, 16-314, 17-325
G1	16-307, 17-325
G2	16-305, 17-326
G3	16-305, 17-326
G4	16-308, 17-326
G5	16-308, 16-313, 16-317, 16-320, 17-327
G6	6-128, 16-309, 16-311, 16-315, 16-318, 16-319, 17-328
G7	16-309, 17-329
G8	16-311, 17-329
G9	16-311, 16-312, 16-314, 17-330
G10	5-107, 16-311, 17-330
G11	15-295, 16-311, 16-314, 16-317, 17-330
G12	16-312, 16-313, 16-314, 16-315, 16-320, 17-331
G13	16-313, 16-314, 17-331

G14 16-314, 17-331
G15 16-315, 17-333
G16 16-315, 17-333
G17 16-315, 17-334, 17-337
G18 16-315, 16-316, 16-317, 17-334
G19 16-316, 16-317, 17-335
G20 16-316, 17-335
G21 16-317, 17-336
G22 16-318, 17-336
G23 3-63, 14-265, 16-319, 17-338
G24 16-320, 17-338
G25 16-320, 17-339
G26 17-340
G27 17-340
G28 15-293, 17-341
G29 15-394, 17-341
G30 15-295, 17-341
G31 15-296, 17-342
G32 15-296, 17-343
G33 15-298, 17-343
G34 3-60, 6-128, 17-344
G35 5-111, 17-345
G36 6-126, 17-346
J1 16-306, 17-347
J2 16-308, 17-347
J3 16-311, 16-312, 17-348
N1 15-296, 16-307, 16-309, 16-310, 16-314, 16-315, 16-316, 16-319, 17-349
N2 16-307, 17-351
N3 16-312, 16-314, 17-352
N4 15-294, 16-316, 17-352
N5 2-45, 14-248, 17-353
N6 15-293, 17-353
N7 15-294, 17-354
T1 16-304, 16-305, 17-354
T2 16-304, 17-354
T3 16-305, 17-354

T4 17-355
T5 16-305, 16-306, 17-355
T6 16-305, 17-355
T7 16-306, 17-355
T8 16-306, 17-355
T9 17-356

Эпилог

На конференции по гибким методологиям, проходившей в Денвере в 2005 году, Элизабет Хедриксон[1] вручила мне зеленый браслет наподобие тех, которые стали такими популярными после Ланса Армстронга. На браслете было написано «Одержим тестированием». Я с гордостью носил этот браслет. С тех пор как Кент Бек научил меня методологии разработки через тестирование (TDD) в 1999 году, я действительно стал буквально одержим этой темой.

Но потом случилось нечто странное. Я обнаружил, что не могу снять этот браслет, причем вовсе не потому, что он застрял у меня на запястье. Браслет открыто формулировал мою профессиональную этику. Он стал визуальным признаком моего стремления к написанию самого лучшего кода, который я могу написать. Мне казалось, что снять его – значит предать эту этику вместе с моими устремлениями.

Браслет так и остается у меня на запястье. Занимаясь программированием, я вижу его периферийным зрением. Он постоянно напоминает мне о том обещании, которое я дал сам себе – обещании писать чистый код.

[1] http://www.qualitytree.com/

Алфавитный указатель

Л

М

Н

О