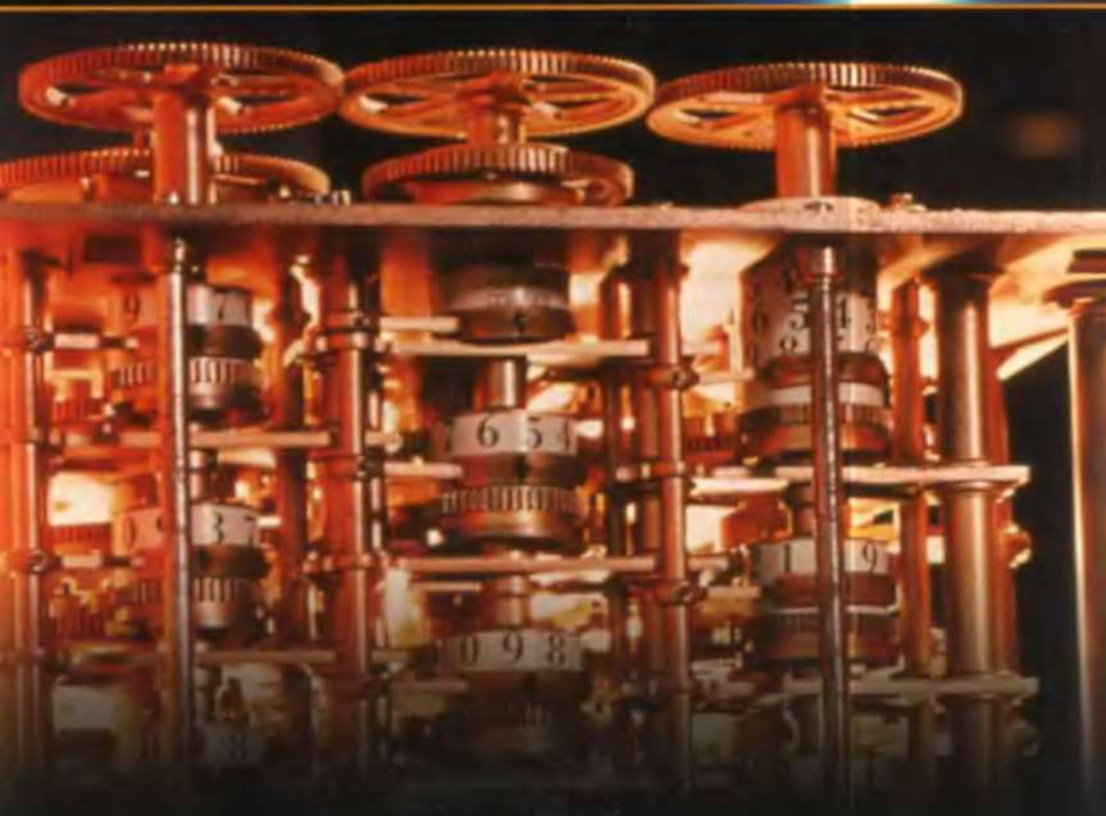
Серия книг Роберта К. Мартина



ЭФФЕКТИВНАЯ РАБОТА С УНАСЛЕДОВАННЫМ КОДОМ

Майкл К. Физерс

WORKING EFFECTIVELY WITH LEGACY CODE

Michael C. Feathers

Prentice Hall Professional Technical Reference
Upper Saddle River, NJ 07458
www.phptr.com

ЭФФЕКТИВНАЯ РАБОТА С УНАСЛЕДОВАННЫМ КОДОМ

Майкл К. Физерс

Издательский дом “ВИЛЬЯМС”
Москва • Санкт-Петербург • Киев
2009

ББК 32.973.26-018.2.75
Ф50
УДК 681.3.07

Издательский дом “Вильямс”

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *И.В. Берштейна*

По общим вопросам обращайтесь в Издательский дом “Вильямс” по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Физерс, Майкл.

Ф50 Эффективная работа с унаследованным кодом. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2009. — 400 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1530-6 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Майкл Физерс

Эффективная работа с унаследованным кодом

Литературный редактор *Л.Н. Важенина*
Верстка *М.А. Удалов*
Художественный редактор *С.А. Чернокозинский*
Корректор *Л.А. Гордиенко*

Подписано в печать 29.01.2009. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 25,0. Уч.-изд. л. 21,8.
Тираж 1000 экз. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО “Печатный двор” им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

ООО “И. Д. Вильямс”, 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1530-6 (рус.)
ISBN 0-13-117705-2 (англ.)

Оглавление

Содержание

Посвящается Энн, Деборе и Райану —
самым ярким средоточиям моей жизни.

— Майкл

Предисловие

“С этого все и началось...” — этой фразой Майкл Физерс описывает во вступлении к своей книге момент, с которого началось его страстное увлечение программированием.

“С этого все и началось...” Знакомо ли вам подобное ощущение? Можете ли вы определить такой момент в своей жизни, когда для вас “все и началось”? Было ли это отдельное событие, круто изменившее течение вашей жизни и приведшее в конечном итоге к тому, что вы приобрели эту книгу и начали ее чтение с настоящего предисловия?

Со мной это случилось в шестом классе школы. Я заинтересовался наукой, техникой и космосом. И тогда моя мать нашла и заказала для меня по каталогу детский пластмассовый конструктор для сборки игрушечного компьютера. Он назывался *Digi-Comp I*. Сорок лет спустя этот пластмассовый конструктор занимает почетное место на моей книжной полке. Он послужил тем стимулом, который пробудил во мне неугасающий до сих пор интерес к программированию. Благодаря ему я получил первое, хотя и весьма отдаленное представление о том, насколько интересно писать программы, способные решать задачи для людей. Тот игрушечный компьютер состоял всего лишь из трех RS-триггеров и шести схем И в виде пластмассовых конструкций, но и этого было достаточно, чтобы научиться азам программирования. С того все для меня и началось...

Но мой пыл вскоре был умерен осознанием того факта, что программные системы практически всегда приходят в полный беспорядок. То, что начинается с ясно выкристаллизовавшегося замысла в уме программиста, со временем загнивает, как кусок испорченного мяса. Небольшая изящная система, построенная год назад, превращается в следующем году в непроходимую трясину из запутанных функций и переменных.

Почему это происходит? Почему программные системы начинают портиться и почему они не могут работать четко?

В одних случаях мы виним во всем заказчиков, а в других — обвиняем их в изменении требований. Ведь нам удобно верить в то, что если бы заказчики удовлетворились своими требованиями к проектированию программной системы, то она и не давала бы сбоев. Всю вину за изменение требований к системе мы привыкли сваливать на заказчиков.

Итак, на повестке дня стоит следующий актуальный вопрос: *изменение требований*. Проектные решения, не предусматривающие изменение требований, изначально считаются неудачными. В связи с этим цель всякого компетентного разработчика программного обеспечения — выработать такое проектное решение, которое допускало бы изменение требований.

На первый взгляд, решить такую задачу не так-то просто. В самом деле, практически каждая проектируемая система страдает недостатком — медленным, разрушительным загниванием. И это загнивание настолько распространено, что для обозначения испорченных программ был придуман специальный термин. Такие программы мы называем *унаследованным кодом*.

Унаследованный код. Это выражение инстинктивно вызывает отвращение у всякого, занимающегося программированием. Ведь воображение тотчас рисует ужасные картины продирания сквозь непроходимые таежные буреломы и топи с упорно цепляющимися за ноги ветками деревьев и кустарников, безжалостно впивающимися в тело тучами мошкары и одуряющими запахами гниения, разложения и застоя. Мучения, которые доставляет разбирательство с унаследованным кодом, нередко гасят любые первые порывы энтузиазма, возникающие при программировании.

Многие из нас пытались найти способы, *препятствующие* устареванию кода. На тему принципов, шаблонов и практических приемов, помогающих программистам поддерживать работоспособность разрабатываемых ими систем, написано немало литературы. Но у Майкла Физерса сложилось по этому поводу совсем иное представление, которого многим из нас недостает: препятствовать устареванию кода совершенно недостаточно. Ведь даже самая дисциплинированная группа разработчиков, владеющая самыми совершенными принципами, пользующаяся самыми лучшими шаблонами и придерживающаяся самых выверенных на практике приемов, время от времени вносит путаницу, которая приводит к постепенной порче и загниванию кода. Воспрепятствовать загниванию недостаточно — следует попытаться *обратить* этот процесс вспять.

Именно такому обращению загнивания вспять и посвящена эта книга. В ней идет речь о том, как запутанная, непонятная и сложная система медленно, но верно, постепенно и по частям превращается в простую, изящно структурированную и правильно спроектированную систему. Это книга об обращении вспять процесса энтропии.

Но предупреждая ваш порыв чрезмерного энтузиазма, должен заметить, что обращение загнивания кода вспять — дело непростое и нескорое. Методы, шаблоны и средства, представленные автором в этой книге, достаточно эффективны, но они требуют усилий, времени, терпения и внимания. Эта книга не является панацеей и не дает никаких рецептов, как исключить сразу все накопившиеся признаки загнивания в вашей системе. Вместо этого в книге описывается ряд дисциплин, концепций и подходов, которые вы можете взять на вооружение и которые *помогут* вам в дальнейшей профессиональной деятельности превратить постепенно *деградирующие* системы в постепенно *совершенствуемые* системы.

— Роберт К. Мартин
29 июня 2004 года

Вступление

Помните ли вы первую написанную вами программу? Я помню. Это была небольшая графическая программа, написанная мной для одной из первых моделей ПК. Я начал заниматься программированием позже, чем большинство моих коллег. Разумеется, я знаком с компьютерами с детских лет. Я даже помню свои первые яркие впечатления от мини-компьютера, увиденного в одном учреждении, но в детстве у меня не было возможности даже сесть за компьютер. Позднее, когда я стал подростком, у некоторых из моих сверстников появились первые модели микрокомпьютеров TRS-80. Меня они заинтересовали, хотя я испытывал некоторые опасения, поскольку знал, что компьютерные игры затягивают. Поэтому я старался оставаться безразличным к ним. Даже не знаю почему, но я проявлял сдержанный интерес к компьютерам. Затем, когда я учился в колледже, у моего соседа по комнате в общежитии появился компьютер, и тогда я приобрел компилятор C, чтобы научиться программировать. С этого все и началось. Я проводил целые ночи, опробуя на практике разные идеи и анализируя исходный код в редакторе emacs, входившем в состав компилятора. Это было увлекательное, интересное занятие, которое мне очень нравилось.

Надеюсь, что нечто подобное пришлось испытать и вам, читатель, т.е. радость от того, что удалось сделать нечто полезное и работоспособное на компьютере. Подобное ощущение знакомо практически всем программистам, которых я спрашивал об этом. Именно оно отчасти побудило многих из нас выбрать эту профессию, но испытываем ли мы его теперь, хотя бы иногда, в наших повседневных трудах?

Несколько лет назад я однажды вечером позвонил после работы своему другу Эрику Миду. Я знал, что Эрик только что начал консультировать новую группу разработчиков, и поэтому я спросил его: “Как у них идут дела?” Он ответил: “Они пишут унаследованный код, дружище”. Это был один из тех немногих случаев, когда слова коллеги задели меня за живое. У меня от них внутри буквально все перевернулось. Эрик очень точно выразил словами то ощущение, которое я нередко испытывал при первом посещении групп разработчиков. Они очень старались, но в конце рабочего дня — то ли потому что на них давили сроки и ответственность момента, то ли из-за отсутствия образцов лучшего кода для сравнения с их трудами — многие из них просто начинали писать унаследованный код.

Что же представляет собой унаследованный код? Раньше я пользовался этим понятием без какого-то конкретного определения. А теперь попробуем проанализировать следующее строгое определение этого понятия: унаследованным называется код, полученный от кого-то другого. Это может быть код, приобретенный одной компанией у другой или же унаследованный одной группой разработчиков от другой при переходе к иному проекту, т.е. *унаследованный код* — это чужой код. Но для программистов это понятие имеет намного более широкий смысл. Оно приобрело со временем множество смысловых оттенков и более глубокое значение.

Какие ассоциации вызывает у вас термин *унаследованный код*, когда вы слышите его? Возможно, у вас, как и у меня, возникают мысли о запутанной, непонятной структуре, коде, который требуется изменить, но на самом деле вы его просто не понимаете. Поэтому у вас появляются воспоминания о бессонных ночах, проведенных в попытках ввести в такой код новые свойства, которые, казалось бы, нетрудно было добавить, о состоянии полной деморализации и болезненном ощущении от базы кода, вызывающем полное безразличие у всей группы разработчиков к коду, от которого проще помереть, чем исправить его. Отчасти вы испытываете ощущение безнадежности своих попыток улучшить подобный

код. Вот поэтому приведенное выше определение не имеет никакого отношения к самому понятию унаследованного кода. Ведь код может деградировать разными путями, которые зачастую никак не связаны с местом его происхождения.

В программировании понятие *унаследованный код* нередко употребляется как жаргонное обозначение трудноизменяемого кода, который совершенно непонятен. Но приобретя многолетний опыт работы с группами разработчиков над разрешением серьезных осложнений с кодом, я пришел к другому определению данного понятия.

С моей точки зрения, *унаследованный код* — это просто код, не прошедший тесты. Такое определение далось мне горьким опытом. Что же должны делать тесты для выявления неудачного кода? Для меня ответ на данный вопрос очевиден, и он составляет главный предмет, подробно разрабатываемый в этой книге.

Код без тестов является неудачным. И совершенно неважно, насколько хорошо он написан, объектно-ориентирован или инкапсулирован. С помощью текстов мы можем быстро и под полным контролем изменить поведение нашего кода. А без них мы на самом деле не знаем, становится ли наш код лучше или хуже.

Такое условие может показаться слишком суровым. Как, например, быть с чистым кодом? Если база кода довольно чиста и правильно структурирована, то разве этого недостаточно? Отвечая на этот вопрос, нужно постараться избежать ошибки. Мне лично нравится чистый код — и даже больше, чем многим из тех, кого я знаю, но одной чистоты кода явно недостаточно. Разработчики зачастую идут на большой риск, пытаясь радикально изменить код без тестирования. Это все равно, что заниматься воздушной гимнастикой без страховочной сетки. Каждый совершаемый шаг требует поразительного умения и ясного понимания того, что может произойти. Точно знать, что произойдет, если изменить пару переменных, — это зачастую все равно, что быть полностью уверенным в том, что ваш партнер-гимнаст подхватит вас за руки, когда вы совершите очередное сальто в воздухе. Если вы работаете в группе, разрабатывающей совершенно чистый код, значит, вы находитесь в лучшем положении, чем большинство других программистов. Мой опыт показывает, что группы, разрабатывающие весь код такой чистоты, встречаются крайне редко. Они, скорее, исключение из общего правила. И знаете почему? Если у них нет поддерживающих тестов, то изменения в их коде будут вноситься намного медленнее, чем в тех группах, где есть такие тесты.

Конечно, группы могут стремиться разрабатывать чистый код с самого начала, но для того, чтобы сделать чище старый код, им потребуется немало времени. И, как правило, этого не удается сделать полностью. Именно поэтому я безо всяких колебаний определяю унаследованный код как код, не прошедший тесты. Это, на мой взгляд, удачное и практичное определение, указывающее на решение проблемы подобного рода.

Выше вскользь упоминалось о тестах, но эта книга посвящена не тестам, а методам, позволяющим уверенно вносить изменения в любую базу кода. В последующих главах рассматриваются методы, применяемые для лучшего понимания кода, его тестирования, реорганизации и дополнения новыми свойствами.

Читая эту книгу, вы непременно обратите внимание на то, что в ней не идет речь о совершенно конкретном коде. Примеры кода, приведенные в этой книге, носят весьма условный характер, поскольку я не имею права полностью раскрывать код своих клиентов. Но во многих примерах я постарался сохранить основную суть прикладного кода. Нельзя сказать, что эти примеры всегда характерны и показательны. Это, безусловно, лишь небольшие фрагменты качественного кода, хотя они представляют собой далеко не самое

лучшее из того, что можно было бы использовать в качестве примера в данной книге. Помимо соблюдения условий конфиденциальности по отношению к своим клиентам, я просто не мог привести в этой книге такой код, чтобы не скрыть важные моменты во множестве мелких деталей и тем самым довести вас, читатель, до полного отчаяния. По этим причинам многие примеры кода в этой книге довольно лаконичны. Если при анализе любого из этих примеров вам покажется, что вы применяете намного более крупные методы и получаете гораздо более худшие результаты, то обратитесь к совету, который я даю по тексту, и сами решите, настолько он вам подходит, несмотря на кажущуюся простоту приводимого примера.

Методы, представленные в этой книге, были проверены на достаточно крупных фрагментах кода. Но рамки этой книги просто не позволяют привести в ней более крупные примеры кода. В частности, когда в примерах кода встречаются знаки многоточия (...), то их следует трактовать как указание вставить 500 или около того строк скверного когда, как, например:

```
m_pDispatcher->register(listener);
...
m_nMargins++;
```

В этой книге речь идет не только о совершенно конкретном коде, но и о совершенно конкретном проектировании. Качественное проектирование должно стать нашей конечной целью, а унаследованный код — это некий промежуточный результат на пути к этой цели. В некоторых главах этой книги описываются способы ввода нового кода в уже существующую базу кода и показывается, как это делается, исходя из удачно выбранных принципов проектирования. Конечно, вы можете постепенно расширить участки очень хорошего, качественного кода в базе унаследованного кода, но не удивляйтесь, если на определенной стадии внесения изменений код станет скверным. Такая работа сродни хирургическому вмешательству. Нам нужно сделать разрез, добраться до внутренностей и, оставив на время эстетические соображения, ответить на вопрос — можно ли улучшить состояние внутренних органов пациента? Если да, то должны ли мы сразу же зашить пациента, посоветовать ему правильно питаться и бегать на длинные дистанции и тут же забыть о его болезни? Конечно, можно поступить и так, но на самом деле нам нужно объективно оценить состояние здоровья пациента, правильно поставить диагноз его заболевания, постараться вылечить и вернуть пациента в более здоровое состояние. Возможно, он уже не будет отличаться олимпийским здоровьем, но нельзя себе позволить, чтобы лучшее стало врагом хорошего. Базы кода должны стать более здоровыми и простыми в использовании. Если пациент почувствует себя чуть лучше, то это зачастую очень удобный момент, чтобы помочь ему придерживаться более здорового образа жизни. Именно эту цель мы и преследуем в отношении унаследованного кода, т.е. мы пытаемся добиться того момента, когда обычно испытываем облегчение. Мы его ожидаем и активно стремимся упростить изменение кода. А когда мы поддерживаем это ощущение во всей группе разработчиков, то результаты проектирования сразу же улучшаются.

Способы, рассматриваемые в этой книге, выявлены и изучены мной вместе с моими коллегами в течение многолетней работы с клиентами, когда мы пытались установить контроль над непокорными базами кода. С остротой проблемы унаследованного кода я столкнулся совершенно случайно. Когда я только начал сотрудничать с компанией Object Mentor, то основной моей задачей была помощь группам разработчиков для разрешения серьезных трудностей и доведении их квалификации и взаимодействия до такого состояния, когда они могли выдавать качественный код. Мы часто пользовались практическими приемами экстремального программирования, чтобы помочь группам разработчиков луч-

ше контролировать свою работу, интенсивнее сотрудничать друг с другом и добиваться нужного результата. Мне часто кажется, что экстремальное программирование больше подходит для сплочения коллектива разработчиков, которым нужно производить качественное программное обеспечение каждые две недели, чем для разработки самого программного обеспечения.

Но трудности начались с самого начала. Многие проекты экстремального программирования оказались незрелыми. У клиентов, с которыми мне приходилось иметь дело, были необычайно крупные базы кода, что доставляло им немало хлопот. Им нужно было каким-то образом контролировать свою работу и начать выдавать результат. Со временем я обнаружил, что постоянно проделываю со своими клиентами одно и то же. Это ощущение достигло апогея своей остроты во время работы над одним из проектов, который группа консультируемых мной разработчиков выполняла в области финансов. До моего прихода в группу ее члены уже пришли к выводу, что блочное тестирование — отличное средство, но тесты, которые они выполняли, были написаны по полному сценарию, совершали многочисленные обращения к базе данных и исполняли крупные фрагменты кода. Писать такие тесты было нелегко, и группа выполняла их не очень часто, поскольку для этого требовалось немало времени. Как только я принялся разрывать вместе с ними зависимости, чтобы получить более мелкие фрагменты кода для тестирования, меня посетило ужасное и малоприятное ощущение, то я уже проделывал нечто подобное едва ли не с каждой группой разработчиков, которую я консультировал. Это была черновая работа, которую приходится выполнять, чтобы работать с кодом, держа его под постоянным контролем, если, конечно, знать, как это делается. И тогда я решил, что мне стоит поразмыслить над тем, как мы разрешаем подобные затруднения и фиксируем их, чтобы помочь группе разработчиков преодолеть их и упростить ей работу с своей базой кода.

Еще одно замечание относительно примеров кода, приведенных в этой книге: они составлены на разных языках программирования. Бо́льшая их часть написана на языках Java, C++ и C. В частности, Java был выбран потому, что это весьма распространенный язык программирования, C++ — поскольку на этом языке можно представить ряд особых трудностей, возникающих в унаследованной среде, а C — из-за того, что он позволяет высветить многие проблемы, характерные для процедурного унаследованного кода. Все эти языки программирования охватывают бо́льшую часть спектра вопросов, возникающих при рассмотрении унаследованного кода. Но даже если вы программируете на другом языке, то примеры кода, представленные на упомянутых выше языках, все равно окажутся полезными для вас, и поэтому стоит их проанализировать. Ведь многие методы, рассматриваемые в этой книге, могут с тем же успехом применяться при программировании на других языках, включая Delphi, Visual Basic, COBOL и Fortran.

Надеюсь, что методы, представленные в этой книге, окажутся полезными для вас и позволят вновь обрести утраченный по тем или иным причинам интерес к программированию, которое приносит удовольствие и удовлетворение от проделанной работы. Если же вы не испытываете ни то, ни другое в своей повседневной работе, то я надеюсь, что предлагаемые мной методы помогут вам и вашим коллегам по работе вновь испытать эти ощущения.

Благодарности

Прежде всего, я в серьезном долгу перед своей женой Энн и моими детьми Деборой и Райаном. Подготовка, написание и выход в свет этой книги стали возможными благодаря их любви и моральной поддержки. Мне бы хотелось также поблагодарить "дядю Боба" Мартина, президента и основателя компании Object Mentor, — за строгий прагматичный подход к разработке и проектированию программного обеспечения, позволяющий отделить самое главное от несущественного и давший мне возможность прочно утвердиться в своей профессии почти 10 лет назад, когда мои перспективы казались весьма зыбкими, а также за возможность лучше понимать код и поработать за прошедшие пять лет с таким количеством людей, которого я просто не мог бы себе раньше представить.

Благодарю также Кента Бека (Kent Beck), Мартина Фаулера (Martin Fowler), Рона Джеффриса (Ron Jeffries) и Уарда Каннингхэма (Ward Cunningham) за полезные советы и обучение искусству проектирования, программирования и работы с людьми. Особая благодарность выражается всем, кто рецензировал рукопись этой книги, в том числе официальным рецензентам Свену Гортсу (Sven Gorts), Роберту К. Мартину (Robert C. Martin), Эрику Миду (Erik Meade) и Биллу Уэйку (Bill Wake), а также неофициальным рецензентам: д-ру Роберту Коссу (Dr. Robert Koss), Джеймсу Греннингу (James Grenning), Лоуэллу Линдстрёму (Lowell Lindstrom), Майке Мартину (Micah Martin), Рассу Руферу (Russ Rufer) и сотрудникам компании Silicon Valley Patterns Group и Джеймсу Ньюкирку (James Newkirk).

Благодарю также рецензентов самых первых вариантов рукописи книги, опубликованных в Интернете. Их отзывы оказали существенное влияние на направленность книги после реорганизации ее формата. Заранее приношу извинения тем, кого я здесь не упомянул. Итак, выражаю свою искреннюю признательность следующим первым рецензентам моей книги: Даррену Хоббсу (Darren Hobbs), Мартину Липперту (Martin Lippert), Киту Николасу (Keith Nicholas), Флипу Пламли (Phlip Plumlee), К. Киту Рэю (C. Keith Ray), Роберту Блюму (Robert Blum), Биллу Беррису (Bill Burris), Уильяму Капуто (William Caputo), Брайану Мэрику (Brian Marick), Стиву Фриману (Steve Freeman), Дэвиду Путмену (David Putman), Эмили Бак (Emily Bache), Дэйву Эстелсу (Dave Astels), Расселу Хиллу (Russel Hill), Кристиану Сепульведе (Christian Sepulveda) и Брайану Кристоферу Робинсону (Brian Christopher Robinson).

Кроме того, выражаю благодарность Джошуа Кериевски (Joshua Kerievsky) — за первое рецензирование книги, а также Джеффу Лангру (Jeff Langr) — за полезные советы и периодическое рецензирование по ходу работы над книгой.

Все рецензенты рукописи моей книги помогли мне значительно улучшить ее качество, и если в ней найдутся ошибки, то только мои собственные.

Благодарю Мартина Фаулера, Ральфа Джонсона (Ralph Johnson), Билла Опдайка (Bill Opdyke), Дона Робертса (Don Roberts) и Джона Бранта (John Brant) за их труды в области реорганизации кода. Они меня вдохновляли.

Кроме того, я в огромном долгу перед Джеем Пакликом (Jay Packlick), Жаком Морелем (Jacques Morel) и Келли Мауэр (Kelly Mower) из фирмы Sabre Holdings, а также перед Грэхемом Райтом (Graham Wright) из компании Workshare Technology за их поддержку и отзывчивость.

Особая благодарность выражается Полу Петралии (Paul Petralia) — за оказанную помощь и своевременную поддержку, так необходимые автору книги, а также Мишель

Винсенти (Michelle Vincenti), Лори Лайонс (Lori Lyons), Кристе Хансинг (Krista Hansing) и остальным сотрудникам издательства Prentice-Hall.

Особая признательность выражается также Гэри (Gary) и Джоан Физерс (Joan Feathers), Эйприл Робертс (April Roberts), д-ру Раймунду Эге (Dr. Raimund Ege), Дэвиду Лопесу де Квинтана (David Lopez de Quintana), Карлосу Пересу (Carlos Perez), Карлосу М. Родригесу (Carlos M. Rodriguez) и почтенному д-ру Джону К. Комфорту (Dr. John C. Comfort) — за многолетнюю поддержку и оказанную помощь. Благодарю и Брайна Баттона (Brian Button) за пример, составленный им к главе 21 этой книги. Он написал код данного примера почти за час, когда мы составляли вместе учебный курс по реорганизации кода, и этот код стал для меня излюбленным учебным примером.

Кроме того, особая благодарность выражается Янику Топу (Janik Top), инструментальная пьеса которого "Будущее" (De Futura) вдохновляла меня в последние недели работы над книгой.

И наконец, мне бы хотелось поблагодарить всех, с кем мне пришлось работать в течение последних нескольких лет, чья пытливость и сомнения только способствовали упрочению материала этой книги.

— Майкл Физерс
mfeathers@objectmentor.com
www.objectmentor.com
www.michaelfeathers.com

Введение

Об организации книги

Я опробовал самые разные форматы этой книги, прежде чем остановиться на ее настоящем формате. Ведь различные способы и практические приемы, приносящие пользу в работе с унаследованным кодом, трудно объяснить по отдельности. Даже самые простые изменения нередко вносятся легче, если удается найти подходящие швы, создать фиктивные объекты и разорвать зависимости, используя пару способов разрывания зависимостей. Поэтому ради удобства чтения книги я решил организовать бо́льшую ее часть, а именно часть II "Изменение программного обеспечения" в формате ответов на часто задаваемые вопросы. Конкретные способы нередко требуют применения других способов, поэтому отдельные главы, составленные из часто задаваемых вопросов, тесно взаимосвязаны. Практически в каждой главе вы найдете ссылки на другие главы и разделы, в которых описываются конкретные способы и примеры реорганизации кода. Заранее приношу извинения за дополнительные хлопоты в связи с листанием страниц книги в поисках ответов на поставленные вопросы, но я вполне допускаю, что вы будете читать эту книгу именно так, а не от корки до корки, пытаясь уяснить принцип действия всех способов подряд.

В части II я попытался дать ответы на самые общие вопросы, возникающие в работе с унаследованным кодом. Каждая глава в этой части озаглавлена по конкретному рассматриваемому вопросу. И хотя названия глав от этого становятся длинными, тем не менее, они позволяют быстро найти раздел с ответами на возникшие у вас вопросы.

Материал части II подкрепляется рядом вводных глав из части I "Внутренний механизм изменений в коде", а также целым рядом примеров реорганизации кода из части III "Методы разрыва зависимостей", которые могут быть очень полезными в работе с унаследованным кодом. Непременно прочитайте вводные главы, особенно главу 4 "Модель шва". В этих главах представлен контекст и терминология для всех рассматриваемых далее способов. Кроме того, если вам встретится термин, который не поясняется в основном тексте книги, загляните в словарь специальных терминов в конце данной книги.

Особенность примеров реорганизации кода, приведенных в части III, состоит в том, что они предназначены для выполнения без тестов, чтобы найти подходящее место для самих тестов. Поэтому рекомендуется ознакомиться с этими примерами, чтобы обнаружить в них дополнительные возможности для приручения непослушного унаследованного кода.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш веб-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: `info@williamspublishing.com`
WWW: `http://www.williamspublishing.com`

Информация для писем из:

России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1
Украины: 03150, Киев, а/я 152

Часть I

Внутренний механизм изменений в коде

Глава 1

Изменения в программном коде

Возможность вносить изменения в код сама по себе замечательна. Этим можно даже зарабатывать себе на жизнь. Но в одних случаях вносить изменения в код намного труднее, чем в других. Среди тех, кто занимается программированием, эта тема обсуждается недостаточно активно. В лучшем случае имеется литература по реорганизации кода. В этой связи целесообразно расширить дискуссию о том, как обращаться с кодом в самых острых ситуациях. И с этой целью мы углубимся во внутренний механизм изменений в коде.

Четыре причины изменений в программном коде

Ради простоты рассмотрим следующие четыре причины изменений в программном коде.

1. Ввод свойства
2. Исправление программной ошибки
3. Улучшение структуры кода
4. Оптимизация использования ресурсов

Ввод свойств и исправление программных ошибок

Ввод свойства можно отнести к самому очевидному виду изменений в программном коде. Замечая определенное поведение программной системы, пользователи нередко заявляют, что она должна делать еще кое-что.

Допустим, что мы работаем над веб-приложением и заказчик просит переместить логотип его компании с левой стороны страницы на правую. Мы пытаемся объяснить ему, что сделать это не так-то просто. Помимо того, заказчик требует внести другие изменения, в том числе анимировать логотип в следующей редакции веб-страницы. Следует ли считать такое изменение исправлением программной ошибки или вводом нового свойства? Это зависит от точки зрения. Так, с точки зрения заказчика это определенно просьба устранить недостаток. Возможно, посмотрев веб-сайт и проведя совещание в своем отделе, он решил внести изменения в местоположение логотипа, а заодно попросить исполнителя придать логотипу дополнительные функции. С точки зрения разработчика такое изменение считается вводом совершенно нового свойства. Ход его рассуждений приблизительно следующий: “Если бы они перестали наконец менять свое решение, мы уже завершили бы проект”. Но в некоторых организациях перемещение логотипа рассматривается лишь как исправление программной ошибки, несмотря на то, что разработчикам придется для этого приложить немало дополнительного труда.

Невольно возникает мысль отнести все эти рассуждения к разряду исключительно субъективных. Вы, например, считаете изменение устранением программной ошибки, а я

считаю его вводом нового свойства — и все тут. Как ни прискорбно, но во многих организациях устранение программных ошибок и ввод новых свойств отслеживаются и учитываются раздельно из-за особенностей составления договоров или выработки инициатив по качеству. Люди могут до бесконечности спорить о том, что представляет собой изменение программного кода — ввод новых свойств или исправление программных ошибок, но в конечном счете речь идет только об изменении кода или других артефактов. К сожалению, эти рассуждения об исправлении программных ошибок или вводе новых свойств заслоняют собой нечто намного более важное с технической точки зрения: изменение поведения. Ведь между вводом нового поведения и изменением старого поведения имеется существенное отличие.

Поведение является едва ли не самой важной составляющей программного обеспечения. Именно от него зависит работа пользователей с программным обеспечением. Пользователи только приветствуют появление дополнительного поведения, если оно отвечает их требованиям, но если мы изменим или удалим поведение, от которого зависит их работа, т.е. введем программные ошибки, то пользователи перестанут нам доверять.

В приведенном выше примере с логотипом компании речь, по существу, идет о вводе нового поведения. После внесения соответствующих изменений система должна отображать логотип на правой стороне веб-страницы. Но в то же время это означает исключение другого, существовавшего ранее поведения — логотип уже не будет появляться на левой стороне веб-страницы.

Рассмотрим более сложный пример. Допустим, что заказчик требует ввести на правой стороне страницы логотип, который до этого не появлялся на левой ее стороне. И в этом случае мы вводим новое поведение, но исключаем ли мы старое? Отображалось ли прежде что-нибудь на том месте, где теперь должен появиться логотип? Изменяем ли мы при этом старое поведение или же старое вместе с вновь добавляемым?

Попробуем провести различие, более полезное с точки зрения программирования. Если нам приходится видоизменять код (в данном случае — код HTML), то мы можем изменить и поведение. Если же мы только вводим код и вызываем его, то зачастую вводим и новое поведение. Обратимся к еще одному примеру. Ниже приведен метод класса `Java`.

```
Java class:
public class CDPlayer
{
    public void addTrackListing(Track track) {
        ...
    }
    ...
}
```

У этого класса имеется метод, позволяющий нам ввести перечни дорожек звукозаписи. Введем еще один метод, который позволит нам заменить перечни дорожек звукозаписи.

```
public class CDPlayer
{
    public void addTrackListing(Track track) {
        ...
    }
```

```
    public void replaceTrackListing(String name, Track track) {
        ...
    }
    ...
}
```

Добавили ли мы новое поведение в наше приложение или же изменили уже имеющееся поведение, введя этот метод? Ни то, ни другое. Сам по себе ввод метода еще не изменяет поведение, если только он не вызывается каким-то образом.

Внесем в код еще одно изменение, поместив новую кнопку в пользовательском интерфейсе проигрывателя компакт-дисков и связав ее с методом `replaceTrackListing`. В этом случае мы уже вводим новое поведение, указанное в методе `replaceTrackListing`, но в то же время мы немного изменяем уже существующее поведение. Ведь пользовательский интерфейс будет выглядеть иначе, а возможно, и чуть медленнее отображаться после добавления новой кнопки. По-видимому, ввести новое поведение практически невозможно, не изменив в какой-то степени уже имеющееся поведение.

Улучшение структуры кода

Улучшение проекта относится к другому виду изменения программного кода. Так, если нам требуется изменить структуру программного обеспечения, чтобы сделать его сопровождение более удобным, то, как правило, мы стараемся оставить без изменения его поведение. Когда же мы затрагиваем поведение в ходе данного процесса, такое изменение мы нередко называем программной ошибкой. Одна из главных причин, по которым многие программисты даже не пытаются улучшить проект, нередко объясняется тем, что в ходе данного процесса можно относительно легко потерять уже имеющееся поведение или же создать плохое поведение.

Действие по улучшению структуры кода без изменения его поведения называется *реорганизацией кода*, или *рефакторингом*, как его нередко называют программисты. Идея реорганизации кода состоит в том, чтобы сделать сопровождение программного обеспечения более удобным, не меняя его поведение. С этой целью пишутся тесты для проверки неизменности поведения по ходу продвижения мелкими шагами в течение всего процесса. Программисты уже давно выполняют окончательное редактирование кода в своих системах, но только за последние годы они стали применять на практике реорганизацию кода. Отличие реорганизации от обычного окончательного редактирования кода состоит в том, что мы просто не делаем никаких рискованных шагов — будь то менее рискованное переформатирование кода или же более рискованное вмешательство вроде переписывания отдельных фрагментов кода. Вместо этого мы делаем ряд мелких структурных изменений, поддерживаемых тестами, чтобы упростить изменение кода. Главная особенность реорганизации кода с точки зрения его изменения состоит в том, что в данном случае не предполагается никаких функциональных изменений, хотя поведение может немного измениться вследствие структурных изменений, которые способны оказать как положительное, так и отрицательное влияние на производительность системы.

Оптимизация

Оптимизация в какой-то степени похожа на реорганизацию кода, хотя и делается с иной целью. Выполняя реорганизацию кода и оптимизацию, мы стремимся сохранить

функциональные возможности точно такими же, как и до внесения изменений, но в то же время мы намерены изменить нечто другое. Если речь идет о реорганизации кода, то это "нечто другое" означает структуру программы, которую требуется сделать более простой для сопровождения. А если речь идет об оптимизации, то "нечто другое" означает определенный ресурс, используемый программой (как правило, время или память).

Собирая все вместе

Некоторое сходство реорганизации кода с оптимизацией кажется, на первый взгляд, странным. Они кажутся ближе, чем ввод новых свойств или исправление программных ошибок. Но так ли это на самом деле? Общим для реорганизации кода и оптимизации является стремление с нашей стороны сохранить функциональные возможности без изменения, но изменив что-то другое.

Работая в системе, мы можем, как правило, изменить три элемента: структуру, функциональные возможности и использование ресурсов.

Покажем, что именно обычно изменяется и что остается более или менее неизменным при внесении четырех рассмотренных выше видов изменений, хотя чаще всего меняются сразу все три указанных элемента.

	Ввод свойства	**Исправление программной ошибки**	**Реорганизация кода**	**Оптимизация**
Структура	Изменения	Изменения	Изменения	—
Функциональные возможности	Изменения	Изменения	—	—
Использование ресурсов	—	—	—	Изменения

Внешне реорганизация кода и оптимизация выглядят очень похоже. Они вообще не затрагивают функциональные возможности. Но что, если мы должны принимать во внимание новые функциональные возможности по отдельности? Ведь когда мы вводим функциональные возможности часто, то добавляем новую функциональную возможность, не изменяя уже существующую.

	Ввод свойства	**Исправление программной ошибки**	**Реорганизация кода**	**Оптимизация**
Структура	Изменения	Изменения	Изменения	—
Новые функциональные возможности	Изменения	—	—	—
Функциональные возможности	—	Изменения	—	—
Использование ресурсов	—	—	—	Изменения

Ввод новых свойств, реорганизация кода и оптимизация — все эти виды изменения программного кода не затрагивают существующие функциональные возможности. Но если внимательно проанализировать исправление ошибок, то можно обнаружить, что оно все же изменяет функциональные возможности, хотя эти изменения зачастую весьма незна-

чительны по сравнению с той массой функциональных возможностей, которые не были изменены.

Ввод свойств и исправление ошибок очень похожи на реорганизацию кода и оптимизацию. Во всех четырех случаях нам требуется изменить отдельные функциональные возможности и поведение, но в то же время мы стремимся сохранить бо́льшую часть из того, что уже имеется (рис. 1.1).

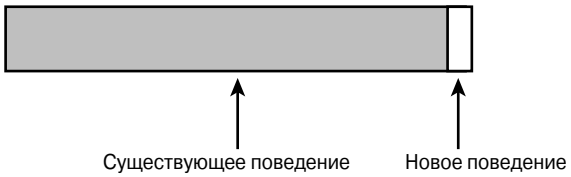


Рис. 1.1. Сохранение поведения

Все рассмотренное выше дает ясное представление о том, что должно произойти, когда мы вносим изменения, но что все это означает на практике? С положительной стороны, такое представление, по-видимому, дает нам возможность лучше понять, на чем именно мы должны сосредоточить свои усилия. Ведь мы должны убедиться в том, что небольшое число изменяемых нами элементов изменяются правильно. А с отрицательной стороны, это не единственный момент, на котором мы должны сосредоточить свое внимание. Ведь нам нужно решить, как мы собираемся сохранить остальное поведение. К сожалению, для сохранения поведения недостаточно просто оставить нетронутым соответствующий код. Мы должны знать, что поведение не меняется, но это не так просто. Обычно приходится сохранять львиную долю поведения, но это еще полбеды. Главное, что нам зачастую неизвестно, какая именно часть этого поведения подвергается риску, когда мы вносим свои изменения. Если бы мы знали это точно, то могли бы сосредоточить основное внимание на данном поведении, не беспокоясь об остальном.

Для того чтобы уменьшить подобный риск, мы должны задать себе четыре вопроса.

1. Какие изменения нам нужно сделать?
2. Как мы узнаем, что сделали их правильно?
3. Как мы узнаем, что ничего не нарушили?
4. До какой степени мы можем пойти на изменения, чтобы они не оказались рискованными?

Большинство групп разработчиков, с которыми мне приходилось работать, пытались справляться с подобным риском довольно консервативным способом. Они сводили к минимуму число изменений, вносимых в базу кода. Иногда для группы разработчиков главным становится следующее правило: “Если не ломается, то и нечего исправлять”. В некоторых случаях это правило соблюдается не всеми членами группы, причем неявно. Так, отдельные разработчики просто очень осмотрительно вносят изменения, рассуждая следующим образом: “Что? Создать для этого еще один метод? Нет уж, я лучше введу в метод ряд строк кода там, где я легко их обнаружу вместе с остальным кодом. Так мне проще вносить правки, да и надежнее”.

Трудно избавиться от мысли, что недостатки программного обеспечения можно свести к минимуму, исключая их в корне, но, к сожалению, мы нередко попадаем в эту ловушку. Избегая создавать новые классы и методы, мы невольно укрупняем существующие классы и усложняем их понимание. Прежде чем внести изменения в любую крупную систему,

зачастую приходится тратить некоторое время на ознакомление с тем, как она работает. Отличие хороших систем от плохих заключается в том, что в хороших системах вы чувствуете себя довольно уверенно, внося в них изменения после ознакомления с ними. А переход от решения внести изменения в плохо структурированный код к непосредственным действиям сродни прыжку с обрыва, чтобы не попасть в лапы преследующего вас тигра. Вы постоянно колеблетесь, не зная точно, насколько вы готовы внести изменения в код.

Еще одна скверная тенденция — стремление избежать изменений. Если изменения не вносятся в код часто, то он быстро становится запущенным. Так, если не внести изменения в крупный класс хотя бы два раза в неделю, то разбить его на отдельные части будет довольно сложно. Если же вносить изменения в код вовремя, то это довольно быстро войдет в привычку. И тогда намного проще решить, что следует и что не следует разбивать в коде.

И самое последнее дело — бояться вносить изменения в код. К сожалению, у многих групп разработчиков постепенно вырабатывается боязнь изменений, которая усиливается с каждым днем. Иногда члены таких групп даже не осознают степень подобной боязни до тех пор, пока не освоят более совершенные методы, после чего боязнь у них постепенно исчезает.

Итак, мы выяснили, что избегать изменений — очень скверно, но есть ли этому какая-то разумная альтернатива? Можно, например, просто работать прилежнее или же нанять больше разработчиков, чтобы у них было больше времени для анализа и тщательной проверки кода на предмет внесения в него нужных изменений. Конечно, чем больше предоставляется времени и тщательнее проверка кода, тем безопаснее вносимые в него изменения. Но еще вопрос, насколько можно доверять результатам самой проверки кода — пусть даже самой тщательной.

Глава 2

Работа с ответной реакцией

Изменения в системе делаются двумя основными способами. Я предпочитаю называть их *правкой наудачу* и *покрытием и модификацией*. К сожалению, *правка наудачу* весьма распространена в программировании. Пользуясь этим способом, вы тщательно планируете изменения, которые собираетесь сделать, убеждаетесь в том, что понимаете видоизменяемый код, а затем приступаете к внесению в него изменений. По завершении вы запускаете систему, чтобы убедиться в активизации изменений, а затем тщательно проверяете, не нарушили ли вы что-нибудь. Выискивание огрех и тщательная проверка системы являются обязательными составляющими данного способа. Внося изменения наудачу, вы надеетесь, что они сделаны правильно, и поэтому вам требуется дополнительное время, чтобы убедиться в этом.

На первый взгляд, правка наудачу кажется подобной "работе с большой осторожностью", т.е. очень профессиональному подходу к делу. "Осторожность" при этом ставится во главу угла, а при радикальных изменениях соблюдается особая осторожность, поскольку в этом случае слишком велика вероятность неудачного исхода. Но соблюдение осторожности — не единственная мера, гарантирующая безопасность. Вряд ли кто-нибудь из нас доверится хирургу, оперирующему кухонным ножом, — даже если он и делает свое дело очень осторожно. Эффективное изменение программного обеспечения, как и эффективная хирургия, требует более серьезной квалификации. Работа с большой осторожностью мало что дает, если вы не пользуетесь правильными средствами и методами.

Совсем иначе изменения вносятся методом покрытия и модификации. В основу этого метода положен принцип работы с *сеткой безопасности* при изменении программного обеспечения. Используемая при этом сетка безопасности не имеет ничего общего с той сеткой, которую мы расставляем под своими столами, чтобы ловить нас, когда мы падаем со своих стульев. Напротив, это своего рода покров, который мы надеваем на код, с которым мы работаем, чтобы исключить из него утечку неудачных изменений, способных отрицательно повлиять на остальную часть программы. Покрытие программного обеспечение в данном случае означает его покрытие тестами. Имея в своем распоряжении хороший набор тестов, окружающих фрагмент кода, мы можем вносить изменения и быстро обнаруживать их положительное или отрицательное воздействие на код. И в этом случае мы соблюдаем осторожность, но, получая ответную реакцию, мы можем вносить изменения более тщательно.

Если вы незнакомы с таким применением тестов, то все сказанное выше покажется вам не совсем привычным. По традиции тесты пишутся и выполняются после разработки программного обеспечения. Группа программистов пишет код, а группа тестировщиков выполняет затем тесты кода, чтобы проверить, насколько они соответствуют определенным техническим требованиям. В некоторых весьма традиционных центрах разработки именно так и разрабатывается программное обеспечение. Группа разработчиков получает ответную реакцию, но не сразу, а с некоторым запаздыванием. Так, разработчики могут получить результаты тестирования кода лишь через несколько недель, а то и месяцев после завершения работы над ним.

Такое тестирование на самом деле является не более чем попыткой выявить правильность кода. И хотя оно преследует верную цель, тесты можно использовать и по-другому. В частности, мы можем выполнять тесты, чтобы обнаружить изменения.

По традиции такое тестирование называется регрессивным. Мы периодически выполняем тесты, чтобы проверить правильность известного нам поведения и выяснить, работает ли система так же, как и прежде.

Когда у нас имеются тесты, окружающие участки кода, в которые мы собираемся внести изменения, такие тесты выполняют роль программных тисков. Это дает возможность сохранять без изменения большую часть поведения, зная, что изменению подлежит только то, что предполагается изменить.

Программные тиски

Тиски — приспособление для закрепления заготовки или детали при обработке или сборке, состоящее из корпуса, подвижной и неподвижной губок. — Советский энциклопедический словарь, 4-е изд., М., 1990.

Тесты, предназначенные для обнаружения изменений, служат в качестве своего рода тисков вокруг кода, как бы закрепляя его поведение на месте. При внесении изменений это позволяет нам заранее знать, что мы изменяем поведение только по отдельным частям. Иными словами, мы в состоянии контролировать свою работу.

Регрессивное тестирование — отличная идея. Так почему же им пользуются так редко? С применением регрессивного тестирования связано одно небольшое затруднение. Те, кто применяет его на практике, делают это зачастую на уровне прикладного интерфейса. И совершенно не важно, какое это приложение: для Интернета, с командной строкой или графическим интерфейсом. По традиции, регрессивное тестирование всегда считалось тестированием на прикладном уровне. Но это не совсем удачное применения такого тестирования, поскольку оно может оказаться полезным и на более мелком уровне.

Проведем небольшой мысленный эксперимент. Допустим, что мы унаследовали крупную функцию, содержащую большое количество сложной логики. Мы анализируем, осмысливаем ее код и обращаемся за помощью к тем, кто знает о нем больше, чем мы, а затем вносим в него изменения. Нам нужно убедиться в том, что этими изменениями мы ничего не нарушили, но как это сделать? К счастью, у нас есть группа контроля, которая сумеет выполнить в нерабочее время имеющийся в ее распоряжении набор регрессивных тестов. Поэтому мы обращаемся к этой группе с просьбой запланировать выполнение тестов на ночное время, и хорошо, если мы сделаем это как можно раньше. Ведь зачастую выполнение регрессивных тестов планируется на середину недели, и поэтому нам приходится ждать, своей очереди, тем более, что мы не одни. Ведь и другие вносят в код свои изменения. Наконец, нам удается сдать свой код на тестирование, и, вздохнув с некоторым облегчением, мы возвращаемся к своей работе. Нам ведь предстоит еще внести целый ряд изменений во фрагменты не менее сложного кода.

На следующее утро нам звонят из группы контроля, сообщая о том, что тесты AE1021 и AE1029, выполненные прошлой ночью, не прошли. Тестировщик не совсем уверен, что причиной этого стали внесенные нами изменения, но он звонит нам, потому что знает, что за данный фрагмент кода отвечаем мы, а не он. Поэтому мы приступаем к отладке, чтобы выяснить причину сбоя, обусловленную нашими или же чужими изменениями.

Насколько такая ситуация реальна? К сожалению, она очень даже реальна.

Рассмотрим другой пример.

Допустим, что нам нужно внести изменения в довольно длинную и сложную функцию. К счастью, нам удается найти для ее проверки набор блочных тестов. Те, кто работал над данным кодом в последний раз, написали набор из 20 тестов, предназначенных для его тщательной проверки. Мы выполняем эти тесты и обнаруживаем, что все они проходят. Затем мы просматриваем тесты, чтобы выяснить конкретное поведение кода.

Теперь мы готовы внести в код свои изменения, но осознаем, что пока еще толком не знаем, как это сделать. Ведь код не совсем ясен, и нам нужно лучше разобраться в нем, прежде чем вносить в него изменения. Тесты не могут выловить все подряд, поэтому нам нужно прояснить код настолько, чтобы более уверенно вносить в него изменения. Кроме того, нам бы не хотелось вынуждать себя и других проделывать ту же самую работу еще раз, чтобы попытаться понять этот код. Зачем зря тратить время!

Итак, мы начинаем понемногу реорганизовывать код. С этой целью мы извлекаем ряд методов и переносим их в блок условной логики. После этого незначительного изменения мы выполняем небольшой набор блочных тестов. Они проходят едва ли не каждый раз, когда мы их выполняем. Если несколько минут назад мы сделали ошибку, обратив логику проверки по условию, а тест не прошел, то буквально за минуту мы исправляем свою ошибку. После реорганизации код становится намного яснее. Мы внесли в него те изменения, которые собирались внести, и теперь мы уверены в их правильности. Далее мы добавляем ряд других тестов для проверки нового поведения. Программистам, которым придется в дальнейшем работать с данным фрагментом кода, будет легче в нем разобраться, а в их распоряжении окажутся тесты, покрывающие функции этого кода.

Какая ответная реакция вам больше подходит — через минуту или на следующее утро? В каком случае изменения в коде делаются более эффективно?

Блочное тестирование является одной из самых важных составляющих работы с унаследованным кодом. Регрессивные тесты на уровне системы сами по себе замечательны, но мелкие, локализованные тесты просто неоценимы в такой работе. Они способны дать ответную реакцию в ходе разработки и позволяют реорганизовывать код намного более безопасным способом.

Что означает блочное тестирование

Термин *блочный тест* уже давно укоренился в разработке программного обеспечения. Для многих концепций блочных тестов общим является принцип их выполнения обособленно от отдельных компонентов программного обеспечения. Каковы эти компоненты? Они определяются по-разному, но в блочном тестировании нас, прежде всего, интересуют самые элементарные единицы поведения системы. В процедурном коде такими единицами зачастую оказываются функции, а в объектно-ориентированном коде — классы.

Средства тестирования

В этой книге понятием *средства тестирования* обозначается выполняемый тестовый код, который пишется для проверки определенного фрагмента программного обеспечения. Для работы с кодом можно использовать разные виды средств тестирования. Так, в главе 5 рассматриваются среды тестирования xUnit и FIT. Обе среды служат для проведения тестирования, описываемого в этой книге.

Можем ли мы проверить лишь одну функцию или класс? В процедурных системах зачастую очень трудно проверить каждую функцию по отдельности. Ведь функции верхнего

уровня вызывают другие функции, а те, в свою очередь, вызывают еще одни функции, и так до самого машинного уровня. А в объектно-ориентированных системах проверять отдельные классы немного проще, но дело в том, что классы не существуют обособленно. Трудно себе представить созданные классы, не использующие другие классы. Такое встречается крайне редко, не так ли? Как правило, это небольшие классы данных или же классы структурных данных, например, стеки или очереди, но и в них используются другие классы.

Обособленное тестирование является важной составляющей определения блочного теста, но в чем его значение? Ведь многие ошибки могут возникать и после интегрирования отдельных частей программного обеспечения. Разве не менее важны крупные тесты, охватывающие обширные функциональные участки кода? Конечно, они важны, и этого нельзя отрицать, но с крупными тестами связан ряд следующих затруднений.

- **Локализация ошибок.** По мере отступления теста от того, что он проверяет, становится все труднее определить, что означает непрохождение теста. Нередко для выявления конкретной причины непрохождения теста приходится немало потрудиться. Для этого нужно проанализировать входные данные теста, сам сбой и определить место его возникновения на пути от входных данных теста к выходным. Все это приходится проделывать и с блочными тестами, хотя в данном случае работы намного меньше, поскольку такие тесты весьма невелики.
- **Время выполнения.** Чем крупнее тест, тем больше времени требуется для его выполнения. Это превращает процедуру тестирования в малоприятное занятие. Тесты, которые требуют слишком долгого выполнения, так и не выполняются.
- **Покрытие.** Зачастую оказывается очень трудно выявить связь между фрагментом кода и значениями, с помощью которых он проверяется. Обычно для проверки фрагмента кода с помощью теста используются средства покрытия, но, вводя новый код, мы вынуждены прилагать немало труда для создания тестов высокого уровня, проверяющих этот новый код.

Одним из самых малоприятных моментов, связанных с крупными тестами, является иллюзия, которую мы питаем в отношении локализации ошибок, считая, что чем чаще мы выполняем тесты, тем легче обнаружить ошибки в коде. Если мы выполняем тесты и они проходят, а затем вносим незначительные изменения и тесты не проходят, то мы точно знаем, где именно возник сбой. Он, скорее всего, связан с последним незначительным изменением. И тогда мы возвращаемся к этому изменению и делаем следующую попытку. Но если мы выполняем крупные тесты, то на это обычно уходит немало времени, и тогда мы стараемся избегать выполнения таких тестов достаточно часто для того, чтобы действительно локализовать ошибки.

Блочные тесты восполняют пробел, который не в состоянии восполнить крупные тесты. С их помощью мы можем проверять фрагменты кода независимо, группировать тесты, чтобы выполнять одни тесты при одних условиях, а другие — при других условиях. Кроме того, с помощью блочных тестов мы можем довольно быстро локализовать ошибки. Если мы считаем, что в конкретном фрагменте кода возникла ошибка и можно проверить этот код средствами тестирования, то мы быстро составляем тест, чтобы выявить с его помощью ошибку.

Ниже перечислены основные достоинства хороших блочных тестов.

1. Быстро выполняются.
2. Помогают локализовать ошибки и выявить недостатки в коде.

Среди программистов нередко возникают разногласия по поводу того, следует ли считать отдельные тесты блочными. Является ли тест действительно блочным, если в нем используется другой выходной класс? Если вернуться к упомянутым выше основным достоинствам блочных тестов, то выполняется ли конкретный тест быстро и позволяет ли он оперативно локализовать ошибки? Естественно, что это крайности, а истина, как всегда, посредине. Так, одни тесты оказываются крупнее, чем другие, и в них используется несколько классов. В действительности, они могут даже представлять собой результат незначительного интегрирования других тестов. По отдельности они могут выполняться быстро, а что произойдет при их совместном выполнении? Если тест должен проверять класс вместе с другими взаимодействующими с ним классами, то такой тест проявляет тенденцию к разрастанию. А если не позаботиться вовремя о получении экземпляра отдельного класса в средствах тестирования, то насколько просто будет сделать это после добавления дополнительного кода? Это никогда не бывает просто, и тогда тестирование откладывается до более удобного момента. Со временем тест может разрастить и уже выполняться в течение 1/10 секунды.

Блочный тест, который выполняется в течение 1/10 секунды, считается медленным блочным тестом.

И это серьезно. На момент написания этой книги выполнение блочного теста в течение 1/10 секунды считалось бесконечно долгим для такого теста. Проведем элементарные расчеты. Если мы работаем над проектом, насчитывающим 3000 классов и на каждый фрагмент кода приходится по 10 тестов, то в итоге получается 30 тыс. тестов. Сколько времени потребуется для выполнения всех этих тестов в данном проекте, если на каждый из них приходится по 1/10 секунды? Около часа. Это слишком долгое ожидание ответной реакции. Даже если классов в проекте окажется наполовину меньше, то ответной реакции все равно придется ждать около получаса. С другой стороны, если тест выполняется в течение 1/100 секунды, то ждать ответной реакции придется уже 5–10 минут. В подобных случаях рекомендуется работать с подмножествами тестов, но тогда все они вряд ли будут выполняться каждые два часа.

Если полагаться на закон Мура, то в перспективе можно ожидать практически мгновенную ответную реакцию на тестирование даже самых крупных систем. У меня есть подозрение, что работа с такими системами станет похожей на работу с кодом, который может сразу же дать нам знать, когда он изменится в худшую сторону.

Блочные тесты должны выполняться быстро. В противном случае это не блочные тесты.

Другие тесты нередко выдают себя за блочные. Такие тесты нельзя считать блочными, если они проявляют следующие признаки.

1. Обращаются к базе данных.
2. Общаются по сети.
3. Обращаются к файловой системе.
4. Для их выполнения приходится специально настраивать среду, например, править конфигурационные файлы.

Тесты, проявляющие подобные признаки, сами по себе не так уж и плохи. Их зачастую стоит писать, причем в средствах блочного тестирования. Но очень важно отделить их от подлинно блочных тестов, чтобы иметь в своем распоряжении набор *быстро* выполняемых тестов при каждом внесении изменений в код.

Тестирование на более высоком уровне

Несмотря не все преимущества блочных тестов, в программном обеспечении имеются места для тестирования на более высоком уровне. Тесты более высокого уровня охватывают своей проверкой различные сценарии и варианты взаимодействия в приложении. С их помощью можно точно определить по очереди поведение отдельных классов. Такая возможность зачастую упрощает написание тестов для отдельных классов.

Тестовое покрытие

Как же приступить к внесению изменений в унаследованный проект? Прежде всего иметь тесты, окружающие участки кода, где вносятся изменения, намного безопаснее, если, конечно, такая возможность предоставляется. Изменяя код, мы можем внести в него ошибки, ведь все мы в конце концов люди. Но когда мы покрываем код тестами перед его изменением, то скорее сумеем выловить те ошибки, которые мы совершили.

На рис. 2.1 схематически показан небольшой ряд классов. Нам нужно внести изменения в метод `getResponseText` (получить текст ответной реакции) класса `InvoiceUpdateResponder` (элемент, реагирующий на обновление счета-фактуры) и в метод `getValue` (получить значение) класса `Invoice` (счет-фактура). Эти методы являются точками изменения кода. Мы можем покрыть их, написав тесты для классов, к которым они принадлежат.

Для того чтобы написать и выполнить тесты, придется создать экземпляры классов `InvoiceUpdateResponder` и `Invoice` в средствах тестирования. А сможем ли мы это сделать? Похоже, что создать экземпляр класса `Invoice` будет нетрудно. Ведь у него имеется конструктор, не принимающий никаких аргументов. А вот создать экземпляр класса `InvoiceUpdateResponder` будет труднее. Его конструктор воспринимает в качестве аргумента `DBConnection` связь с реальной базой данных. Как учесть эту ситуацию в тесте? Следует ли нам ввести в базу данные для наших тестов? Все это потребует немалого труда. Насколько тестирование с помощью базы данных замедлит проверку кода? В данный момент база данных нас не особенно интересует. Ведь для нас главное — охватить тестами изменения в классах `InvoiceUpdateResponder` и `Invoice`. Еще одно серьезное затруднение связано с тем, что для конструктора класса `InvoiceUpdateResponder` требуется `InvoiceUpdateServlet` (сервлет обновления счета-фактуры) в качестве аргумента. Насколько просто будет создать один из этих экземпляров? Мы могли бы изменить код, чтобы он больше не воспринимал сервлет. Так, если экземпляру класса `InvoiceUpdateResponder` нужна какая-то информация от сервлета `InvoiceUpdateServlet`, то мы можем передать именно ее вместо всего сервлета в целом, но требуется ли тест для того, чтобы убедиться в правильности такого изменения?

Все эти затруднения связаны с зависимостями. Если классы зависят напрямую от объектов, которые трудно охватить тестами, то видоизменить их трудно и работать с ними нелегко.

Зависимость относится к одним из самых сложных проблем разработки программного обеспечения. Большая часть унаследованного кода, с которым приходится работать, требует разрывания зависимостей, чтобы упростить изменение такого кода.

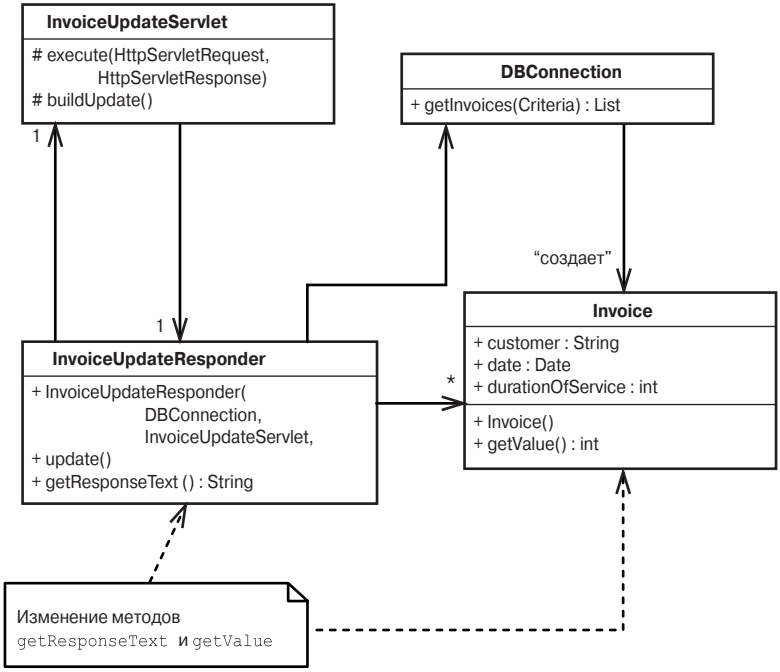


Рис. 2.1. Классы обновления счета-фактуры

Как же нам поступить? Как нам разместить тесты на своих местах, не изменяя код? Как ни прискорбно, но во многих случаях это трудно осуществимо на практике, а иногда и просто невозможно. В рассмотренном выше примере можно попытаться каким-то образом обойти затруднение, связанное со связью `DBConnection` с реальной базой данных, но как быть с сервлетом? Следует ли нам создать полноценный сервлет и передать его конструктору класса `InvoiceUpdateResponder`? Сможем ли мы тогда достичь правильного состояния? Вряд ли это возможно. А что бы мы делали, если бы работали с приложением для настольной системы с графическим пользовательским интерфейсом? У нас не было бы программного интерфейса. Вся логика была бы связана непосредственно с классами графического пользовательского интерфейса. Что же тогда нам делать?

Дилемма унаследованного кода

При изменении кода тесты должны находиться на своих местах. А для размещения тестов на их места зачастую приходится изменять сам код.

В примере с классом `Invoice` мы можем попытаться выполнить тестирование на более высоком уровне. Если писать тесты, не изменяя конкретный класс, трудно, то иногда проще проверить класс, который его использует, но в любом случае нам придется разорвать в каком-то месте зависимости между классами. В данном случае мы можем разорвать зависимость от класса `InvoiceUpdateServlet`, передав классу `InvoiceUpdateResponder` то, что действительно ему требуется, а именно — совокупность идентификационных номеров, содержащихся в классе `InvoiceUpdateServlet`. Кроме того, мы можем разорвать зависимость класса `InvoiceUpdateResponder` от класса `DBConnection`, внедрив интерфейс (`IDBConnection`) и внеся изменения в класс `InvoiceUpdateResponder`, что-

бы использовать в нем интерфейс вместо связи с реальной базой данных. На рис. 2.2 показано состояние упомянутых классов после внесения подобных изменений.

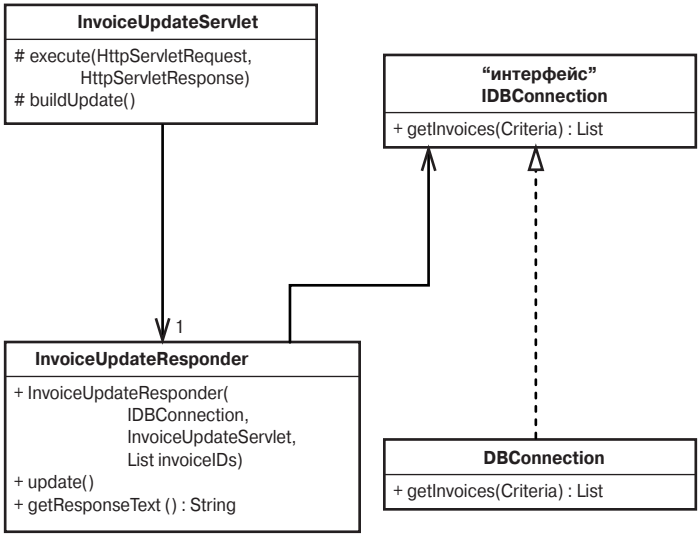


Рис. 2.2. Классы обновления счета-фактуры с разорванными зависимостями

Насколько безопасно проводить такую реорганизацию кода без тестирования? Она может быть безопасной. Такие виды реорганизации кода называются *примитивизацией параметра* и *извлечением интерфейса* соответственно. Они описываются в главе 25, посвященной способам разрыва зависимостей. Когда мы разрываем зависимости, то зачастую можем писать тесты, делающие более безопасными радикальные изменения. Главное, чтобы эти начальные виды реорганизации кода делались весьма осторожно.

Соблюдение осторожности — самая подходящая мера против возможного внесения ошибок, но иногда при разрыве зависимостей для покрытия кода это получается не так аккуратно, как в предыдущем примере. Мы можем, в частности, ввести в методы параметры, которые нестрого обязательны в выходном коде, или же разделить классы не совсем обычным образом, чтобы иметь возможность разместить тесты по местам. В подобных случаях код на таких участках может стать несколько хуже, чем изначально. И если бы мы проявили меньшую осторожность, то сразу же исправили бы подобную оплошность. Мы, конечно, можем это сделать, но все зависит от степени риска, который с этим связан. Если ошибки вызывают серьезные осложнения, а зачастую так и бывает, то целесообразно соблюдать осторожность.

При разрыве зависимостей в унаследованном коде приходится немного сдерживать в себе стремление к эстетичности. Одни зависимости разрываются отчетливо, а другие выглядят не так идеально с эстетической точки зрения. Они похожи на надрезы в хирургии, после которых могут остаться шрамы, хотя все внутри станет, скорее всего, лучше.

Если вокруг точки предполагаемого разрыва зависимостей допускается покрытие кода, то в дальнейшем можно загладить и нанесенный шрам.

Алгоритм изменения унаследованного кода

Ниже представлены основные стадии реализации алгоритма для внесения изменений в базу унаследованного кода.

1. Определение точек изменения.
2. Нахождение тестовых точек.
3. Разрывание зависимостей.
4. Написание тестов.
5. Внесение изменений и реорганизация кода.

Типичная цель, преследуемая в унаследованном коде, состоит во внесении изменений, но не каких угодно. Нам требуются функциональные изменения, имеющие определенную ценность и в то же время позволяющие проверить большую часть системы. В конце каждой стадии программирования мы должны быть в состоянии указать не только на код, обеспечивающий новое свойство, но и на соответствующие тесты. Со временем проверенные участки базы кода поднимутся на поверхность, как острова в океане. Работать на таких участках станет легче. А в конечном итоге островки охваченного тестами кода превратятся в крупные участки земли и даже в целые материки.

Итак, рассмотрим каждую из перечисленных выше стадий и укажем на те места книги, где они подробно разъясняются.

Определение точек изменения

Места, где требуется внести изменения, напрямую зависят от конкретной архитектуры. Если вы недостаточно хорошо знаете архитектуру программного обеспечения, чтобы ясно почувствовать, что вы вносите изменения в нужном месте, то обратитесь к материалу глав 16 и 17.

Нахождение тестовых точек

Иногда найти места для написания тестов нетрудно, но в унаследованном коде это нередко оказывается затруднительно. В связи с этим обратитесь к материалу глав 11 и 12. В этих главах предлагаются методы, применяемые для определения мест для написания тестов, предназначенных для конкретных изменений.

Разрывание зависимостей

Зависимости являются самой очевидной помехой для тестирования. Данное затруднение проявляется двояким образом — при получении экземпляров объектов в средствах тестирования и при выполнении методов этими средствами. Зачастую зависимости приходится разрывать в унаследованном коде для размещения тестов по местам. В идеальном случае у нас бывают тесты, способные указать на осложнения, которые могут возникнуть в связи с разрыванием зависимостей, но чаще всего такая возможность отсутствует. Практические приемы, позволяющие сделать безопасными первоначальные шаги при вмешательстве в систему для ее тестирования, приведены в главе 23. После этого обратитесь к материалу глав 9 и 10, где представлены ситуации, показывающие, как обойти типичные

затруднения, связанные с зависимостями. В этих главах делаются многочисленные ссылки на способы разрывания зависимостей, приведенные в главе 25, хотя рассматриваются не все эти способы. Поэтому уделите время изучению материала главы 25, чтобы получить более полное представление о разрыве зависимостей.

Затруднения, связанные с зависимостями, проявляются также в том случае, когда у нас возникает мысль о тесте, но написать его нам непросто. Если у вас возникнут затруднения с написанием тестов из-за зависимостей в крупных методах, обратитесь к материалу главы 22. Если же вы в состоянии разорвать зависимости, но для составления тестов вам требуется слишком много времени, обратитесь к материалу главы 7. В этой главе описываются дополнительные шаги по разрыванию зависимостей, которые вы можете предпринять для того, чтобы ускорить в среднем составление тестов.

Написание тестов

На мой взгляд, написание тестов в унаследованном коде несколько отличается от их написания в новом коде. Подробнее о роли тестов в работе с унаследованным код вы можете узнать из материала главы 13.

Внесение изменений и реорганизация кода

Я лично сторонник разработки посредством тестирования (TDD) для ввода новых свойств в унаследованный код. Описание разработки посредством тестирования и ряда других способов ввода новых свойств приведено в главе 8. После внесения изменений в унаследованный код мы нередко лучше начинаем понимать его недостатки, а написанные нами тесты для ввода новых свойств зачастую служат нам надежным прикрытием для реорганизации кода. В главах 20, 21 и 22 представлены многие методы, применяемые для усовершенствования структуры унаследованного кода. Не следует, однако, забывать, что шаги, описываемые в этих главах, являются лишь первоначальными. Они не показывают, как добиться идеальной, отчетливой и обогащенной шаблонами структуры программного обеспечения. На эту тему имеется немало другой литературы, и если у вас есть возможность применить описанные в ней методы, то непременно воспользуйтесь ими. В указанных главах поясняется, как усовершенствовать структуру программного обеспечения, хотя это зависит от контекста и подразумевает ряд шагов, которые предпринимаются для того, чтобы сделать более удобным сопровождение программного обеспечения. Эту работу не стоит недооценивать. Ведь нередко такие простые приемы, как разделение крупного класса для упрощения работы с ним, играют значительную роль в разработке приложений, несмотря на их несколько механический характер.

Остальная часть книги

В остальной части этой книги показывается, как вносить изменения в унаследованный код. В двух последующих главах представлен основной материал о трех главных принципах работы с унаследованным кодом: распознавании, разделении и швах.

Глава 3

Распознавание и разделение

В идеальном случае для того, чтобы приступить к работе с классом, ничего особенного не требуется: мы можем создать объекты любого класса, написать для них тесты, проверить их в среде тестирования, а затем перейти к другой работе. Но если бы это было так просто, то не нужно было бы писать на данную тему книгу. К сожалению, в реальной ситуации все зачастую оказывается намного сложнее. Зависимости между классами способны существенно затруднить получение конкретных групп объектов для тестирования. Так, если нам нужно создать для проверки объект одного класса, то для этого нам могут потребоваться объекты другого класса, а для них — объекты еще одного класса и т.д. В конечном счете, в среде тестирования оказывается едва ли не вся система. Для одних языков программирования это не так важно, как для других, особенно C++, где практически нельзя ускорить цикл обработки только из-за одного времени компоновки, если не разорвать зависимости.

В системах, которые не разрабатывались параллельно с блочными тестами, нам нередко приходится разрывать связи, чтобы ввести классы в среду тестирования, но это не единственная причина для разрыва зависимостей. Иногда тестируемый класс оказывает воздействие на другие классы, и этот факт необходимо учитывать в тестах. В одних случаях мы можем распознать такие воздействия с помощью интерфейса другого класса, а в других случаях — нет. И тогда у нас остается только одна возможность: сымитировать другой класс, чтобы распознать воздействия непосредственно.

Как правило, при размещении тестов по местам появляются две причины для разрыва зависимостей — *распознавание* и *разделение*.

1. **Распознавание.** Мы разрываем зависимости для *распознавания*, если не можем получить доступ к значениям, которые вычисляет код.
2. **Разделение.** Мы разрываем зависимости для *разделения*, если не можем ввести в среду тестирования даже фрагмент кода для выполнения.

Рассмотрим следующий пример. Допустим, что у нас имеется класс `NetworkBridge` (сетевой мост) в приложении для управления сетью.

```
public class NetworkBridge
{
    public NetworkBridge(EndPoint [] endpoints) {
        ...
    }

    public void formRouting(String sourceID, String destID) {
        ...
    }
    ...
}
```

Класс `NetworkBridge` воспринимает массив конечных точек (`EndPoint`) и управляет их конфигурацией, используя определенную местную сетевую аппаратуру. Пользователи класса `NetworkBridge` могут применять его методы для маршрутизации трафика из од-

ной конечной точки в другую. Для этого в классе `NetworkBridge` изменяются настройки, хранящиеся в классе `EndPoint`. Каждый экземпляр класса `EndPoint` открывает сокет и связывается по сети с конкретным устройством.

Это лишь краткое описание функций класса `NetworkBridge`. Его можно детализировать, но с точки зрения тестирования уже вполне очевидны некоторые затруднения. Если нам требуется написать тесты для класса `NetworkBridge`, то как нам это сделать? Ведь при построении этого класса вполне возможны обращения к реальной сетевой аппаратуре. Требуется ли такая аппаратура для создания экземпляра данного класса? Более того, как нам узнать, что делает сетевой мост с этой аппаратурой или конечными точками сети? Ведь с нашей точки зрения данный класс — это закрытый черный ящик.

Возможно, дело обстоит не так уж и плохо. Так, мы можем написать определенный код для анализа пакетов в сети или же получить доступ к некоторой сетевой аппаратуре для обращения к ней со стороны класса `NetworkBridge`, чтобы он, по крайней мере, не перешел в замороженное состояние, когда мы попытаемся создать его экземпляр. А возможно, для написания и выполнения тестов в нашем распоряжении окажется совокупность локальных конечных точек. Все эти варианты оказываются вполне работоспособными, но для их реализации потребуется приложить немало труда. Для логики, которую нам нужно изменить в классе `NetworkBridge`, возможно, ничего из этого не потребуется, просто потому что мы не в состоянии овладеть ею. Мы не можем выполнить объект данного класса и проверить непосредственно, как он работает.

Данный пример наглядно иллюстрирует обе проблемы распознавания и разделения. Мы не можем распознать результаты наших вызовов методов данного класса и не в состоянии выполнить его отдельно от остальной части приложения.

Какая из этих проблем сложнее — распознавание или разделение? Ясного ответа на этот вопрос не существует. Как правило, нам требуется и то и другое, причем и в том и другом случае у нас появляются веские основания для разрыва зависимостей. Но ясно одно: программное обеспечение может быть разделено разными способами. В действительности, для этой цели существует целый ряд способов, приведенных в главе 25, специально посвященной данной теме. А для распознавания имеется одно основное средство: имитация взаимодействующих объектов.

Имитация взаимодействующих объектов

Одна из самых больших трудностей, возникающих в работе с унаследованным кодом, связана с зависимостью. Если нам требуется выполнить фрагмент кода отдельно, чтобы посмотреть, что он делает, то для этого нам зачастую приходится разрывать его зависимости от другого кода. Но сделать это не так-то просто. Ведь другой код нередко оказывается единственным местом для распознавания результатов наших действий. Если мы можем поместить в этом месте другой код и протестировать его, значит, у нас есть возможность написать тесты. В объектно-ориентированном программировании такого рода другие фрагменты кода нередко называются *фиктивными объектами*.

Фиктивные объекты

Фиктивным называется такой объект, который имитирует некоторый объект, взаимодействующий с тестируемым классом. Рассмотрим следующий пример. Допустим, что в системе для автоматизации учета в розничной торговле имеется класс `Sale` (продажа;

рис. 3.1). У него имеется метод `scan()`, воспринимающий штриховой код товара, который хочет приобрести покупатель. Всякий раз, когда вызывается метод `scan()`, объекту класса `Sale` требуется отобразить на индикаторе кассового аппарата название товара со сканированным штриховым кодом наряду с его ценой.

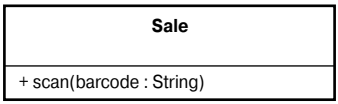


Рис. 3.1. Класс `Sale`

Как нам проверить правильность отображения текста на индикаторе кассового аппарата? Если вызовы интерфейса API для индикатора кассового аппарата глубоко скрыты в классе `Sale`, то сделать это будет непросто. В частности, нелегко распознать результат вывода на индикатор кассового аппарата. Но если мы найдем место в коде, где вывод на этот индикатор обновляется, то сможем перейти к конструкции, приведенной на рис. 3.2.

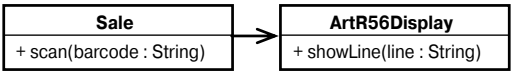


Рис. 3.2. Связь класса `Sale` *с классом отображения результата*

В данном случае мы ввели новый класс `ArtR56Display`. Этот класс содержит весь код, необходимый для обращения к конкретному устройству отображения. Нам остается только снабдить его строкой текста, содержащей результат, который нам требуется отобразить. Весь код отображения мы можем перенести из класса `Sale` в класс `ArtR56Display`, и при этом система будет делать то же, что и прежде. А что это нам даст? Сделав это, мы можем, пожалуй, перейти к конструкции, приведенной на рис. 3.3.

Теперь класс `Sale` может опираться на класс `ArtR56Display` или нечто другое, например, на класс `FakeDisplay` (фиктивное отображение). Достоинство такого фиктивного отображения заключается в том, что у нас появилась возможность писать для него тесты, чтобы выяснить функции класса `Sale`.

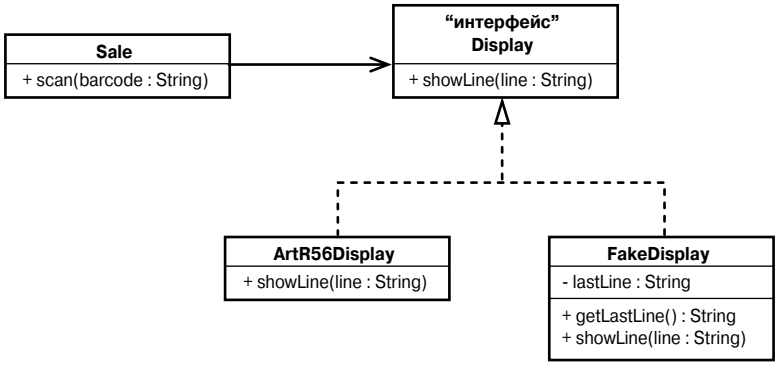


Рис. 3.3. Класс `Sale` *с иерархией классов отображения*

Как все это работает? Класс `Sale` воспринимает отображение, которое представляет собой объект любого класса, реализующий интерфейс отображения `Display`.

```
public interface Display
{
    void showLine(String line);
}
```

Оба класса `ArtR56Display` и `FakeDisplay` реализуют интерфейс `Display`.

Объект `Sale` может воспринимать отображение с помощью конструктора и опираться на него внутренним образом.

```
public class Sale
{
    private Display display;

    public Sale(Display display) {
        this.display = display;
    }

    public void scan(String barcode) {
        ...
        String itemLine = item.name()
            + " " + item.price().asDisplayText();
        display.showLine(itemLine);
        ...
    }
}
```

Код в методе `scan` вызывает метод `showLine` (отобразить строку) объекта `display`. Но происходящее далее зависит от вида отображения, переданного объекту `Sale` при его создании. Так, если мы передали ему объект класса `ArtR56Display`, то он попытается отобразить результат на индикаторе реального кассового аппарата. А если мы передали ему объект класса `FakeDisplay`, то он не сделает этого, но мы сумеем увидеть то, что он мог бы отобразить. Для этой цели воспользуемся приведенным ниже тестом.

```
import junit.framework.*;

public class SaleTest extends TestCase
{
    public void testDisplayAnItem() {
        FakeDisplay display = new FakeDisplay();
        Sale sale = new Sale(display);

        sale.scan("1");
        assertEquals("Milk $3.99", display.getLastLine());
    }
}
```

У класса `FakeDisplay` имеются свои незначительные особенности. Он выглядит следующим образом.

```
public class FakeDisplay implements Display
{
```

```
    private String lastLine = "";

    public void showLine(String line) {
        lastLine = line;
    }

    public String getLastLine() {
       return lastLine;
    }
}
```

Метод `showLine` воспринимает текстовую строку и присваивает ее переменной `lastLine`. А метод `getLastLine` (получить последнюю строку) возвращает эту текстовую строку всякий раз, когда он вызывается. Такое, на первый взгляд, незначительное поведение оказывает существенную помощь. Благодаря написанному нами тесту мы можем выяснить, выводится ли на отображение правильный текст при использовании класса `Sale`.

Фиктивные объекты, поддерживающие настоящие тесты

Узнав о применении фиктивных объектов при тестировании кода, некоторые прямо заявляют, что это ненастоящее тестирование. Ведь оно не показывает реальный результат, т.е. что именно отображается в действительности на экране индикатора в приведенном выше примере. Допустим, что часть программного обеспечения для вывода информации на индикатор кассового аппарат работает неправильно. Такой тест никогда не выявит подобный недостаток. С таким выводом нельзя не согласиться, но это совсем не означает, что данный тест не является настоящим. Даже если бы нам удалось придумать тест, точно показывающий, какие именно пиксели активизированы на экране индикатора кассового аппарата, означает ли это, что данное программное обеспечение будет работать со всей остальной аппаратурой? Нет, конечно, но это и не означает, что данный тест ничего не проверяет. Когда мы пишем тесты, то должны действовать по принципу "разделяй и властвуй". Упомянутый выше тест просто сообщает нам, каким образом объекты класса `Sale` воздействуют на устройства отображения. Но такое тестирование нельзя считать несущественным. Если мы обнаружим программную ошибку, то, выполнив подобный тест, сможем лучше понять, что дело совсем не в классе `Sale`. Используя подобную информацию как вспомогательное средство для локализации ошибок, мы можем сэкономить немало драгоценного времени.

Когда мы пишем тесты для отдельных структурных единиц кода, то в конечном счете получаем небольшие вполне понятные фрагменты кода. Это облегчает нам осмысление самого кода.

Две стороны фиктивных объектов

Поначалу применение фиктивных объектов может вызвать определенные затруднения. Самое необычное в таких объектах — наличие у них двух "сторон". Для примера обратимся еще раз к классу `FakeDisplay`, как показано на рис. 3.4.

Метод `showLine` требуется в классе `FakeDisplay`, поскольку в нем реализуется интерфейс `Display`. Это единственный метод интерфейса `Display`, видимый в классе `Sale`. А другой метод, `getLastLine`, служит для тестирования. Именно поэтому мы объявляем `display` как объект класса `FakeDisplay`, а не интерфейса `Display`.

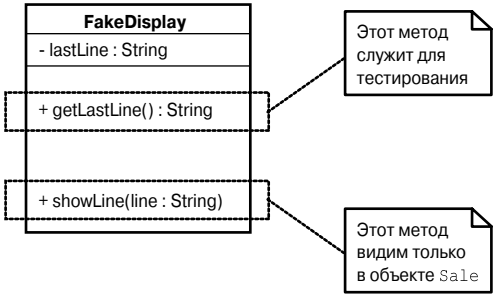


Рис. 3.4. Две стороны фиктивного объекта

```
import junit.framework.*;

public class SaleTest extends TestCase
{
    public void testDisplayAnItem() {
        FakeDisplay display = new FakeDisplay();
        Sale sale = new Sale(display);

        sale.scan("1");
        assertEquals("Milk $3.99", display.getLastLine());
    }
}
```

Класс `Sale` будет видеть фиктивное отображение только в виде интерфейса `Display`, но в тесте мы должны опираться на объект класса `FakeDisplay`. В противном случае мы не сможем вызвать метод `getLastLine()`, чтобы выяснить, что же отображается в момент продажи товара.

Реализация фиктивных объектов

Приведенный выше пример довольно прост, но он ясно показывает главный принцип, положенный в основу фиктивных объектов. Они могут быть реализованы самыми разными способами. В языках объектно-ориентированного программирования фиктивные объекты зачастую реализуются в виде простых классов, подобных классу `FakeDisplay` в приведенном выше примере. А в языках, не относящихся к категории объектно-ориентированного программирования, фиктивный объект можно реализовать в виде определяемой альтернативной функции, записывающей значения в некоторой глобальной структуре данных, доступной в тестах. Подробнее об этом речь пойдет в главе 19.

Имитирующие объекты

Фиктивные объекты легко создаются и являются весьма ценным средством для распознавания. Если же их приходится создавать в большом количестве, то целесообразно рассмотреть другую возможность — использовать более совершенный тип фиктивных объектов, называемых *имитирующими*. Такие объекты выполняют утверждение внутренним образом. Ниже приведен пример теста, в котором используется имитирующий объект.

```
import junit.framework.*;

public class SaleTest extends TestCase
{
    public void testDisplayAnItem() {
        MockDisplay display = new MockDisplay();
        display.setExpectation("showLine", "Milk $3.99");
        Sale sale = new Sale(display);
        sale.scan("1");
        display.verify();
    }
}
```

В данном тесте создается объект, имитирующий отображение. Замечательной особенностью имитирующих объектов является возможность указать им сначала, какие именно вызовы следует ожидать, а затем предписать им проверить, получили ли они эти вызовы. Именно это и происходит в приведенном выше тесте. Мы указываем имитирующему объекту `display` ожидать вызова его метода `showLine` с аргументом `"Milk $3.99"`. Задав ожидание этого вызова, мы просто используем имитирующий объект внутри теста. В данном случае мы вызываем метод `scan()`. В дальнейшем мы вызываем метод `verify()`, который проверяет, оправдались ли все ожидания. Если они не оправдались, то можно считать, что тест не прошел.

Имитирующие объекты являются весьма эффективным средством, и поэтому существуют самые разные структуры имитирующих объектов. Но структуры имитирующих объектов доступны не во всех языках программирования, и поэтому их, как правило, заменяют простые фиктивные объекты.

Глава 4

Модель шва

При попытке написать тесты для существующего кода нельзя не заметить, насколько плохо код подходит для тестирования. И дело не в самих программах или языках программирования. Как правило, языки программирования не имеют достаточных средств для поддержки тестирования. И по-видимому, для того чтобы упростить тестирование программ, остается лишь одно из двух — писать тесты во время разработки этих программ или же тратить дополнительное время на "проектирование ради тестируемости". Немало надежд обычно связывают с первым подходом, но если бо́льшая часть прикладного кода уже имеется, то такой подход не приносит заметного успеха.

По себе я заметил, что при попытке подвергнуть код тестированию я начинаю осмысливать этот код совсем по-другому. Поначалу я приписывал это своему субъективному отношению к коду, но затем я обнаружил, что такой взгляд на код под иным углом зрения помогает мне в работе с новыми и неизвестными мне языками программирования. В этой книге нет места, чтобы охватить все языки программирования, и поэтому я решил указать на свой особый взгляд на код именно здесь, надеясь, что это поможет вам так же, как помогло и мне.

Программа как большой лист печатного текста

Когда я только начинал программировать, невольно заметил, как мне повезло, что я начал заниматься этим делом позже своих коллег, имея в своем распоряжении персональный компьютер и компилятор для выполнения на этом компьютере, тогда как многие из моих коллег и друзей начинали программировать еще во времена больших ЭВМ и перфокарт. Когда я решил изучать программирование в школе, то начал работать за терминалом в школьной вычислительной лаборатории. Я и мои соученики имели возможность компилировать свой код в удаленном режиме на мини-ЭВМ DEC VAX. Для учета машинного времени на этой ЭВМ существовала специальная система. Каждая компиляция стоила определенную сумму денег, вычитывавшуюся из нашей учетной записи, и поэтому каждую четверть нам выделялось фиксированное количество машинного времени.

В то время программы представляли собой обычные листинги. Каждые два часа мне приходилось ходить за распечаткой моей программы из вычислительной лаборатории в помещение для печати, после чего я анализировал свои ошибки по распечатанному листингу программы. Тогда у меня еще не было достаточных знаний, чтобы уделять внимание модульности кода. Нам приходилось писать модульный код, чтобы показать, что мы умеем это делать, но в то время меня больше интересовало получение правильных результатов с помощью кода. А когда я обратился к написанию объектно-ориентированного кода, то модульность кода уже считалась слишком академическим понятием. Мне уже не нужно было ходить из одной классной комнаты в другую по ходу выполнения своего школьного задания. Но когда я стал профессионально заниматься программированием, то начал уделять больше внимания подобным вещам, хотя в школе программа была для меня не

более чем листингом, т.е. длинным перечнем функций, которые мне приходилось писать и разбираться в них по очереди.

Такой взгляд на программу как на листинг мне кажется правильным — по крайней мере, с точки зрения отношения к своим программам тех, кто их пишет. Если бы мы, вообще ничего не зная о программировании, заглянули в помещение, где работают программисты, то могли бы подумать, что это какие-то школяры, изучающие и правящие крупные и очень важные документы. Ведь программа очень похожа на большой лист печатного текста. Достаточно поправить текст хотя бы немного, чтобы изменить смысл всего документа, поэтому изменения в него вносятся очень осторожно во избежание ошибок.

Все это, в общем, справедливо, но как же модульность? Нам всегда говорили, что программы писать лучше небольшими неоднократно используемыми фрагментами, но как часто такие фрагменты используются многократно и независимо? Не очень часто, поскольку использовать их неоднократно не так-то просто. Даже если отдельные фрагменты программного обеспечения выглядят независимыми, они зачастую зависят один от другого — хотя бы и незначительно.

Швы

Когда мы делаем первые попытки извлечь из существующего кода отдельные классы для блочного тестирования, нам зачастую приходится разрывать немало зависимостей. Любопытно, что для этого приходится немало потрудиться, независимо от того, насколько хорошо разработана программа. Извлечение отдельных классов из существующего кода для тестирования совершенно меняет наше представление о том, что означает хорошо разработанная программа. Это заставляет нас также взглянуть на программное обеспечение под совершенно иным углом зрения. И представление о программе как о большом листе печатного текста для этого уже не подходит. Как же нам рассматривать ее? Обратимся к примеру функции, написанной на C++.

```
bool CAsyncSslRec::Init()
{
    if (m_bSslInitialized) {
        return true;
    }
    m_smutex.Unlock();
    m_nSslRefCount++;

    m_bSslInitialized = true;

    FreeLibrary(m_hSslDll1);
    m_hSslDll1=0;
    FreeLibrary(m_hSslDll2);
    m_hSslDll2=0;

    if (!m_bFailureSent) {
        m_bFailureSent=TRUE;
        PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
    }

    CreateLibrary(m_hSslDll1,"syncesel1.dll");
```

```
    CreateLibrary(m_hSslDll2,"syncesel2.dll");

    m_hSslDll1->Init();
    m_hSslDll2->Init();

    return true;
}
```

Этот фрагмент кода очень похож на большой лист печатного текста, не так ли? Допустим, что нам требуется выполнить весь этот код, за исключением следующей строки:

```
PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
```

Как же нам это сделать? Очень просто. Для этого достаточно удалить приведенную выше строку из данного фрагмента кода.

А теперь немного конкретизируем задачу. Нам нужно избежать выполнения указанной строки кода, поскольку `PostReceiveError` — это глобальная функция, связывающаяся с другой подсистемой, которую очень трудно подвергнуть тестированию. Итак, наша задача заключается в следующем: как нам выполнить приведенную выше функцию (или метод), не вызывая во время тестирования функцию `PostReceiveError`? Как нам сделать это, разрешив в то же время вызов функции `PostReceiveError` в выходном коде?

Для меня лично это вопрос со многими возможными ответами, что приводит нас к понятию шва.

Ниже приведено определение шва. Рассмотрим его сначала теоретически, а затем на нескольких примерах.

Шов

Шов — место, где можно изменить поведение программы, не правя ее в этом месте.

Имеется ли шов в вызове функции `PostReceiveError`? Да, имеется. Мы можем избавиться от такого поведения двумя способами. Рассмотрим сначала самый простой способ. `PostReceiveError` — это глобальная функция, а не часть класса `CAsynchSslRec`. Что, если ввести метод с такой же, как и у данной функции сигнатурой, в класс `CAsynchSslRec`?

```
class CAsyncSslRec
{
    ...
    virtual void PostReceiveError(UINT type, UINT errorcode);
    ...
};
```

В файле реализации мы можем ввести следующее тело данного метода:

```
void CAsyncSslRec::PostReceiveError(UINT type, UINT errorcode)
{
    ::PostReceiveError(type, errorcode);
}
```

Такое изменение должно сохранить поведение. Мы используем этот новый метод, чтобы делегировать полномочия глобальной функции `PostReceiveError` с помощью оператора разрешения контекста (`::`) в C++. Несмотря на несущественную косвенность в этом коде, мы в конечном итоге вызываем ту же самую глобальную функцию.

А что, если выполнить теперь подклассификацию класса `CAsynchSslRec` и переопределить метод `PostReceiveError`?

```
class TestingAsyncSslRec : public CAsyncSslRec
{
    virtual void PostReceiveError(UINT type, UINT errorcode)
    {
    }
};
```

Если мы сделаем это и вернемся к тому месту, откуда мы начинали создание объекта `CAsynchSslRec`, создав вместо него объект `TestingAsyncSslRec`, то, по сути, аннулируем поведение вызова функции `PostReceiveError` в следующем коде.

```
bool CAsyncSslRec::Init()
{
    if (m_bSslInitialized) {
        return true;
    }
    m_smutex.Unlock();
    m_nSslRefCount++;

    m_bSslInitialized = true;

    FreeLibrary(m_hSslDll1);
    m_hSslDll1=0;
    FreeLibrary(m_hSslDll2);
    m_hSslDll2=0;

    if (!m_bFailureSent) {
        m_bFailureSent=TRUE;
        PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
    }

    CreateLibrary(m_hSslDll1,"syncesel1.dll");
    CreateLibrary(m_hSslDll2,"syncesel2.dll");

    m_hSslDll1->Init();
    m_hSslDll2->Init();

    return true;
}
```

Теперь мы можем писать тесты для этого кода без каких-либо неприятных побочных эффектов.

Такой шов называется *объектным*. Мы сумели изменить вызываемый метод, не меняя вызывающий его метод. Объектные швы доступны в объектно-ориентированных языках программирования и являются одним из множества различных типов швов.

А почему, собственно, швы? И для чего это понятие пригодно?

Одно из самых больших затруднений при подготовке унаследованного кода к тестированию вызывает разрыв зависимостей. Если нам повезет, то зависимости окажутся немногочисленными и локализованными, но в патологических случаях они многочисленны

и распределены по всей базе кода. Представление программного обеспечение в виде шва помогает нам увидеть возможности, которые есть в базе кода. Если нам удастся заменить поведение в швах, то мы сможем выборочно исключить зависимости из наших тестов. Кроме того, мы можем выполнить другой код там, где эти зависимости находились, если нам требуется распознать условия в коде и написать тесты по этим условиям. Такая работа нередко помогает нам разместить достаточное количество тестов по местам, чтобы подкрепить более решительные действия.

Типы швов

В языках программирования доступны разные типы швов. Самый лучший способ их изучения — проанализировать все этапы превращения исходного текста программы в исполняемый машинный код. На каждом распознаваемом этапе выявляются разные типы швов.

Швы предварительной обработки

Во многих программных средах исходный текст программы читается компилятором. Затем компилятор порождает команды в объектном коде или же в байт-коде. В зависимости от используемого языка программирования возможны более поздние стадии обработки, а как насчет более ранних стадий?

Лишь в некоторых языках программирования имеется встроенная стадия, предшествующая компиляции. К числу этих языков относятся C и C++.

В C и C++ перед компилятором выполняется макропрепроцессор. Многие годы макропрепроцессор подвергался постоянным нападкам и проклятиям. Но с его помощью мы можем взять строки исходного текста, которые выглядят так же безобидно, как и следующие строки:

```
TEST(getBalance,Account)
{
    Account account;
    LONGS_EQUAL(0, account.getBalance());
}
```

и сделать так, чтобы они предстали перед компилятором в следующем виде:

```
class AccountgetBalanceTest : public Test
    { public: AccountgetBalanceTest () : Test ("getBalance" "Test") {}
        void run (TestResult& result_); }
    AccountgetBalanceInstance;
        void AccountgetBalanceTest::run (TestResult& result_)
{
    Account account;
{ result_.countCheck();
    long actualTemp = (account.getBalance());
    long expectedTemp = (0);
    if ((expectedTemp) != (actualTemp))
{ result_.addFailure (Failure (name_, "c:\\seamexample.cpp", 24,
StringFrom(expectedTemp),
StringFrom(actualTemp))); return; } }

}
```

Кроме того, мы можем вложить код в операторы условной компиляции, как в приведенном ниже фрагменте кода, чтобы организовать поддержку отладки на разных платформах.

```
...
m_pRtg->Adj(2.0);

#ifdef DEBUG
#ifndef WINDOWS
    { FILE *fp = fopen(TGLOGNAME,"w");
    if (fp) { fprintf(fp,"%s", m_pRtg->pszState); fclose(fp); }}
#endif

m_pTSRTable->p_nFlush |= GF_FLOT;
#endif

...
```

Чрезмерное использование предварительной обработки в выходном коде не приветствуется, поскольку это приводит к снижению ясности кода. Директивы условной компиляции (`#ifdef`, `#ifndef`, `#if` и т.д.) просто вынуждают сохранять несколько разных программ в одном и том же исходном коде. Макрокоманды (определяемые директивой `#define`) могут принести некоторую пользу, но ведь они выполняют простую замену текста. Так, очень легко создать макрокоманды, скрывающие серьезные программные ошибки, которые становятся совершенно незаметными.

Но несмотря на приведенные выше оговорки, наличие препроцессора в языках C и C++ не может не радовать, поскольку препроцессор предоставляет нам больше швов. Рассмотрим следующий пример программы на языке C, в которой имеются зависимости от библиотечной программы `db_update`. Эта программа, а точнее функция, обращается непосредственно к базе данных. Если не подставить другую реализацию этой функции, то мы не сможем распознать ее поведение.

```
#include <DFHLItem.h>
#include <DHLSRecord.h>

extern int db_update(int, struct DFHLItem *);

void account_update(
    int account_no, struct DHLSRecord *record, int activated)
{
    if (activated) {
        if (record->dateStamped && record->quantity > MAX_ITEMS) {
            db_update(account_no, record->item);
        } else {
            db_update(account_no, record->backup_item);
        }
    }
    db_update(MASTER_ACCOUNT, record->item);
}
```

Для замены вызовов функции `db_update` лучше воспользоваться швами предварительной обработки. С этой целью мы можем ввести заголовочный файл `localdefs.h` следующим образом.

```
include <DFHLItem.h>
#include <DHLSRecord.h>

extern int db_update(int, struct DFHLItem *);

#include "localdefs.h"

void account_update(
    int account_no, struct DHLSRecord *record, int activated)
{
    if (activated) {
        if (record->dateStamped && record->quantity > MAX_ITEMS) {
            db_update(account_no, record->item);
        } else {
            db_update(account_no, record->backup_item);
        }
    }
    db_update(MASTER_ACCOUNT, record->item);
}
```

Внутри этого файла мы можем предоставить определение функции `db_update` и ряда полезных для нас переменных следующим образом.

```
#ifdef TESTING
...
struct DFHLItem *last_item = NULL;
int last_account_no = -1;

#define db_update(account_no,item)\
    {last_item = (item); last_account_no = (account_no);}
...
#endif
```

После этой замены функции `db_update` можно написать тесты, чтобы проверить, вызывается ли она с правильными параметрами. У нас есть такая возможность, поскольку директива `#include` препроцессора C предоставляет нам шов, позволяющий заменить исходный текст программы перед его компиляцией.

Швы предварительной обработки довольно эффективны. Я не уверен, что мне может потребоваться препроцессор для Java или других более современных языков программирования, но иметь такое средство в C и C++ очень полезно, чтобы как-то возместить ряд других недостатков, которые у них имеются в отношении тестирования кода.

Хотя это и не упоминалось раньше, но в отношении швов следует также иметь в виду следующее: у каждого шва имеется *разрешающая точка*. Приведем определение шва еще раз.

Шов

Шов — место, где можно изменить поведение программы, не правя ее в этом месте.

Если имеется шов, то имеется и место, где можно изменить поведение. Но мы не можем перейти в это место и изменить код только для его тестирования. Исходный код должен оставаться одинаковым как для эксплуатации, так и для тестирования. В предыдущем примере нам требовалось изменить поведение в исходном тексте вызова функции `db_update`. Для того чтобы воспользоваться данным швом, придется сделать изменение где-нибудь

еще. В данном случае в качестве разрешающей точки служит директива препроцессора `#define` под названием `TESTING`. Если определено тестирование (`TESTING`), то в заголовочном файле `localdefs.h` определяются макрокоманды, заменяющие вызов функции `db_update` в исходном коде.

Разрешающая точка

У каждого шва имеется разрешающая точка — место, где можно выбрать то или иное поведение.

Компоновочные швы

Во многих языковых системах компиляция не является самой последней стадией процесса формирования кода. Компилятор порождает промежуточное представление кода, в котором содержатся вызовы кода из других файлов. Такие представления кода объединяются компоновщиками, которые разрешают каждый вызов, чтобы полностью скомпоновать программу для выполнения.

В таких языках программирования, как C и C++, имеется отдельный компоновщик, выполняющий описанную выше операцию. В Java и аналогичных языках процесс компоновки выполняется компилятором подспудно. Если в исходном файле содержится оператор `import`, то компилятор проверяет, действительно ли скомпилирован импортируемый класс. Если же этот класс не скомпилирован, то он компилируется при необходимости, после чего компилятор проверяет, насколько правильно будут разрешены все вызовы данного класса во время выполнения.

Независимо от алгоритма, применяемого в конкретном языке программирования для разрешения ссылок, им можно воспользоваться для замены фрагментов программы. Рассмотрим пример кода на языке Java. Ниже приведен небольшой класс `FitFilter` (фильтрация на пригодность).

```
package fitnesse;

import fit.Parse;
import fit.Fixture;

import java.io.*;
import java.util.Date;

import java.io.*;
import java.util.*;

public class FitFilter {

    public String input;
    public Parse tables;
    public Fixture fixture = new Fixture();
    public PrintWriter output;

    public static void main (String argv[]) {
        new FitFilter().run(argv);
   }
```

```
    public void run (String argv[]) {
        args(argv);
        process();
        exit();
    }

    public void process() {
        try {
            tables = new Parse(input);
            fixture.doTables(tables);
        } catch (Exception e) {
            exception(e);
        }
        tables.print(output);
    }
    ...
}
```

В этом файле импортируются классы `fit.Parse` и `fit.Fixture`. Как найти эти классы компилятору и виртуальной машине Java? Для определения места поиска этих классов в Java служит переменная среды `classpath`. Мы можем создать классы с такими же именами, поместить их в другой каталог и изменить путь в переменной среды `classpath`, чтобы установить связь с другими классами `fit.Parse` и `fit.Fixture`. Несмотря на известные трудности применения такого приема в выходном коде, при тестировании кода это очень удобный способ разорвать зависимости.

Допустим, что нам требуется предоставить другой вариант класса `Parse` для тестирования. Где должен находиться шов?

Швом в данном случае является вызов `new Parse` в методе `process`.

А где находится разрешающая точка?

Разрешающей точкой является переменная среды `classpath`.

Подобного рода динамическая компоновка осуществляется во многих языках программирования. В большинстве языков предоставляется в той или иной степени возможность для использования компоновочных швов. Но не всякая компоновка является динамической. Во многих старых языках программирования практически вся компоновка статическая, т.е. она происходит после компиляции.

Во многих системах сборки кода C и C++ для формирования исполняемых файлов осуществляется статическая компоновка. Зачастую самый простой способ воспользоваться компоновочным швом — создать отдельную библиотеку для любых классов или функций, которые требуется заменить. Сделав это, мы можем изменить сценарии сборки, чтобы скомпоновать библиотеку классов для тестирования вместо выходных классов. Для этого придется немного потрудиться, но затраченные усилия могут окупиться сторицей, если имеется база кода, изобилующая вызовами сторонних библиотек. Например, приложение САПР может содержать множество встроенных вызовов библиотеки графических функций. Ниже приведен типичный код из такой библиотеки.

```
void CrossPlaneFigure::rerender()
{
```

```
    // нарисовать метку
    drawText(m_nX, m_nY, m_pchLabel, getClipLen());
    drawLine(m_nX, m_nY, m_nX + getClipLen(), m_nY);
    drawLine(m_nX, m_nY, m_nX, m_nY + getDropLen());
    if (!m_bShadowBox) {
        drawLine(m_nX + getClipLen(), m_nY,
            m_nX + getClipLen(), m_nY + getDropLen());
        drawLine(m_nX, m_nY + getDropLen(),
            m_nX + getClipLen(), m_nY + getDropLen());
    }

    // нарисовать фигуру
    for (int n = 0; n < edges.size(); n++) {
        ...
    }

    ...
}
```

В этом коде имеется много прямых вызовов библиотеки графических функций. К сожалению, единственный способ проверить правильность функционирования этого кода — наблюдать на экране монитора за вычерчиваемыми фигурами. Тестирование столь сложного кода зачастую чревато ошибками, не говоря уже о том, насколько это вообще трудно сделать. Поэтому в качестве альтернативы можно воспользоваться компоновочными швами. Если все функции рисования входят в отдельную библиотеку, то вместо нее можно создать тестовую заглушку и скомпоновать ее с остальной частью приложения. Если же требуется только отделение от зависимости, то функции рисования можно сделать просто пустыми.

```
void drawText(int x, int y, char *text, int textLength)
{
}

void drawLine(int firstX, int firstY, int secondX, int secondY)
{
}
```

Если функции возвращают значения, то придется организовать возврат каких-то значений. Зачастую для этой цели подходит код, обозначающий успешное выполнение функции, или же стандартное значение соответствующего типа данных, как в приведенном ниже фрагменте кода.

```
int getStatus()
{
    return FLAG_OKAY;
}
```

Пример библиотеки графических функций не совсем типичен. Он вполне подходит для применения рассматриваемого здесь метода, поскольку демонстрирует исключительно “указательный” интерфейс. Вызывая функции, вы указываете им сделать что-то конкретное и не просите их вернуть много информации обратно. Запросить информацию намного сложнее, поскольку возвращаемые стандартные значения не годятся для проверки работоспособности кода.

Компоновочные швы часто используются по причине разделения. Они подходят и для распознавания, но в этом случае придется приложить больше труда. Так, в упомянутом выше примере имитации библиотеки графических функций мы могли бы ввести ряд дополнительных структур данных для регистрации вызовов следующим образом:

```
std::queue<GraphicsAction> actions;

void drawLine(int firstX, int firstY, int secondX, int secondY)
{
    actions.push_back(GraphicsAction(LINE_DRAW,
        firstX, firstY, secondX, secondY);
}
```

С помощью этих структур данных мы можем распознать в тесте результаты выполнения функции.

```
TEST(simpleRender,Figure)
{
    std::string text = "simple";
    Figure figure(text, 0, 0);

    figure.rerender();
    LONGS_EQUAL(5, actions.size());

    GraphicsAction action;
    action = actions.pop_front();
    LONGS_EQUAL(LABEL_DRAW, action.type);

    action = actions.pop_front();
    LONGS_EQUAL(0, action.firstX);
    LONGS_EQUAL(0, action.firstY);
    LONGS_EQUAL(text.size(), action.secondX);
}
```

Алгоритмы для распознавания результатов могут стать довольно сложными, поэтому начинать лучше всего с очень простых алгоритмов, усложняя их лишь по мере возникновения текущих потребностей в распознавании.

Разрешающая точка для компоновочных швов всегда находится за пределами исходного текста программы. Иногда она находится в сценарии сборки или развертывания. Это несколько затрудняет выявление последствий использования компоновочных швов.

Рекомендации по применению

Если вы пользуетесь компоновочными швами, то непременно убедитесь в том, что тестовая и производственная среды явно отличаются между собой.

Объектные швы

Объектные швы наиболее пригодны для применения среди всех типов швов, доступных в объектно-ориентированных языках программирования. Прежде всего следует признать, что по вызову в объектно-ориентированной программе мы не можем явно определить, ка-

кой именно метод фактически выполняется. Рассмотрим следующий пример вызова метода на языке Java:

```
cell.Recalculate();
```

На первый взгляд, в этом коде вызывается на выполнение метод под названием `Recalculate` (пересчитать). Но при выполнении программы обнаруживается не только метод с этим именем, но и нечто другое (рис. 4.1).

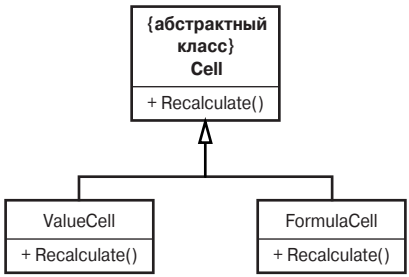


Рис. 4.1. Иерархия класса `Cell`

Какой же метод вызывается в приведенной выше строке кода? Если мы не знаем, на что указывает объект `cell`, то вряд ли мы будем знать точно, вызывается ли метод `Recalculate` класса `ValueCell` (ячейка значения) или же метод `Recalculate` класса `FormulaCell` (ячейка формулы). Это может быть даже метод `Recalculate` какого-нибудь другого класса, не наследующий от класса `Cell` (в этом случае имя `cell` очень неудачно выбрано для данной переменной). Если мы сумеем заменить вызов конкретного метода `Recalculate` в рассматриваемой здесь строке кода, не меняя окружающий ее код, то данный вызов может считаться швом.

В объектно-ориентированных языках программирования не все вызовы методов являются швами. Ниже приведен пример вызова, не являющегося швом.

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet() {
        ...
        Cell cell = new FormulaCell(this, "A1", "=A2+A3");
        ...
        cell.Recalculate();
        ...
    }
    ...
}
```

В этом фрагменте кода создается объект `cell`, который затем используется в том же самом методе. Является ли вызов метода `Recalculate` объектным швом? Нет, не является, поскольку у него нет разрешающей точки. Мы не можем заменить вызов конкретного метода `Recalculate`, поскольку выбор зависит от класса, к которому принадлежит объект ячейки. Этот класс определяется при создании объекта ячейки, и мы не можем изменить это положение, не видоизменив сам метод.

А что, если код будет выглядеть следующим образом?

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet(Cell cell) {
        ...
        cell.Recalculate();
        ...
    }
    ...
}
```

Является ли теперь швом вызов метода `cell.Recalculate` в методе `buildMartSheet`? Да, является. Мы можем создать в тесте объект класса `CustomSpreadsheet` (электронная таблица покупателей) и вызвать метод `buildMartSheet` (составить рыночный список), используя какой угодно объект класса `Cell`. В конечном итоге мы заменили вызов метода `cell.Recalculate`, не меняя вызывающий его метод.

А где же находится разрешающая точка?

В данном примере разрешающей точкой является список аргументов метода `buildMartSheet`. В этой точке мы можем решить, какой именно объект следует передать, а также изменить поведение метода `Recalculate` любым удобным для тестирования способом.

Итак, мы рассмотрели простой пример объектного шва, что характерно для большинства швов данного типа. А теперь обратимся к более сложному примеру. Имеется ли объектный шов при вызове метода `Recalculate` в приведенном ниже варианте метода `buildMartSheet`?

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet(Cell cell) {
        ...
        Recalculate(cell);
        ...
    }

    private static void Recalculate(Cell cell) {
        ...
    }

    ...
}
```

В данном случае метод `Recalculate` будет статическим. Является ли вызов метода `Recalculate` швом? Да, является. Ведь нам не нужно править метод `buildMartSheet`, чтобы изменить поведение в данном вызове. Если же мы удалим ключевое слово `static` в объявлении метода `Recalculate` и сделаем этот метод защищенным (`protected`), а не частным (`private`), то сможем выполнить подклассификацию и переопределить его во время теста следующим образом:

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet(Cell cell) {
        ...
        Recalculate(cell);
        ...
```

```
    }

    protected void Recalculate(Cell cell) {
        ...
    }
    ...
}

public class TestingCustomSpreadsheet extends CustomSpreadsheet {
    protected void Recalculate(Cell cell) {
        ...
    }
}
```

Не имеет ли все это слишком косвенный характер? Если нам так не нравится зависимость, то почему бы нам просто не перейти к самому коду и изменить его? Иногда такой прием срабатывает, но если приходится иметь дело с особенно глубоко вложенным унаследованным кодом, то зачастую при размещении тестов по местам лучше всего стараться видоизменять такой код как можно меньше. Если вам известны швы, предоставляемые выбранным языком программирования, и вы знаете, как с ними обращаться, то можно разместить тесты по местам более безопасным, как правило, способом, чем в том случае, если бы вы действовали иначе.

Рассмотренные выше типы швов относятся к числу основных. Их можно обнаружить во многих языках программирования. Рассмотрим еще раз пример кода, приведенный в начале этой главы, и попробуем выявить в нем какие-нибудь швы.

```
bool CAsyncSslRec::Init()
{
    if (m_bSslInitialized) {
        return true;
    }
    m_smutex.Unlock();
    m_nSslRefCount++;

    m_bSslInitialized = true;

    FreeLibrary(m_hSslDll1);
    m_hSslDll1=0;
    FreeLibrary(m_hSslDll2);
    m_hSslDll2=0;

    if (!m_bFailureSent) {
       m_bFailureSent=TRUE;
       PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
    }

    CreateLibrary(m_hSslDll1,"syncesel1.dll");
    CreateLibrary(m_hSslDll2,"syncesel2.dll");

    m_hSslDll1->Init();
    m_hSslDll2->Init();
    return true;
}
```

Какие же швы имеются в вызове функции `PostReceiveError` Ниже перечислены все эти швы.

1. `PostReceiveError` — это глобальная функция, и поэтому при ее вызове мы можем без труда использовать *компоновочный шов*. С этой целью мы можем создать библиотеку с функцией заглушки и прикомпоновать ее, чтобы изменить поведение. Разрешающей точкой в данном случае будет служить сборочный файл проекта или же определенная настройка в интегрированной среде разработки (IDE). Нам придется изменить сборку, чтобы прикомпоновать тестовую библиотеку для тестирования или же выходную библиотеку для построения реальной системы.
2. Мы можем добавить в код директиву `#include` и использовать препроцессор для определения макрокоманды `PostReceiveError` при тестировании. В этом случае у нас имеется *шов предварительной обработки*. А где его разрешающая точка? Для включения и выключения макроопределения мы можем воспользоваться директивой препроцессора `#define`.
3. Кроме того, мы можем объявить виртуальную функцию, заменяющую функцию `PostReceiveError`, как было показано в начале этой главы. И для этого у нас имеется *объектный шов*. Где же находится его разрешающая точка? В данном случае разрешающая точка окажется в том месте, где мы решим создать объект. В частности, мы можем создать объект `CAsyncSslRec` или же объект некоторого тестового подкласса, переопределяющего функцию `PostReceiveError`.

Просто поразительно, сколько способов у нас имеется для того, чтобы заменить поведение в данном вызове функции, не прибегая к правке метода.

```
bool CAsyncSslRec::Init()
{
    ...
    if (!m_bFailureSent) {
        m_bFailureSent=TRUE;
        PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
    }
    ...

    return true;
}
```

Очень важно выбрать правильный тип шва, когда требуется подвергнуть тестированию отдельные фрагменты кода. Как правило, *объектные швы* лучше всего выбирать в языках объектно-ориентированного программирования. А *швы предварительной обработки* и *компоновочные швы* полезно выбирать периодически, хотя они не столь очевидны, как объектные швы. Кроме того, зависящие от них тесты трудно сопровождать. Поэтому я лично предпочитаю оставлять швы предварительной обработки и компоновочные швы на тот случай, когда зависимости слишком распространены в коде и лучшей альтернативы справиться с ними не существует.

Когда вы привыкните рассматривать код с точки зрения швов, будет легче выбрать способ тестирования существующего кода и структуру нового кода, упрощающего само тестирование.

Глава 5

Инструментальные средства

Какие же инструментальные средства требуются для работы с унаследованным кодом? Для этого необходим редактор или интегрированная среда разработки (IDE), система сборки кода, а также среда тестирования. Если же для избранного языка программирования имеются инструментальные средства реорганизации кода, то и они могут оказаться весьма полезными.

В этой главе рассматриваются некоторые доступные в настоящее время инструментальные средства, а также роль, которую они играют в работе с унаследованным код.

Инструментальные средства автоматической реорганизации кода

Конечно, код может быть реорганизован и вручную, но если имеется инструментальное средство, автоматизирующее этот процесс, то с его помощью можно сэкономить немало времени. В 1990-е годы Билл Опдайк начал разработку инструментального средства реорганизации кода C++ как часть своей диссертации на тему реорганизации кода. И хотя это средство так и не стало коммерчески доступным, мне хорошо известно, что оно вдохновило многих на аналогичные разработки для других языков программирования. Одним из примечательных средств данного типа стал браузер реорганизации кода Smalltalk, разработанный Джоном Брантом и Доном Робертсом в университете штата Иллинойс. Этот браузер поддерживал обширный ряд видов реорганизации кода и еще долго служил образцом современного подхода к автоматической реорганизации кода. С тех пор были предприняты многочисленные попытки внедрить поддержку реорганизации кода в различные широко распространенные языки программирования. На момент написания этой книги имелось немало инструментальных средств реорганизации кода Java, причем большинство из них встроены в интегрированные среды разработки. Имеются также инструментальные средства реорганизации кода для Delphi и ряд относительно новых средств для C++. Аналогичные средства для C# находились на стадии разработки, когда писались эти строки.

На первый взгляд, все эти средства упрощают реорганизацию кода, но это справедливо лишь для некоторых сред. К сожалению, реорганизация кода в этих средствах поддерживается по-разному. Прежде всего, напомним определение самой реорганизации кода из книги Мартина Фаулера, *Рефакторинг. Улучшение существующего кода*, "Символ-Плюс", 2008.

Реорганизация кода, или рефакторинг, — изменение, вносимое во внутреннюю структуру программного обеспечения, чтобы упростить понимание и удешевить модификацию его кода, не изменяя его существующее поведение.

Изменение может считаться реорганизацией кода лишь в том случае, если оно не изменяет поведение. Инструментальные средства реорганизации кода должны проверять тот факт, что изменение кода не приводит к изменению поведения, и многие из них дейс-

твительно делают это. Данное правило было главным для браузера реорганизации кода Smalltalk, созданного Биллом Опдайком, а также для многих ранних вариантов инструментальных средств реорганизации кода Java. В крайних случаях некоторые средства не выполняют подобную проверку, и тогда они способны внести незначительные программные ошибки в код при его реорганизации.

Выбирать инструментальные средства реорганизации кода следует очень внимательно. Сначала выясните мнение разработчиков о безопасности их средств, а затем выполните свои собственные тесты. Когда мне приходится иметь дело с новым инструментальным средством реорганизации кода, я нередко подвергаю его незначительным санитарным проверкам. Так, если попытаться извлечь метод и присвоить ему имя метода, который уже существует в данном классе, то укажет ли такое средство на подобное действие как на ошибочное? А если это имя метода в базовом классе, то обнаружит ли такое средство данный факт? Если оно не в состоянии сделать ни то, ни другое, то метод может быть ошибочно переопределен, а код — нарушен.

В этой книге речь идет о работе с унаследованным кодом как с поддержкой автоматической реорганизации кода, так и без таковой. В примерах, приводимых на страницах данной книги, специально упоминается о том, что наличие инструментального средства реорганизации кода предполагается.

Но в любом случае предполагается, что виды организации кода, поддерживаемые инструментальным средством, сохраняют поведение. Если вы обнаружите обратное, то не пользуйтесь автоматической реорганизацией кода в избранном вами инструментальном средстве, и следуйте рекомендациям, которые даются на тот случай, если у вас нет инструментального средства реорганизации кода — так будет надежнее.

Тесты и автоматическая реорганизация кода

Если у вас имеется инструментальное средство автоматической реорганизации кода, то не поддавайтесь искушающей мысли о том, что вам не нужно писать тесты для кода, который вы собираетесь реорганизовать. Тесты не требуется писать лишь в некоторых случаях. Если избранное вами инструментальное средство реорганизует код безопасным способом и вы переходите от одного вида автоматической реорганизации кода к другой, не внося больше никаких других правок, то может сложиться представление, будто сделанные правки не изменили поведение. Но так бывает далеко не всегда. И вот тому пример.

```
public class A {
    private int alpha = 0;
    private int getValue() {
        alpha++;
        return 12;
    }
    public void doSomething() {
        int v = getValue();
        int total = 0;
        for (int n = 0; n < 10; n++) {
            total += v;
        }
    }
}
```

Как минимум в двух инструментальных средствах реорганизации кода Java мы можем реорганизовать приведенный выше код, удалив переменную `v` из метода `doSomething` (сделать что-нибудь). После реорганизации этот код примет следующий вид:

```
public class A {
    private int alpha = 0;
    private int getValue() {
        alpha++;
        return 12;
    }
    public void doSomething() {
        int total = 0;
        for (int n = 0; n < 10; n++) {
            total += getValue();
        }
    }
}
```

Заметили ли вы ошибку? Упомянутая выше переменная была удалена, но теперь значение переменной `alpha` увеличивается с приращением 10, а не 1. Очевидно, что при таком изменении кода поведение не сохраняется.

Прежде чем воспользоваться инструментальным средством автоматической реорганизации кода, целесообразно окружить реорганизуемый код соответствующими тестами. Конечно, автоматическую реорганизацию кода можно отчасти выполнять из без тестов, но тогда вы должны точно знать, что именно проверяет используемое вами инструментальное средство и что оно не в состоянии проверить. Когда я начинаю пользоваться новым инструментальным средством, то, прежде всего, проверяю на деле его истинные возможности поддерживать извлечение методов. Если я доверяю ему в достаточной мере, чтобы пользоваться им без тестов, значит, я могу перевести код в более подходящее для тестирования состояние.

Имитирующие объекты

Как упоминалось ранее, одно из самых больших затруднений в работе с унаследованным кодом связано с зависимостью. Ведь нам нужно выполнить фрагмент кода отдельно, чтобы выяснить, что он делает. И для этого нам зачастую приходится разрывать зависимости от другого кода. Но сделать это не так-то просто. Если мы удалим другой код, то нам придется заменить его какими-то элементами, предоставляющими правильные значения при тестировании, чтобы тщательно проверить интересующий нас фрагмент кода. В объектно-ориентированном коде такими элементами являются так называемые имитирующие объекты.

Имеется целый ряд свободно доступных библиотек имитирующих объектов. Ссылки на большинство из них можно, в частности, найти на веб-сайте `www.mockobjects.com`.

Средства блочного тестирования

Инструментальные средства тестирования имеют долгую и разную историю. Не проходит и года, чтобы я не консультировал четыре или пять групп разработчиков, пользующихся специально приобретенными дорогими инструментальными средствами с лицензиями на каждое рабочее место, которые в конечном итоге не оправдывают потраченные на них деньги. Справедливости ради следует отметить, что выбрать подходящие средства для тестирования не так-то просто. Ведь многих соблазняет одна только мысль, что они смогут тестировать код с помощью пользовательского (графического или Интернета) интерфейса, не делая ничего особенного с проверяемым приложением. Такое, конечно, возможно, но, как правило, группу разработчиков ожидает намного более серьезная работа, чем они готовы предположить. Кроме того, пользовательский интерфейс — зачастую не самое лучшее место для написания тестов. Пользовательские интерфейсы весьма непостоянны и слишком далеки от тестируемых функциональных возможностей. Если тест, выполняемый с помощью пользовательского интерфейса, не проходит, то выяснить, почему это произошло, будет очень непросто. Несмотря на это разработчики тратят немалые деньги на то, чтобы попытаться организовать тестирование полностью подобными инструментальными средствами.

Самые эффективные инструментальные средства тестирования, с которыми мне пришлось иметь дело, были свободно доступны. Прежде всего, это среда тестирования xUnit. Это средство было первоначально разработано Кентом Беком, а затем перенесено на платформу Java тем же Кентом Беком и Эрихом Гаммой. Оно имеет небольшую, но эффективную конструкцию для среды блочного тестирования. Ниже перечислены основные возможности этого инструментального средства.

- Позволяет программистам писать тесты на том языке, на котором они ведут разработку.
- Все тесты выполняются обособленно.
- Тесты могут быть сгруппированы в наборы, чтобы неоднократно выполнять их по требованию.

Среда xUnit перенесена на большинство основных языков программирования и ряд менее распространенных, экзотических языков.

Самым революционным и замечательным свойством xUnit является простота и конкретность конструкции этой среды тестирования. Она дает нам возможность писать тесты с минимальными хлопотами. Несмотря на то, что среда xUnit была первоначально разработана для блочного тестирования, ею можно пользоваться и для написания крупных тестов, поскольку для xUnit не имеет никакого значения, насколько велик или мал тест. Если тест написан на том языке, на котором ведется разработка, то его можно выполнить и в xUnit.

Многие примеры, приведенные в этой книге, составлены на языках Java и C++. Для программирования на Java лучше подходит средство тестирования JUnit, которое очень похоже на большинство других средств тестирования типа xUnit. А для программирования на C++ я лично пользуюсь средством тестирования CppUnitLite, которое я написал сам. Оно заметно отличается от упомянутых средств, и поэтому рассматривается в этой главе отдельно. Справедливости ради следует отметить, что за основу CppUnitLite было взято другое средство, называвшееся CppUnit, написанное мной первоначально. Но после выпуска CppUnit я обнаружил, что это средство тестирования может стать более компактным, простым в использовании и намного более переносимым, если использовать в нем идиомы C и только открытое подмножество языка C++.

Средство тестирования JUnit

В JUnit для написания тестов требуется подклассификация класса `TestCase` (контрольный пример).

```
import junit.framework.*;

public class FormulaTest extends TestCase {
    public void testEmpty() {
        assertEquals(0, new Formula("").value());
    }

    public void testDigit() {
        assertEquals(1, new Formula("1").value());
    }
}
```

Каждый метод в тестовом классе определяет тест, если у него имеется сигнатура в следующей форме: `testXXX()`, где `XXX` — имя, присваиваемое тесту. Кроме того, каждый тестовый метод может содержать код и утверждения. Так, в приведенном выше методе `testEmpty` (тест на пустое значение) имеется код для создания нового объекта `Formula` и вызова метода `value` (значение). В нем имеется также код утверждения, которое проверяет, равно ли значение 0. Если это так, то тест проходит, а иначе он не проходит.

При выполнении тестов в среде JUnit происходит, по существу, следующее. Модуль выполнения тестов в среде JUnit загружает тестовый класс, подобный приведенному выше, а затем он использует рефлексию для поиска всех тестовых методов данного класса. Следующее действие данного модуля из разряда более изощренных. Он создает совершенно отдельный объект для каждого из обнаруженных тестовых методов. Как следует из приведенного выше фрагмента кода, в тестовом классе создаются два объекта: один — для выполнения метода `testEmpty`, а другой — для выполнения метода `testDigit` (тест на цифру). Если же вас интересуют классы этих объектов, то в обоих случаях используется один и тот же класс `FormulaTest` (тест по формуле). Каждый объект точно настраивается на выполнение одного из тестовых методов в классе `FormulaTest`. Они никак не могут взаимно влиять. Ниже приведен соответствующий пример.

```
public class EmployeeTest extends TestCase {
    private Employee employee;

    protected void setUp() {
        employee = new Employee("Fred", 0, 10);
        TDate cardDate = new TDate(10, 10, 2000);
        employee.addTimeCard(new TimeCard(cardDate,40));
    }

    public void testOvertime() {
        TDate newCardDate = new TDate(11, 10, 2000);
        employee.addTimeCard(new TimeCard(newCardDate, 50));
        assertTrue(employee.hasOvertimeFor(newCardDate));
    }

    public void testNormalPay() {
```

```
        assertEquals(400, employee.getPay());
    }
}
```

В классе `EmployeeTest` (тест по служащим) имеется специальный метод `setUp` (настройка). Этот метод определяется в классе `TestCase` и выполняется в каждом тестовом объекте перед выполнением тестового метода. Метод `setUp` позволяет создать ряд объектов, используемых в тесте. Эти объекты создаются одним и тем же способом перед выполнением каждого теста. В том объекте, где выполняется метод `testNormalPay` (тест на обычную оплату), объект служащего (`employee`), созданный в методе `setUp`, проверяется на правильность начисления зарплаты по карточке табельного учета, введенной в виде отдельного объекта в методе `setUp`. А в том объекте, где выполняется метод `testOvertime` (тест на сверхурочные), объект служащего, созданный для данного объекта в методе `setUp`, получает дополнительную карточку табельного учета, после чего проверяется запуск условия оплаты сверхурочных по второй карточке табельного учета. Метод `setUp` вызывается для каждого объекта класса `EmployeeTest`, и каждый из этих объектов получает свой ряд дополнительных объектов, создаваемых в методе `setUp`. Если же по завершении теста требуется выполнить что-нибудь особенное, то для этой цели можно переопределить другой метод, называемый `tearDown` (сброс) и определяемый в классе `TestCase`. Этот метод выполняется после тестового метода каждого объекта.

При первом знакомстве со средством тестирования xUnit оно выглядит несколько странно. Зачем вообще в классах контрольных примеров (`TestCase`) нужны методы `setUp` и `tearDown`? Почему нельзя просто создать объекты, которые требуются в конструкторе класса? Конечно, это можно сделать, но не следует забывать, как именно модуль выполнения тестов обращается с классами контрольных примеров. Он переходит к каждому классу контрольного примера и создает ряд объектов — по одному для каждого тестового метода. Это довольно большой ряд объектов, но ведь нет ничего плохого в том, что они еще не распределили то, что им требуется. Поместив код в метод `setUp`, чтобы создать то, что нам требуется, именно тогда, когда нам это будет нужно, мы экономим немало ресурсов. Кроме того, задерживая выполнение метода `setUp`, мы можем выполнить его и в тот момент, когда сумеем выявить любые осложнения, которые возникают во время настройки теста, и своевременно сообщить о них.

Средство тестирования CppUnitLite

Разрабатывая первоначальную переносимую версию средства тестирования CppUnitLite, я старался сделать его как можно более похожим на JUnit. Я решил, что это будет удобнее тем пользователям, которые уже знакомы с архитектурой JUnit, и поэтому такой подход к разработке данного средства показался мне наиболее подходящим. Но почти сразу я убедился в том, что ряд свойств JUnit будет очень трудно, а то и просто невозможно, реализовать полностью на C++ из-за отличий в возможностях C++ и Java. Главная трудность заключалась в отсутствии свойства рефлексии в C++. Если в Java можно опираться на ссылки, которые делаются на методы порожденного класса, находить методы во время выполнения и т.д., то в C++ приходится писать код для регистрации метода, к которому требуется получить доступ во время выполнения. Это в итоге вынуждает писать собственную функцию в тестовом классе, чтобы модуль выполнения тестов мог выполнять объекты для отдельных методов.

```
Test *EmployeeTest::suite()
{
    TestSuite *suite = new TestSuite;
    suite.addTest(new TestCaller<EmployeeTest>("testNormalPay",
        testNormalPay));
    suite.addTest(new TestCaller<EmployeeTest>("testOvertime",
        testOvertime));
    return suite;
}
```

Нечего и говорить, насколько это неприятное занятие. Очень трудно сохранить темп написания тестов, когда приходится объявлять тестовые методы в заголовке класса, определять их в исходном файле и регистрировать в методе тестового набора (`suite`). Для разрешения всех этих затруднений имеются самые разные алгоритмы с применением макрокоманд, но в конечном итоге я остановил свой выбор на алгоритме, позволяющем писать тест с помощью следующего исходного файла.

```
#include "testharness.h"
#include "employee.h"
#include <memory>

using namespace std;

TEST(testNormalPay,Employee)
{
    auto_ptr<Employee> employee(new Employee("Fred", 0, 10));
    LONGS_EQUALS(400, employee->getPay());
}
```

В этом тесте используется макрокоманда `LONGS_EQUALS`, сравнивающая два длинных целых значения. Она ведет себя так же, как и метод `assertEquals` (утвердить равенство) в JUnit, только она приспособлена для значений типа `long`.

Макрокоманда `TEST` подспудно выполняет целый ряд действий. Она создает подкласс тестового класса и присваивает ему имя, соединяя вместе два аргумента (имя теста — с именем тестируемого класса). Затем она создает экземпляр этого класса, который настраивается на выполнение кода, указанного в фигурных скобках. Этот экземпляр статический. Когда программа загружается, она добавляется к статическому списку тестовых объектов. А в дальнейшем модуль выполнения тестов просматривает этот список и выполняет каждый тест.

Создав эту небольшую среду тестирования, я решил не выпускать ее, поскольку код макрокоманд не был совершенно ясен, и мне пришлось долго убеждать других писать более ясный код. С аналогичными трудностями столкнулся мой друг Майк Хилл еще до того, как мы совместно создали среду тестирования TestKit, ориентированную на приложения компании Microsoft и выполнявшую регистрацию аналогичным образом. Ободренный Майком, я начал сокращать количество последних свойств C++, использованных в моей небольшой среде тестирования, и только после этого выпустил ее. (Упомянутые выше затруднения были главным недостатком среды CppUnit. Буквально каждый день я получал сообщения по электронной почте от пользователей моей среды тестирования, которые не могли применять шаблоны, исключения или стандартные библиотеки в их компиляторе C++.)

Обе среды CppUnit и CppUnitLite вполне подходят в качестве средств тестирования. Тесты, написанные с помощью среды CppUnitLite, получаются немного короче, и поэтому именно она используется в примерах на языке C++, приведенных в этой книге.

Средство тестирования NUnit

Средство NUnit представляет собой среду тестирования для языков платформы .NET. В частности, она позволяет писать тесты для кода C#, VB.NET или любого другого языка, применяемого на платформе .NET. По своему принципу действия среда NUnit очень похожа на JUnit. Ее главное отличие заключается в том, что в ней используются атрибуты для пометки тестовых методов и классов. Синтаксис этих атрибутов зависит от конкретного языка платформы .NET, на котором пишутся тесты.

Ниже приведен пример теста, написанного в среде NUnit на языке VB.NET.

```
Imports NUnit.Framework

<TestFixture()> Public Class LogOnTest
    Inherits Assertion

    <Test()> Public Sub TestRunValid()
        Dim display As New MockDisplay()
        Dim reader As New MockATMReader()
        Dim logon As New LogOn(display, reader)
        logon.Run()
        AssertEquals("Please Enter Card", display.LastDisplayedText)
         AssertEquals("MainMenu",logon.GetNextTransaction().GetType.Name)
    End Sub
End Class
```

В этом тесте `<TestFixture()>` и `<Test()>` являются атрибутами, помечающими `LogonTest` (тест на регистрацию в системе) в качестве тестового класса, а `TestRunValid` (прохождение теста) — в качестве тестового метода.

Другие средства блочного тестирования типа xUnit

Существует множество переносимых версий xUnit для самых разных языков и платформ. Как правило, они поддерживают определение, группирование и выполнение блочных тестов. Для того чтобы найти конкретную переносимую версию xUnit для используемых вами языка и платформы, обратитесь по адресу `www.xprogramming.com` и перейдите к разделу `Software Downloads` (Загружаемые программные средства). Этот сайт, организованный Роном Джеффрисом, фактически представляет собой хранилище всех переносимых версий xUnit.

Средства общего тестирования

Среды тестирования xUnit, описанные в предыдущем разделе, предназначены для блочного тестирования. Их можно использовать и для поочередного тестирования нескольких классов, но эту работу лучше выполнять в средах FIT и Fitnesse.

Среда интегрированного тестирования FIT

Среда интегрированного тестирования (Framework for Integrated Tests — FIT) разработана очень компактно и изящно Уардом Каннингхэмом. В ее основу положен принцип простоты и эффективности. Если составить документы на систему и вставить в них таблицы, описывающие входные и выходные данные системы, а затем сохранить эти документы в формате HTML, то среда FIT может выполнить их в виде тестов.

Среда FIT воспринимает HTML-документ, выполняет тесты, определенные в HTML-таблицах этого документа, и выдает результат в виде выходного HTML-документа. Этот выходной документ выглядит так же, как и входной документ со всеми сохраненными в нем таблицами и текстом. Но в клетках таблиц зеленым цветом выделены значения, при которых тест проходит, а красным цветом — значения, при которых тест не проходит. Имеется также возможность для вывода итоговой информации в результирующий HTML-документ.

Для того чтобы все это работало, достаточно выполнить специальную настройку кода обработки таблиц. Благодаря этому данная среда тестирования будет знать, как ей выполнять отдельные фрагменты кода и извлекать из них результаты. Как правило, такая настройка не составляет особого труда, поскольку среда FIT предоставляет код для поддержки самых разных типов таблиц.

Одним из самых замечательных свойств среды FIT является возможность стимулировать общение между теми, кто разрабатывает программное обеспечение, и теми, кто определяет его функции и назначение. Последние могут составлять документы и встраивать в них конкретные тесты. Такие тесты выполняются, но обычно не проходят. А после того как разработчики добавят конкретные свойства в программное обеспечение, тесты будут проходить. Таким образом, у пользователей и разработчиков формируется общее и самое последнее представление о возможностях разрабатываемой системы.

У среды FIT имеет немало других полезных свойств. Подробнее о ней можно узнать по адресу `http://fit.c2.com`.

Среда тестирования Fitnesse

Среда тестирования Fitnesse, по сути, представляет собой среду FIT, размещенную в общедоступной вики-сети. Большая ее часть разработана Робертом и Майкой Мартинами. Мне лично мало приходилось работать в среде Fitnesse, поскольку я вынужден был сосредоточить основное внимание на этой книге. Но я надеюсь вскоре вернуться к этой среде и поработать в ней больше.

В среде Fitnesse поддерживаются иерархические веб-страницы, определяющие тесты типа FIT. Страницы тестовых таблиц могут выполняться как отдельно, так и наборами, а различные дополнительные возможности позволяют легко организовать совместную работу над разрабатываемым проектом. Среда Fitnesse доступна по адресу `http://www.fitnesse.org`. Как и все остальные инструментальные средства тестирования, описанные в этой главе, она свободно доступна и поддерживается сообществом разработчиков программного обеспечения.

Часть II

Изменение программного обеспечения

Глава 6

Изменения необходимы, а времени на это нет

Откровенно говоря, в книге, которую вы читаете, описывается дополнительная работа. Такая работа, делается, как правило, не сразу, но для ее завершения может потребоваться немало времени, чтобы внести предполагаемые изменения в код.

Дело в том, что работа, которую приходится проделывать для разрыва зависимостей и написания тестов, чтобы внести изменения в код, требует времени, но, с другой стороны, хотелось бы сэкономить драгоценное время и заодно избежать немалых хлопот. Когда же выполнять такую работу? Это зависит от конкретного проекта. Иногда тесты для кода, который требуется изменить, можно написать за два часа, а на само изменение кода потратить около 15 минут. Тут невольно возникает вопрос: стоит ли тратить на такую работу целых два часа? Это зависит от обстоятельств. Ведь если в вашем распоряжении нет написанных тестов, то нельзя заранее сказать, сколько времени у вас уйдет на подобную работу. Вы не можете также заранее знать, сколько времени потребуется на отладку кода, если вы совершили ошибку, т.е. времени, которое вы могли бы сэкономить, если бы у вас были подходящие тесты. Речь идет не только о том времени, которое можно сэкономить, если тесты выловят ошибку, но и о том времени, которое тесты позволяют сэкономить при попытке обнаружить ошибку. Если код окружен тестами, то зачастую его функциональные недостатки легче выявить.

А теперь оценим затраты. Допустим, что нам нужно внести в код простое изменение и с этой целью нам требуется написать тесты, чтобы как-то подстраховаться. В конечном итоге мы вносим все необходимые изменения правильно. Стоило ли тогда писать тесты? Но ведь мы не знаем, когда еще вернемся к данному фрагменту кода, чтобы внести в него очередное изменение. В лучшем случае мы вернемся к нему на следующем шаге итерации, и тогда наши затраты на тесты окупятся быстро. А в худшем случае пройдут годы, прежде чем кто-нибудь вернется к данному коду и видоизменит его. Но, скорее всего, нам придется периодически читать его только для того, чтобы выяснить, требуются ли изменения в этом или другом месте кода. Не проще ли было бы сразу понять, что классы должны быть небольшими, а тесты — блочными? Возможно, и проще. Но ведь это лишь худший случай, и как часто он бывает? Как правило, изменения накапливаются в системах. Если изменения вносятся сегодня, то, скорее всего, их придется внести еще раз, причем очень скоро.

Когда я приступаю к работе с группами разработчиков, то предлагаю им принять участь в следующем эксперименте: попытаться не вносить изменения в код без тестов, покрывающих эти изменения. Если кто-нибудь из разработчиков считает, что не в состоянии писать тесты, им придется немедленно собрать группу на совещание и спросить ее, можно ли вообще писать тесты. Первоначальные результаты такого эксперимента ужасны. Разработчики начинают чувствовать, что не успевают сделать вовремя всю необходимую работу. Но постепенно они начинают обнаруживать, что невольно пересматривают код, чтобы сделать его лучше. Их изменения постепенно упрощаются, и они инстинктивно по-

нимают, что это самый лучший способ продвигаться в работе вперед. Для преодоления этого препятствия группе разработчиков обычно требуется время, но если и есть полезный совет, который можно дать сразу всем группам разработчиков в мире, то он заключается в том, чтобы все члены группы вместе испытали приятное чувство облегчения, написанное на их лицах: “Ребята, мы уже больше не возвращаемся к этому”.

Если вам еще не приходилось испытывать ничего подобно, то вы должны это непременно испытать.

В конечном счете тестирование ускоряет работу, что очень важно практически в любой организации, занимающейся разработкой программного обеспечения. Но откровенно говоря, я, программист, могу только порадоваться тому, что тестирование приносит намного меньше неудовлетворенности от работы.

Как только вы преодолеете подобное препятствие, жизнь не покажется вам прекраснее, но станет явно лучше. Зная цену тестированию и почувствовав разницу, вам остается лишь действовать хладнокровно и рассудительно, принимая решение, что делать в каждом конкретном случае.

Типичный случай из практики разработки программного обеспечения

Приходит начальник и говорит:

— Заказчики недовольны этим свойством. Можем ли мы исправить его сегодня?

— Я не знаю.

Вы оглядываетесь вокруг в поисках тестов. Их нет на месте.

И тогда вы спрашиваете:

— Какой вариант вас устроит?

Вы хорошо знаете, что можете внести изменения во всех 10 местах исходного текста, где эти изменения требуются, и сделать это к концу рабочего дня. Ясно, что это экстренный случай, и на следующий день все можно поправить.

Но не забывайте, что код — это дом, в котором вам придется жить.

Самое трудное при принятии решения о том, следует ли писать тесты при напряженном графике работы, заключается в следующем: вы точно не знаете, сколько времени потребуется на ввод конкретного свойства. Это особенно трудно оценить здраво в унаследованном коде. Впрочем, здесь могут прийти на помощь некоторые способы, подробнее рассматриваемые в главе 16. Если вы не знаете точно, сколько времени вам потребуется на ввод конкретного свойства, и подозреваете, что на это уйдет больше времени, чем у вас есть, то вы невольно поддаетесь искушению ввести это свойство наспех и как можно быстрее. Если затем у вас будет достаточно времени, то можно вернуться к данному свойству, протестировать и реорганизовать его код. Самое трудное во всем этом — вернуться к коду, чтобы протестировать и реорганизовать его. Прежде чем разработчики преодолеют упомянутое выше препятствие, они зачастую избегают подобной работы. Это может быть проблемой морального состояния. Подробнее о некоторых конструктивных способах продвижения в работе вперед речь пойдет в главе 24.

Все сказанное выше может показаться похожим на настоящую дилемму: заплатить теперь или же заплатить позже, но больше, т.е. писать тесты по ходу внесения изменений или же примириться с тем фактом, что со временем сделать это будет труднее. В одних случаях это действительно сделать труднее, а в других — легче.

Если вам приходится вносить изменения в класс безотлагательно, попробуйте получить экземпляр данного класса в средствах тестирования. Если же у вас нет такой возможности, то обратитесь к материалу главы 9 или 10. Внести изменяемый код в средства тестирования, возможно, окажется легче, чем вы думаете. Если после ознакомления с материалом этих глав вы придете к выводу, что не можете позволить себе теперь разорвать зависимости и разместить тесты по местам, то проанализируйте тщательно изменения, которые вам необходимо сделать. Можете ли вы сделать их, написав новый код? Как правило, это можно сделать. В остальной части этой главы рассматривается ряд способов, позволяющих добиться этого.

Ознакомьтесь с этими методами и проанализируйте их, но помните, что ими следует пользоваться очень аккуратно. Ведь когда вы применяете их, то вводите в свою систему тестируемый код, но если вы не покроете вызывающий его код, то не сможете проверить его функции. Поэтому пользуйтесь этими методами аккуратно.

Почкование метода

Если в систему требуется ввести новое свойство, которое можно выразить в виде совершенно нового кода, то напишите этот код в новом методе, а затем вызовите его из тех мест, где требуются новые функциональные возможности. Вполне вероятно, что добраться до точек этих тестируемых вызовов будет нелегко, но, по крайней мере, вы можете написать тесты для нового кода. Ниже приведен соответствующий пример.

```
public class TransactionGate
{
    public void postEntries(List entries) {
        for (Iterator it = entries.iterator(); it.hasNext(); ) {
        Entry entry = (Entry)it.next();
        entry.postDate();
        }
        transactionBundle.getListManager().add(entries);
    }
    ...
}
```

Нам нужно ввести код, чтобы проверить, что ни один из новых элементов (`entries`) не окажется в пакете транзакции (`transactionBundle`) до отправки их дат и последующего их ввода. Глядя на этот код, можно подумать, что он должен быть расположен в начале метода, т.е. до цикла. Но на самом деле его можно расположить и внутри цикла. Для этого код можно изменить следующим образом:

```
public class TransactionGate
{
    public void postEntries(List entries) {
        List entriesToAdd = new LinkedList();
        for (Iterator it = entries.iterator(); it.hasNext(); ) {
            Entry entry = (Entry)it.next();
            if (!transactionBundle.getListManager().hasEntry(entry) {
                entry.postDate();
                entriesToAdd.add(entry);
            }
```

```
        }
        transactionBundle.getListManager().add(entriesToAdd);
    }
    ...
}
```

Такое изменение кажется довольно простым, но на самом деле оно достаточно радикальное. Как же нам узнать, насколько правильно мы поступили? Между вновь введенным кодом и старым кодом нет никакого разделения. Более того, мы сделали код чуть менее ясным, объединив в нем две операции — отправку даты и обнаружение дублированного элемента. Новый метод довольно мал, но он уже содержит менее ясный код, а кроме того, мы ввели в него временную переменную. Временные переменные не всегда являются злом для программирования, но иногда они привлекают наше внимание к новому коду. Если следующее изменение, которое нам нужно сделать, связано с манипулированием всеми недублированными элементами до их ввода, то единственное место в коде для размещения временной переменной находится непосредственно в данном методе. Невольно возникает искушение ввести в метод и сам код. Можно ли сделать это иначе?

Да, можно. Удаление дублированного элемента можно рассматривать как отдельную операцию. Для создания нового метода под названием `uniqueEntries` (уникальные элементы) мы можем воспользоваться способом *разработки посредством тестирования*.

```
public class TransactionGate
{
    ...
    List uniqueEntries(List entries) {
        List result = new ArrayList();
        for (Iterator it = entries.iterator(); it.hasNext(); ) {
            Entry entry = (Entry)it.next();
            if (!transactionBundle.getListManager().hasEntry(entry) {
                result.add(entry);
            }
        }
        return result;
    }
    ...
}
```

Проще было бы написать тесты, которые привели бы нас к такому коду, как приведенный выше для данного метода. Когда у нас есть метод, мы можем всегда вернуться к первоначальному коду и добавить вызов данного метода.

```
public class TransactionGate
{
    ...
    public void postEntries(List entries) {
        List entriesToAdd = uniqueEntries(entries);
        for (Iterator it = entriesToAdd.iterator(); it.hasNext(); ) {
            Entry entry = (Entry)it.next();
            entry.postDate();
        }
        transactionBundle.getListManager().add(entriesToAdd);
```

```
    }
    . . .
}
```

И в этом коде по-прежнему присутствует временная переменная, но сам код выглядит намного менее запутанным. Если же нам требуется ввести дополнительный код, обрабатывающий недублированные элементы, то мы можем создать метод и для этого кода, а затем вызвать его отсюда. А если нам в конечном итоге потребуется еще больше кода для обработки недублированных элементов, то мы можем ввести класс и перенести в него все новые методы. Это в конечном итоге позволит нам сохранить все новые методы компактными, краткими и легко понятными.

Это был пример *почкования метода*. Ниже перечислены по пунктам шаги, которые следует предпринять для почкования метода.

1. Определите место, где требуется внести изменения в код.
2. Если изменение можно выразить в виде отдельной последовательности операторов, располагаемых в одном месте в отдельном методе, то напишите код вызова нового метода, в котором должны быть выполнены все необходимые действия, снабдив его соответствующим комментариями. (Я предпочитаю делать это перед написанием самого метода, чтобы лучше представлять себе, как будет выглядеть вызов метода в определенном контексте.)
3. Определите в исходном методе все локальные переменные, которые могут вам потребоваться, и сделайте их аргументами в вызове нового метода.
4. Выясните, должен ли отпочковавшийся метод возвращать какие-нибудь значения в исходный метод. Если должен, то измените вызов отпочковавшегося метода так, чтобы он возвращал значение, присваиваемое переменной.
5. Разработайте метод для почкования, применяя *разработку посредством тестирования*.
6. Удалите комментарии в исходном методе, чтобы активизировать вызов.

Почкование метода рекомендуется использовать всякий раз, когда добавляемый код представляет собой отдельную часть работы или же когда метод нельзя еще окружить тестами. Этот способ предпочтителен для добавления кода подстановкой.

Иногда зависимости в классе, где предполагается применить почкование метода, настолько запутаны, что создать экземпляр этого класса нельзя, не сымитировав множество аргументов его конструктора. В качестве одной из альтернатив можно воспользоваться *передачей пустого значения*. Когда же и это не помогает, рекомендуется сделать отпочковавшийся метод общедоступным статическим методом. В этом случае, вероятно, придется передать переменные экземпляров исходного класса в виде аргументов, но это все же позволит сделать необходимые изменения в коде. На первый взгляд, идея сделать данный метод статическим кажется довольно странной, но в унаследованном коде она может оказаться плодотворной. Статические методы в классах можно рассматривать в качестве своеобразной постановочной площадки. Нередко, получив несколько статических методов, вы замечаете, что у них общие переменные, и тогда легче увидеть возможность для создания нового класса и переноса в него статических методов в качестве методов экземпляров. Если же они действительно заслуживают того, чтобы стать методами экземпляров в текущем классе, то их можно перенести обратно в этот класс, окончательно переходя к его тестированию.

Преимущества и недостатки

У почкования метода имеются свои преимущества и недостатки. Рассмотрим сначала его недостатки. Прежде всего, используя почкование метода, вы, по существу, оставляете исходный метод и его класс на какое-то время без внимания, не собираясь его тестировать или совершенствовать, а вместо этого вводите ряд новых функций в новый метод. Иногда оставить метод или класс без внимания можно из практических соображений, но все равно это скверная практика, поскольку код остается в неопределенном состоянии. Ведь исходный метод может содержать немало сложного кода и одно отпочкование нового метода. Иногда оказывается совершенно не ясно, почему такая работа делается в каком-то другом месте, тогда как исходный метод остается в неопределенном состоянии. Но, по крайней мере, это указывает на некоторую дополнительную работу, которую можно сделать в дальнейшем, переходя к тестированию исходного класса.

Несмотря на некоторые недостатки, у почкования метода имеются и определенные преимущества. Используя почкование метода, вы отчетливо отделяете новый код от старого. Даже если вам не удастся сразу же перейти к тестированию старого кода, то, по крайней мере, вы сможете наблюдать за изменениями отдельно, благодаря четкой границе раздела между новым и старым кодом. Наблюдение за воздействием на все интересующие вас переменные поможет легче определить, насколько данный код соответствует своему контексту.

Почкование класса

Почкование метода довольно эффективно, но его эффективности оказывается недостаточно в некоторых ситуациях с запутанными зависимостями.

Рассмотрим пример, в котором требуется внести изменения в класс, но в средствах тестирования отсутствует возможность создавать объекты этого класса в течение приемлемого периода времени, а следовательно, и возможность для почкования метода и написания для него тестов в данном классе. Возможно, причиной тому являются зависимости создания, которые сильно затрудняют получение экземпляра класса, или же многочисленные скрытые зависимости. Для того чтобы избавиться от них, придется выполнить немало работы по радикальной реорганизации кода и тем самым отчетливо отделить эти зависимости от класса для его компиляции в средствах тестирования.

В подобных случаях можно создать другой класс для хранения изменений и обращаться к нему из исходного класса. Рассмотрим такую возможность на упрощенном примере.

Ниже приведен код C++ старого метода в классе `QuarterlyReportGenerator` (генератор квартального отчета).

```
std::string QuarterlyReportGenerator::generate()
{
    std::vector<Result> results = database.queryResults(
beginDate, endDate);
    std::string pageText;

    pageText += "<html><head><title>"
        "Quarterly Report"
        "</title></head><body><table>";
    if (results.size() != 0) {
```

```
        for (std::vector<Result>::iterator it = results.begin();
                it != results.end();
                ++it) {
            pageText += "<tr>";
            pageText += "<td>" + it->department + "</td>";
            pageText += "<td>" + it->manager + "</td>";
            char buffer [128];
            sprintf(buffer, "<td>$%d</td>", it->netProfit / 100);
            pageText += std::string(buffer);
            sprintf(buffer, "<td>$%d</td>", it->operatingExpense / 100);
            pageText += std::string(buffer);
            pageText += "</tr>";
        }

    } else {
        pageText += "No results for this period";
    }
    pageText += "</table>";
    pageText += "</body>";
    pageText += "</html>";

    return pageText;
}
```

Допустим, что изменение в коде состоит в том, чтобы ввести строку заголовка для HTML-таблицы, которую формирует этот код. Строка заголовка должна выглядеть следующим образом:

```
"<tr><td>Department</td><td>Manager</td><td>Profit</td><td>Expenses</td></tr>"
```

Допустим далее, что данный класс является достаточно крупным и что на его перенос в средства тестирования потребуется целый рабочий день, а на это времени у нас просто нет.

Изменения в коде мы можем выразить в виде небольшого класса под названием `QuarterlyReportTableHeaderProducer` (генератор заголовка таблицы квартального отчета), получив его способом *разработки посредством тестирования*.

```
using namespace std;

class QuarterlyReportTableHeaderProducer
{
public:
    string makeHeader();
};

string QuarterlyReportTableProducer::makeHeader()
{
    return "<tr><td>Department</td><td>Manager</td>"
        "<td>Profit</td><td>Expenses</td>";
}
```

Получив этот класс, мы можем создать его экземпляр и вызвать его непосредственно в методе `QuarterlyReportGenerator::generate()` следующим образом:

```
...
QuarterlyReportTableHeaderProducer producer;
pageText += producer.makeHeader();
...
```

Уверен, что, глядя на этот код, вы скажете: “Это же несерьезно. Ведь просто смешно создавать класс для такого рода изменения в коде! Этот мелкий класс не дает никаких преимуществ в разработке, а просто вводит совершенно новое понятие, которое только затемняет код”. На данном этапе такое замечание совершенно справедливо. Единственная причина, по которой мы сделали это, состоит в том, чтобы избавиться от плохой зависимости. Но проанализируем наши действия более тщательно.

Что, если назвать класс `QuarterlyReportTableHeaderGenerator` и снабдить его следующим интерфейсом?

```
class QuarterlyReportTableHeaderGenerator
{
public:
    string generate();
};
```

Теперь этот класс является частью понятия, которое нам уже знакомо. `QuarterlyReportTableHeaderGenerator` — генератор, аналогичный `QuarterlyReportGenerator`. У обоих классов имеются методы `generate()`, возвращающие строки. Мы можем задокументировать эту общность обоих классов в коде, создав класс интерфейса и организовав наследование обоих классов от него.

```
class HTMLGenerator
{
public:
    virtual ~HTMLGenerator() = 0;
    virtual string generate() = 0;
};

class QuarterlyReportTableHeaderGenerator : public HTMLGenerator
{
public:
    ...
    virtual string generate();
    ...
};

class QuarterlyReportGenerator : public HTMLGenerator
{
public:
    ...
    virtual string generate();
    ...
};
```

Приложив еще немного усилий, мы можем перейти непосредственно к тестированию класса `QuarterlyReportGenerator` и изменить его реализацию, чтобы бо́льшую часть своих функций он выполнял, пользуясь классами генераторов.

В данном случае мы можем быстро свести класс к ряду понятий, которые уже имеются в приложении. Но во многих других случаях такая возможность отсутствует, хотя это совсем не означает, что мы должны воздержаться от подобных действий. Некоторые отпочковавшиеся классы не сводятся к основным понятиям в приложении, а вместо этого становятся новыми понятиями. Отпочковав класс, можно посчитать такое действие несущественным для проекта до тех пор, пока нечто подобное не будет сделано где-нибудь еще. Иногда дублированный код можно вынести за скобки в новых классах и зачастую их приходится переименовывать, но не стоит надеяться, что все это происходит сразу.

Отпочковавшийся класс нередко воспринимается совсем иначе, когда он только что создан, чем несколько месяцев спустя. Сам факт наличия этого дополнительного нового класса в системе дает немало пищи для размышления. Если требуется внести изменение рядом с этим классом, то возникает вопрос: следует ли отнести такое изменение частично к новому понятию или же немного изменить существующее понятие? Все это будет неотъемлемой частью непрерывного процесса проектирования.

К почкованию класса мы, по существу, прибегаем в двух случаях. В первом случае внесенные нами изменения приводят к добавлению совершенно новой ответственности в один из классов. Например, в программном обеспечении для подготовки налоговых отчислений некоторые виды отчислений могут не проводиться в определенное время года. Проверку по дате можно ввести в класс `TaxCalculator`, но, может быть, лучше возложить основную ответственность за такую проверку не на класс, отвечающий за расчет налогов, а на новый класс? Другой случай мы уже рассматривали в начале этой главы, когда нам требовалось немного расширить функции существующего класса, но мы не могли ввести этот класс в средства тестирования. Если бы это можно было сделать хотя бы в скомпилированном виде, то можно было бы попытаться отпочковать метод, но такая попытка далеко не всегда приносит удачу.

Трудность выявления обоих упомянутых выше случаев состоит в том, что, несмотря на разную мотивацию, их очень непросто различить по конечным результатам. В самом деле, является ли добавление части функций достаточным основанием для того, чтобы отнести его к новой ответственности? Более того, код со временем изменяется, и поэтому решение о почковании класса нередко лучше рассматривать ретроспективно.

Ниже перечислены по пунктам шаги, которые следует предпринять для почкования класса.

1. Определите место, где требуется внести изменения в код.
2. Если изменение можно выразить в виде отдельной последовательности операторов, располагаемых в одном месте в отдельном методе, то придумайте подходящее имя для класса, который мог бы выполнить эту функцию. Затем напишите код для создания объекта этого класса в данном месте и вызовите метод, в котором должны быть выполнены все необходимые действия, после чего закомментируйте строки этого кода.
3. Определите в исходном методе все локальные переменные, которые могут вам потребоваться, и сделайте их аргументами в вызове нового метода.
4. Выясните, должны ли возвращаться какие-нибудь значения из отпочковавшегося класса в исходный метод. Если должны, то введите в отпочковавшийся класс метод, возвращающий эти значения, а затем добавьте вызов в исходный метод для получения этих значений.
5. Разработайте класс для почкования, применяя *разработку посредством тестирования*.
6. Удалите комментарии в исходном методе, чтобы активизировать соответствующие вызовы и создание объектов.

Преимущества и недостатки

Главное преимущество почкования класса заключается в том, что оно позволяет вам продвигаться в своей работе дальше с большей уверенностью, чем в том случае, если бы вам пришлось вносить радикальные изменения. Дополнительное преимущество почкования класса в C++ состоит в том, что вам не нужно видоизменять ни один из существующих заголовочных файлов, чтобы внести изменения в код на месте. Заголовок для нового класса вы можете включить в файл реализации для исходного класса. Еще одним положительным моментом является ввод нового заголовочного файла в проект. Со временем объявления, введенные в новый заголовочный файл, окажутся в конечном итоге в заголовке исходного класса. Благодаря этому сокращается компиляционная нагрузка на исходный класс. По крайней мере, вы будете уверены в том, что не усугубите еще больше и без того непростую ситуацию. А впоследствии вы сможете вернуться к исходному классу и протестировать его.

Главный недостаток почкования класса заключается в его концептуальной сложности. Когда программисты знакомятся с новыми базами кода, у них создается определенное представление о совместном функционировании ключевых классов. Применяя почкование класса, вы должны понять суть абстракций и немало поработать с другими классами. В одних случаях это единственно верный путь, а в других — единственно возможный путь, когда иного выхода просто не существует. Все, что в идеальном случае должно было бы остаться в одном отпочковавшемся классе, в конечном итоге распространяется на ряд других отпочковавшихся классов просто ради большей безопасности самих изменений.

Охват метода

Ввести новое поведение в существующие методы несложно, но зачастую такой способ оказывается не совсем верным. Когда метод только создается, он, как правило, выполняет лишь одну полезную функцию, а любой дополнительный код, добавляемый впоследствии, уже вызывает подозрение. Он чаще всего вводится потому, что должен выполняться одновременно с кодом, к которому он добавляется. В программировании это называлось раньше *временным связыванием*, нередко приводившим к довольно неприятным последствиям при чрезмерном использовании. Ведь при группировании отдельных функций только ради того, чтобы они выполнялись одновременно, взаимосвязь между ними не всегда оказывается достаточно сильной. В дальнейшем иногда оказывается, то одни функции способны выполняться без других, но первоначально они могут развиваться вместе. И разделить их без шва будет очень трудно.

Если требуется ввести иное поведение, то сделать это можно и более простым способом и, в частности, воспользоваться *почкованием метода*. Но иногда более полезным оказывается другой способ, называемый *охватом метода*. Рассмотрим его на следующем простом примере.

```
public class Employee
{
    ...
    public void pay() {
        Money amount = new Money();
        for (Iterator it = timecards.iterator(); it.hasNext(); ) {
            Timecard card = (Timecard)it.next();
```

```
            if (payPeriod.contains(date)) {
                amount.add(card.getHours() * payRate);
            }
        }
        payDispatcher.pay(this, date, amount);
    }
}
```

В приведенном выше методе вводятся карточки ежедневного табельного учета работника, а затем сведения об оплате его труда передаются объекту класса `payDispatcher` (диспетчер оплаты). Всякий раз, когда работнику выплачивается зарплата, приходится обновлять файл, указывая Ф.И.О. этого работника, чтобы затем отправить обновленный файл определенной программе отчетности. Ввести соответствующий код проще всего в метод оплаты (`pay`). Ведь все это должно происходить одновременно, не так ли? А что, если сделать вместо этого следующее.

```
public class Employee
{
    private void dispatchPayment() {
        Money amount = new Money();
        for (Iterator it = timecards.iterator(); it.hasNext(); ) {
            Timecard card = (Timecard)it.next();
            if (payPeriod.contains(date)) {
                amount.add(card.getHours() * payRate);
            }
        }
        payDispatcher.pay(this, date, amount);
    }

    public void pay() {
        logPayment();
        dispatchPayment();
    }

    private void logPayment() {
    ...
    }
}
```

В приведенном выше фрагменте кода метод `pay()` переименован в метод `dispatchPayment()` и сделан частным, а затем создан новый вызывающий его метод `pay()`. В новом методе `pay()` оплата сначала регистрируется, а затем диспетчеризуется. Клиентам, обычно вызывающим метод `pay()`, необязательно знать о подобных изменениях, и они не должны их даже интересовать. Им достаточно сделать свой вызов, и все.

Это одна из форм *охвата метода*. Мы просто создаем метод с именем первоначального метода и делегируем его старому коду. Это полезно в тех случаях, если требуется ввести иное поведение в существующие вызовы первоначального метода. Такой способ оказывается полезным, например, том случае, если при каждом вызове клиентом метода `pay()` нам требуется регистрация.

Имеется и другая форма охвата метода, которая оказывается полезной в тех случаях, когда требуется лишь добавить новый метод, который еще не вызывался ниоткуда. Так, если бы в предыдущем примере потребовалось сделать регистрацию более явной, то нам

бы пришлось ввести в класс `Employee` метод `makeLoggedPayment` (зарегистрировать оплату) следующим образом:

```
public class Employee
{
    public void makeLoggedPayment() {
        logPayment();
        pay();
    }

    public void pay() {
        ...
    }

    private void logPayment() {
        ...
    }

}
```

Теперь у пользователей имеется возможность выбрать тот или иной способ оплаты.

Охват метода — это отличный способ ввести швы при добавлении новых свойств. Но у него имеются два недостатка. Первый недостаток состоит в том, что добавляемое новое свойство не может быть связано логически со старым свойством. Это должно быть нечто, выполняемое до или после старого свойства. Но разве это так уж плохо? Нет, конечно. Если есть возможность сделать это, то ею не грех и воспользоваться. Второй (и более реальный) недостаток заключается в том, что старому коду в отдельном методе приходится присваивать новое имя. В приведенном выше примере коду из метода `pay()` было присвоено имя `dispatchPayment()`. Это несколько усложняет дело, и мне, откровенно говоря, не нравится код, который в итоге получился в данном примере. Метод `dispatchPayment()` на самом деле делает нечто большее, чем диспетчеризацию: он рассчитывает оплату. Если бы у меня были расположенные на своих местах тесты, я извлек бы первую часть метода `dispatchPayment()` в отдельный метод под названием `calculatePay()` и организовал бы его вызов в методе `pay()` следующим образом:

```
public void pay() {
    logPayment();
    Money amount = calculatePay();
    dispatchPayment(amount);
}
```

Благодаря этому все виды ответственности вполне разделены.

Ниже перечислены по пунктам шаги, которые следует предпринять для охвата метода.

1. Определите метод, который требуется изменить.
2. Если изменение можно выразить в виде отдельной последовательности операторов, располагаемых в одном месте, то переименуйте метод, а затем создайте новый метод с такими же именем и сигнатурой, как и у старого метода. Не забудьте затем выполнить *сохранение сигнатур*.
3. Поместите вызов старого метода в новом методе.

4. Разработайте метод для нового свойства, применяя *разработку посредством тестирования*, а затем вызовите его из нового метода.

Во втором случае, когда нас не интересует использование такого же имени, как и у старого метода, необходимо предпринять следующие шаги.

1. Определите метод, который требуется изменить.
2. Если изменение можно выразить в виде отдельной последовательности операторов, располагаемых в одном месте, то создайте для нее новый метод, применяя *разработку посредством тестирования*.
3. Создайте еще один метод, вызывающий новый и старый метод.

Преимущества и недостатки

Охват метода вполне подходит для ввода новых, тестируемых функций в приложение, когда написать тесты для вызывающего кода не так-то просто. При почковании метода или класса код вводится в существующие методы, удлиняя их, по крайней мере, на одну строку, тогда как при охвате метода размеры существующих методов не изменяются.

Еще одно преимущество охвата метода заключается в том, что он делает новые функции явно независимыми от существующих функций. При охвате код, предназначенный для одних целей, никак не связан логически с кодом, служащим иным целям.

Главный недостаток охвата метода состоит в том, то он может привести к появлению неудачных имен. В предыдущем примере мы переименовали метод `pay()` на метод `dispatchPay()` просто потому, что нам потребовалось новое имя для кода в исходном методе. Если этот код не слишком сложен и неподатлив или же если в нашем распоряжении имеется инструментальное средство реорганизации кода, безопасно выполняющее *извлечение метода*, то, осуществив дополнительные извлечения, мы сможем в конечном итоге сделать имена лучше. Но зачастую мы прибегаем к охвату, потому что у нас просто нет других тестов, код неподатлив, а подходящие инструментальные средства отсутствуют.

Охват класса

Охвату метода сопутствует также *охват класса*, действующий почти по тому же принципу. Так, если нам требуется добавить в систему какое-нибудь поведение, то сделать это мы можем не только в существующем методе, но и в другом месте, где этот метод используется. При охвате класса этим местом является другой класс.

Обратимся вновь к примеру кода из класса `Employee`.

```
class Employee
{
    public void pay() {
        Money amount = new Money();
        for (Iterator it = timecards.iterator(); it.hasNext(); ) {
            Timecard card = (Timecard)it.next();
            if (payPeriod.contains(date)) {
                amount.add(card.getHours() * payRate);
            }
        }
```

```
        payDispatcher.pay(this, date, amount);
    }
    ...
}
```

Нам требуется зарегистрировать факт оплаты труда конкретного работника. С этой целью мы можем, в частности, создать еще один класс, содержащий метод `pay()`. Объекты данного класса могут, опираясь на класс `Employee`, выполнить регистрацию оплаты труда работника в методе `pay()`, а затем делегировать его полномочия классу `Employee`, чтобы провести оплату. Зачастую самый простой способ сделать это, если нельзя получить экземпляр первоначального класса в средствах тестирования, состоит в *извлечении средства реализации* или в *извлечении интерфейса* из данного класса и реализации полученного в итоге интерфейса в охватывающем объекте.

В приведенном ниже коде для превращения класса `Employee` в интерфейс использовано извлечение средства реализации, после чего новый класс `LoggingEmployee` (регистрация работника) реализует данный класс. Любой объект класса `Employee` теперь может быть передан классу `LoggingEmployee` для регистрации и оплаты труда работника.

```
class LoggingEmployee extends Employee
{
    public LoggingEmployee(Employee e) {
        employee = e;
    }

    public void pay() {
        logPayment();
        employee.pay();
    }

    private void logPayment() {
    ...
    }
    ...
}
```

Такой способ иначе называется *шаблоном декоратора*. Мы создаем сначала объекты класса, охватывающего другой класс, а затем передаем их. У охватывающего класса должен быть такой же интерфейс, как и у того класса, который он охватывает, чтобы клиенты не знали, что они на самом деле работают с охватывающим классом. В приведенном выше примере класс `LoggingEmployee` является декоратором для класса `Employee`. Он должен непременно содержать метод `pay()`, а также другие методы класса `Employee`, которые используются клиентом.

Шаблон декоратора

Декоратор позволяет сформировать сложное поведение, составляя объекты во время выполнения. Например, в автоматизированной системе управления технологическими процессами производства у нас может быть класс, называемый `ToolController` (контроллер инструментов) с такими методами, как `raise()`, `lower()`, `step()`, `on()` и `off()` для поднятия, опускания, пошагового перемещения, включения и выключения инструмента соответственно. Если нам требуется принять дополнительные меры всякий раз, когда вызывается метод `raise()` или `lower()`, например, предупредить людей, чтобы они

посторонились, то для этой цели мы могли бы ввести ряд дополнительных функций непосредственно в оба метода класса `ToolController`. Но, скорее всего, усовершенствования на этом не закончатся. В конечном итоге нам, возможно, понадобится регистрировать каждое включение или выключение инструмента, а также уведомлять другие, расположенные близко контроллеры при пошаговом перемещении данного инструмента, чтобы они не делали аналогичное перемещение в тот же самый момент времени. Перечень операций, наряду с пятью простыми упомянутыми выше операциями (поднятием, опусканием, пошаговым перемещением, включением и выключением инструмента), может вырасти до бесконечности, не ограничиваясь только созданием подклассов для каждой операции. Ведь число сочетаний этих операций, а следовательно, и разных видов поведения, может быть бесконечным.

Шаблон декоратора идеально подходит для решения подобного рода задач. С его помощью сначала создается абстрактный класс, определяющий ряд операций, требующих поддержки, а затем подкласс, наследующий от этого абстрактного класса, воспринимающий экземпляр класса в своем конструкторе и предоставляющий тело для каждого из методов, поддерживающих упомянутые операции. Ниже приведен пример такого класса для ввода иного поведения в класс `ToolController`.

```
abstract class ToolControllerDecorator extends ToolController
{
protected ToolController controller;
    public ToolControllerDecorator(ToolController controller) {
        this.controller = controller;
    }
    public void raise() { controller.raise(); }
    public void lower() { controller.lower(); }
    public void step() { controller.step(); }
    public void on() { controller.on(); }
    public void off() { controller.off(); }
}
```

На первый взгляд, приведенный выше класс кажется малопригодным, но это не совсем так. Выполнив его подклассификацию и переопределив любые или все его методы, мы можем ввести дополнительное поведение. Так, если требуется уведомить другие контроллеры о пошаговом перемещении инструмента, мы можем получить следующий подкласс `StepNotifyingController`:

```
public class StepNotifyingController extends ToolControllerDecorator
{
    private List notifyees;
    public StepNotifyingController(ToolController controller,
            List notifyees) {
        super(controller);
        this.notifyees = notifyees;
    }
    public void step() {
        // уведомить всех остальных здесь
           ...
        controller.step();
    }
}
```

Самое интересное, что подклассы класса `ToolControllerDecorator` могут быть вложенными.

```
ToolController controller = new StepNotifyingController(
                new AlarmingController
                (new ACMEController()), notifyees);
```

Когда мы выполняем операцию в контроллере, например `step()`, то все остальные контроллеры (`notifyees`) уведомляются об этом, выдается предупреждение и затем выполняется пошаговое перемещение инструмента. Это последнее действие фактически происходит в подклассе `ACMEController`, который уже относится к классу `ToolController`, а не к классу `ToolControllerDecorator`. Он не передает ответственность дальше, а просто выполняет все необходимые операции контроллера. Для использования шаблона декоратора необходимо иметь хотя бы один из "базовых" классов, которые требуется охватить.

Шаблон декоратора очень удобен, но им следует пользоваться весьма экономно. Перемещение по коду, содержащему одни декораторы, отделывающие другие декораторы, можно сравнить со снятием кожуры луковицы слоями. Эта работа необходима, но от нее слезятся глаза.

Рассматриваемый здесь способ отлично подходит для ввода новых функций при наличии в программе множества мест, откуда вызывается такой метод, как `pay()`. Но имеется и другой способ охвата, не носящий столь "декоративный" характер. Рассмотрим ситуацию, в которой нам требуется регистрировать вызовы метода `pay()` только в одном месте. Вместо охвата функций данного метода аналогично декоратору, мы можем поместит этот метод в другой класс, воспринимающий исходные данные о работнике, производящий его оплату и регистрирующий выходные данные о нем.

Ниже приведен небольшой класс, который все это делает.

```
class LoggingPayDispatcher
{
    private Employee e;

    public LoggingPayDispatcher(Employee e) {
        this.e = e;
    }

    public void pay() {
        employee.pay();
        logPayment();
    }

    private void logPayment() {
    ...
    }
    ...
}
```

Теперь мы можем создать класс `LogPayDispatcher` в одном месте, а именно там, где нам требуется регистрировать оплату.

Главная особенность охвата класса состоит в том, что он позволяет вводить новое поведение в систему, по сути, не вводя его в существующий класс. Если требуется охватить

множество вызовов конкретного кода, в таком случае целесообразно охватить их классом декораторного типа. Шаблон декоратора позволяет вводить прозрачным способом новое поведение сразу в целый ряд существующих вызовов такого метода, как `pay()`. С другой стороны, если новое поведение должно проявиться лишь в двух местах программы, то его целесообразно охватить классом недекораторного типа. А со временем следует уделить особое внимание видам ответственности охватывающего класса и проверить, сможет ли этот класс стать еще одним понятием высокого уровня в системе.

Ниже перечислены по пунктам шаги, которые следует предпринять для охвата класса.

1. Определите место, где требуется внести изменения в код.
2. Если изменение можно выразить в виде отдельной последовательности операторов, то создайте класс, воспринимающий тот класс, который предполагается охватить, в виде аргумента конструктора. Если же вы испытываете трудности при создании в средствах тестирования класса, охватывающего исходный класс, то можете воспользоваться *извлечением средства реализации* или *извлечением интерфейса* из охватываемого класса, чтобы получить экземпляр охватывающего класса.
3. Создайте в данном классе метод, выполняющий новые функции, применяя *разработку посредством тестирования*. Напишите еще один метод, вызывающий новый и старый методы в охватываемом классе.
4. Получите экземпляр охватывающего класса в том месте своего кода, где вам требуется активизировать новое поведение.

Отличие почкования метода от охвата метода весьма тривиально. Почкование метода используется в том случае, если требуется написать новый метод и вызвать его из существующего метода. А охват метода применяется в том случае, если требуется переименовать метод и заменить его новым методом, выполняющим новые функции и вызывающим старый метод. Я обычно пользуюсь почкованием метода, когда код, находящийся в существующем методе, реализует ясный алгоритм. А к охвату метода я прибегаю, когда вновь добавляемое свойство оказывается, на мой взгляд, не менее важным, чем то, что использовалось до него. В этом случае я получаю после охвата новый алгоритм высокого уровня, подобный следующему:

```
public void pay() {
    logPayment();
    Money amount = calculatePay();
    dispatchPayment(amount);
}
```

Совсем иначе выбирается охват класса. Критерии для применения шаблона в этом случае более строгие. Как правило, я прибегаю к охвату класса в двух следующих случаях.

1. Поведение, которое мне нужно ввести, является совершенно независимым, и мне не хотелось бы засорять существующий класс низкоуровневым или неуместным поведением.
2. Класс стал настолько крупным, что дальше некуда. В таком случае я охватываю такой класс, чтобы наметить пути для последующих изменений кода.

Второй случай более трудный и непривычный. Если у вас имеется очень крупный класс с 10 или 15 видами ответственности, то его охват может показаться, на первый взгляд, не совсем логичным для ввода ряда тривиальных функций. На самом деле, если вы не предоставите убедительные доводы своим коллегам по работе, то вас могут просто побить, а

еще хуже — обвинить в профессиональной непригодности и вообще оставить без работы. Поэтому мне бы хотелось помочь вам обосновать свою точку зрения.

Самым трудным препятствием для улучшения крупной базы кода является существующий код. В этом, конечно, нет ничего нового, но ведь речь идет не о том, насколько трудно работать со сложным кодом, а о том, какое представление складывается у вас о таком коде. Если вы потратите целый день на то, чтобы разобраться в скверном коде, то у вас может довольно легко сложиться представление, что такой код так и останется скверным, и вам даже не стоит пытаться сделать его хотя бы чуть лучше. Так, вы можете подумать: "Какой мне смысл улучшать этот фрагмент кода, если 90% своего времени я трачу на то, что бы не увязнуть в этом непроходимом болоте? Конечно, я могу немного улучшить его, но что это даст мне в ближайшей перспективе?" С такой точкой зрения на труднопонимаемый код я согласен лишь отчасти. Ведь если вы будете понемногу, но постоянно улучшать свою систему, то через пару месяцев она станет совсем другой. Однажды утром, придя на работу и морально приготовившись опять продираться сквозь непролазные дебри унаследованного кода, чтобы отыскать в нем хоть что-нибудь осмысленное, вы неожиданно обнаружите, что код стал намного лучше, как будто кто-то реорганизовал его за время вашего отсутствия.

Как только вы почувствуете в такой момент разницу между качественным и недоброкачественным кодом, то станете совсем другим человеком. Более того, с этого момента вы будете ловить себя на том, что вам требуется реорганизация кода просто для того, чтобы облегчить себе жизнь, а не только справиться с очередным заданием. Если вы не испытывали прежде ничего подобного, то все сказанное выше покажется вам просто смешным, но я не раз наблюдал это, работая со многими группами разработчиков. Самое трудное — сделать первые шаги, поскольку они кажутся бессмысленными: "Как? Охватывать класс только для того, чтобы ввести мелкое свойство? Ведь от этого код станет еще хуже, чем прежде. Он станет сложнее". Да, именно так и следует поступить теперь. Но когда вы начнете разрывать зависимости между 10 или 15 видами ответственности в охваченном классе, то сразу оцените правильность и дальновидность шагов, предпринятых вами.

Резюме

В этой главе рассмотрены способы, позволяющие вносить изменения, не подвергая существующие классы тестированию. Им трудно дать надлежащую оценку с точки зрения проектирования. Как правило, они позволяют нам отделить новые виды ответственности от старых. Иными словами, мы начинаем постепенно переходить к более совершенной структуре кода. Но в некоторых случаях оказывается, что единственной причиной для создания класса было наше желание написать новый код вместе с тестами, поскольку мы оказались неготовыми к тому, чтобы тратить время на тестирование существующего класса. Такая ситуация вполне реальна. Когда разработчики делают нечто подобное во время работы над проектом, то конструкции старых крупных классов постепенно обрастают новыми классами и методами. И тогда происходит нечто весьма любопытное. Наскучив уклоняться в сторону от старых конструкций, разработчики начинают тестировать их код и находят в нем что-то знакомое. Ведь они уже не раз просматривали крупные нетестированные классы, чтобы выявить в них места для наращивания нового кода. Но это лишь самая простая часть дела, а настоящие трудности их ждут впереди. Устав, наконец, от попыток привести весь свой код в какой-то порядок, они уже не могут смотреть на него как на заброшенное жилище, которое хочется поскорее покинуть. Если и вы окажетесь в таком же положении, то для начала обратитесь к материалу глав 9 и 20.

Глава 7

Изменения до бесконечности

Сколько же времени требуется на изменение кода? Точного ответа на этот вопрос не существует. В проектах с особенно неясным кодом многочисленные изменения требуют немало времени. Ведь нам нужно просмотреть код, понять все последствия его изменения, а затем сделать их. Если же код оказывается более ясным, то изменения вносятся в него быстрее, но на участках очень запутанного кода для этого потребуется намного больше времени. В некоторых проектах ситуация с изменением кода оказывается еще хуже. Даже для самых простых изменений в коде разработчикам требуется много времени. В таких группах разработчики выясняют свойства, которые им требуется дополнить, наглядно представляют себе места для внесения изменений, переходят непосредственно к коду, вносят изменения за считанные минуты и, тем не менее, они не в состоянии целыми часами выпустить измененную ими версию программного обеспечения.

Рассмотрим причины подобных затруднений и некоторые возможные пути их решения.

Понимание кода

По мере своего нарастания код в проекте постепенно достигает критической массы, когда его перестают понимать сами разработчики. Соответственно увеличивается и время, которое требуется на выяснение тех изменений, которые требуются в коде.

Отчасти это оказывается неизбежным. Когда мы вводим код в систему, можно делать это в существующих классах, методах или функциях или же создавать новые классы, методы или функции. Но в любом случае нам требуется время, чтобы понять, как внести изменения, если мы незнакомы с контекстом.

Тем не менее между удобными в сопровождении системами и унаследованными системами имеется одно существенное различие. На то чтобы выяснить, как внести изменения в удобную для сопровождения систему, требуется некоторое время, но после этого сами изменения, как правило, вносятся легко, а работать с системой становится еще удобнее. На то чтобы выяснить, как внести изменения в унаследованную систему, требуется немало времени, а изменения в нее вносятся трудно. Кроме того, внесение изменений в унаследованную систему нередко оказывается затруднительным из-за возникающего ощущения недостаточного ее понимания. В худшем случае создается впечатления, что для полного понимания такой системы потребуется слишком много времени, и тогда для изменения кода в ней приходится действовать на ощупь, надеясь преодолеть в конце концов все возникшие трудности.

Работать с системами, которые разделяются на мелкие, вполне понятные части, намного проще и быстрее. Если понимание кода представляет для вас большие трудности в работе над конкретным проектом, обратитесь к материалу глав 16 и 17, где даются некоторые рекомендации относительно того, как действовать в подобных случаях.

Время задержки

Изменения нередко делаются долго по еще одной весьма распространенной причине — времени задержки. Это время, которое проходит от момента внесения изменений в код до момента реальной ответной реакции на эти изменения. Когда писались эти строки, очередная экспедиция достигла Марса, и на поверхность этой планеты был спущен марсоход Mars rover Spirit для ее съемки. При передаче этих фотографий сигналы с Марса достигали Земли лишь семь минут спустя. Правда, на борту марсохода было установлена программно управляемая система наведения, с помощью которой он перемещался по поверхности Марса. Представьте себе, насколько трудно было бы управлять марсоходом вручную с Земли, учитывая задержку 14 минут на прохождение управляющей команды с Земли и ответной реакции с Марса. Такое управление кажется совершенно неэффективным, не так ли? Но именно так многим из нас приходится работать теперь, разрабатывая программное обеспечение. Мы вносим в программу определенные коррективы, запускаем ее компоновку и затем смотрим, что из этого получится. К сожалению, у нас нет программно управляемой системы наведения, чтобы обходить различные препятствия в структуре программы, в том числе непрохождение тестов. Вместо этого мы пытаемся связать целый ряд изменений воедино и внести их все сразу, чтобы как можно реже прибегать к компоновке программы. Если изменения сделаны удачно, мы двигаемся дальше, хотя и не быстрее, чем марсоход. Если же мы наталкиваемся на препятствия, то продвигаемся еще медленнее.

Самое печальное, что во многих языках программирования такая работа оказывается совершенно ненужной. Это просто напрасная трата времени. Многие основные языки программирования допускают разрывание зависимостей, перекомпиляцию и тестирование любого фрагмента кода менее чем за 10 секунд. А если члены группы разработки постараются, то могут зачастую сократить это время до 5 секунд. Из этого следует важный вывод: каждый класс или модуль может и должен быть откомпилирован отдельно от других классов или модулей системы в собственных средствах тестирования. В этом случае ответная реакция на изменения в коде наступает очень быстро, что способствует ускорению самого процесса разработки.

Человеческому разуму присущи любопытные особенности. Если нам приходится выполнять короткое задание (продолжительностью 5–10 секунд) и мы можем совершить очередной шаг только раз в минуту, то мы обычно так и поступаем, а затем делаем паузу. Если же нам нужно решить, что делать на следующем шаге, то мы начинаем планировать его. Спланировав следующий шаг, наш ум освобождается до тех пор, пока этот шаг не будет сделан. Если бы нам удалось сократить время между последовательными шагами в нашей умственной деятельности с одной минуты до нескольких секунд, то ее качество стало бы совершенно другим. Мы смогли бы тогда быстрее реагировать на ответную реакцию разрабатываемой системы и оперативнее оценивать разные варианты ее изменения. Наша работа стала бы в большей степени похожей на езду на автомашине, чем на ожидание автобуса на остановке. Это позволило бы нам добиться большей сосредоточенности на работе, поскольку нам не нужно было бы постоянно ждать следующей возможности сделать что-нибудь. А самое главное, что мы тратили бы намного меньше времени на выявление и исправление своих ошибок.

Что же мешает нам работать подобным образом постоянно? Те, кто программирует на интерпретируемых языках, зачастую могут работать подобным образом, получая практически мгновенную ответную реакцию на свои действия. А для остальных, программирующих на транслируемых языках, главной задержкой будет зависимость, т.е. необходимость компилировать не только то, что нас интересует, но и то, что нам не нужно.

Разрыв зависимостей

Несмотря на все трудности разрыва зависимостей, мы все же можем это сделать. Первым шагом в этом направлении в объектно-ориентированном коде обычно является попытка получить экземпляры классов, которые требуется ввести в средства тестирования. В простейшем случае мы можем сделать это, импортируя или включая объявление классов, от которых зависят тестируемые классы. А в более сложном случае на помощь приходят методы, рассматриваемые в главе 9. Если же нам удастся создать объект определенного класса в средствах тестирования, то, скорее всего, нам придется разорвать и другие зависимости, чтобы протестировать отдельные методы в данном классе. О подобных случаях речь пойдет в главе 10.

Если класс, который требуется изменить, уже находится в средствах тестирования, то, как правило, время цикла правки, компиляции, компоновки и тестирования существенно сокращается. Обычно затраты времени на выполнение почти всех методов относительно невелики по сравнению с затратами на выполнение методов, которые их вызывают, особенно если это вызовы с обращением к таким внешним ресурсам, как база данных, аппаратные средства или инфраструктура каналов связи. Те случаи, когда этого не происходит, обычно относятся к методам с весьма интенсивными вычислениями, и тогда полезными оказываются способы, рассматриваемые в главе 22.

Сами изменения часто оказываются несложными, но нередко те, кто работает с унаследованным кодом, останавливаются на первом же шаге, пытаясь ввести класс в средства тестирования. В некоторых системах для этого приходится прилагать немалые усилия. Одни классы оказываются довольно крупными, а другие — со столь многочисленными зависимостями, что они, на первый взгляд, просто подавляют те функции, с которыми требуется работать. В подобных случаях целесообразно рассмотреть возможность выделить крупный фрагмент кода и подвергнуть его тестированию. Подробнее об этом речь пойдет в главе 12, а в этой главе рассматриваются способы обнаружения *точек сужения*, т.е. мест, где проще писать тесты.

В остальной части этой главы показывается, как приступить к изменению порядка организации кода, чтобы упростить его компоновку.

Зависимости компоновки

Если в объектно-ориентированной системе имеются группы классов, которые требуется скомпоновать быстрее, то для этой цели следует, прежде всего, выявить те зависимости, которые этому препятствуют. Обычно для этого достаточно попытаться использовать отдельные классы в средствах тестирования. Практически все возникающие при этом затруднения являются результатом некоторой зависимости, которую следует разорвать. После выполнения отдельных классов в средствах тестирования по-прежнему остаются некоторые зависимости, способные оказывать отрицательное влияние на время компиляции. В связи с этим целесообразно проанализировать все, что зависит от объектов, экземпляры которых удалось получить. Такие объекты придется перекомпилировать при перекомпоновке системы. Как же свести эти операции к минимуму?

Для этого можно, в частности, извлечь интерфейсы из группы для тех классов, которые используются классами, находящимися за пределами данной группы. Во многих интегрированных средах разработки (IDE) для извлечения интерфейса достаточно выбрать сначала нужный класс, а затем из списка, появляющегося в меню, все методы этого класса,

которые должны стать составной частью нового интерфейса. После этого соответствующие инструментальные средства предоставляют возможность указать имя нового интерфейса, а также заменить ссылки на исходный класс ссылками на этот интерфейс везде, где это можно сделать в базе коде. Такая возможность оказывается необычайно полезной. В C++ *извлечение средства реализации* осуществляется немного проще, чем *извлечение интерфейса*. В этом случае не нужно изменять повсюду имена ссылок, но приходится менять места, в которых создаются экземпляры старого класса (подобнее об этом — в главе 25).

В подобной группе тестируемых классов можно изменить физическую структуру проекта, чтобы упростить его компоновку. Для этого группы классов переносятся в новый пакет или библиотеку. В результате сама компоновка усложняется, т.е. после разрыва зависимостей и выделения классов в новые пакеты или библиотеки средние затраты на перекомпоновку всей системы немного возрастают, но в то же время существенно сокращается время самой компоновки.

Рассмотрим следующий пример. На рис. 7.1 приведена небольшая группа классов, взаимодействующих в одном пакете.

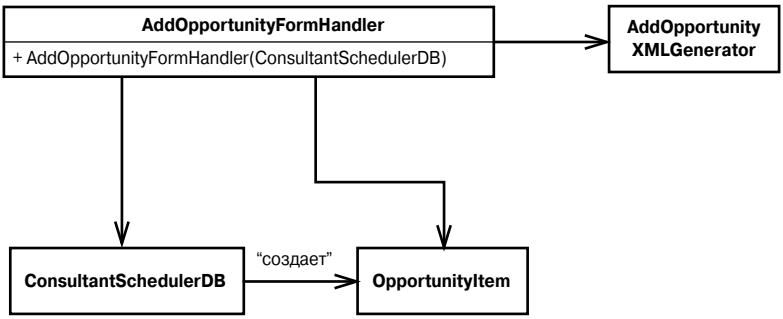


Рис. 7.1. Классы обработки возможностей

Нам требуется внести некоторые изменения в класс `AddOpportunityFormHandler` (обработчик формы ввода возможностей), но в то же время было бы неплохо ускорить и компоновку. Прежде всего можно попытаться получить экземпляр класса `AddOpportunityFormHandler`. Но, к сожалению, классы, от которых он зависит, вполне конкретны. В частности, классу `AddOpportunityFormHandler` требуются классы `ConsultantSchedulerDB` (планировщик базы данных консультанта) и `AddOpportunityXMLGenerator` (генератор кода XML добавляемой возможности). Но вполне возможно, что оба эти класса, в свою очередь, зависят от других классов, не показанных на блок-схеме, приведенной на рис. 7.1.

Если мы попытаемся получить экземпляр класса `AddOpportunityFormHandler`, то еще неизвестно, сколько еще классов он за собой потянет? Во избежание этого мы можем приступить к разрыванию зависимостей. Первой оказывается зависимость от класса `ConsultantSchedulerDB`. Нам нужно создать экземпляр этого класса, чтобы передать его конструктору класса `AddOpportunityFormHandler`. Использовать класс `ConsultantSchedulerDB` было бы не совсем удобно, поскольку он связан с базой данных, а во время тестирования нам это не нужно. Но мы могли бы воспользоваться *извлечением средства реализации*, чтобы разорвать зависимость, как показано на рис. 7.2.

Теперь, когда класс `ConsultantSchedulerDB` стал интерфейсом, мы можем создать экземпляр класса `AddOpportunityFormHandler`, используя фиктивный объект, реализующий интерфейс `ConsultantSchedulerDB`. Любопытно, что, разрывая эту зависи-

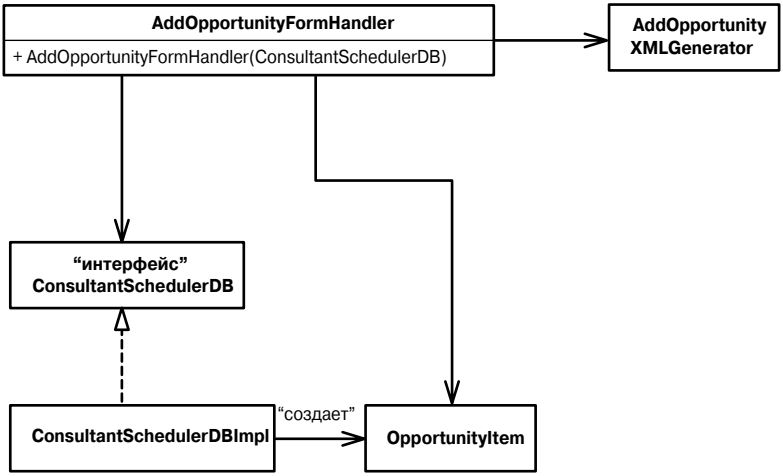


Рис. 7.2. Извлечение средства реализации из класса `ConsultantSchedulerDB`

мость, мы добились ускорения компоновки при определенных условиях. В следующий раз, когда нам потребуется видоизменить интерфейс `ConsultantSchedulerDBImpl` (реализация планировщика базы данных консультанта), перекомпилировать класс `AddOpportunityFormHandler` нам уже не придется. Почему? А потому, что он уже не зависит напрямую от кода в интерфейсе `ConsultantSchedulerDBImpl`. Мы можем внести столько изменений в файл интерфейса `ConsultantSchedulerDBImpl`, сколько потребуется, но если мы не сделаем нечто, вынуждающее нас изменить интерфейс `ConsultantSchedulerDBImpl`, то перекомпоновывать класс `AddOpportunityFormHandler` нам уже не придется.

При желании мы можем еще больше застраховаться от вынужденной перекомпиляции. На рис. 7.3. показана структура системы, к которой мы можем прийти, если мы применим *извлечение средства реализации* из класса `OpportunityItem` (элемент возможности).

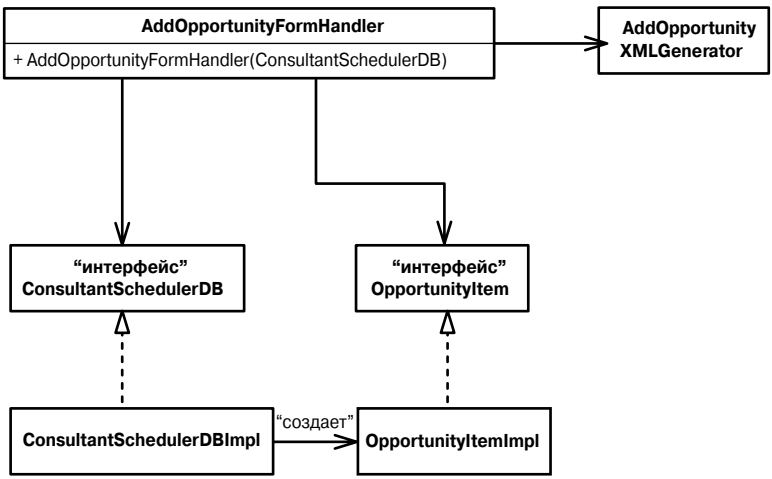


Рис. 7.3. Извлечение средства реализации из класса `OpportunityItem`

Теперь класс AddOpportunityFormHandler вообще не зависит от первоначального кода в классе OpportunityItem. Такой разрыв зависимостей можно в известной степени сравнить с размещением в коде своеобразного “брандмауэра компиляции”. Любые изменения в интерфейсах ConsultantSchedulerDBImpl и OpportunityItemImpl не потребуют вынужденной перекомпиляции ни класса AddOpportunityFormHandler, ни его пользователей. Если бы нам потребовалось сделать это явно в структуре пакетов приложения, то мы могли бы разделить наш проект на отдельные пакеты так, как показано на рис. 7.4.

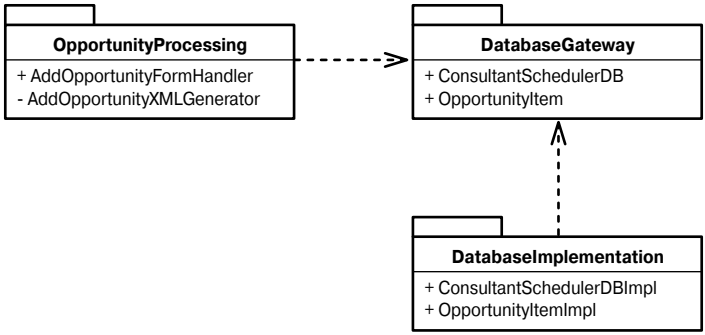


Рис. 7.4. Реорганизованная структура пакетов

Теперь у нас имеется пакет OpportunityProcessing (обработка возможностей), который действительно не зависит от реализации базы данных. Какие бы тесты мы ни написали и поместили в этот пакет, они должны выполняться быстро, а сам пакет не придется перекомпилировать после изменения кода в классах реализации базы данных.

Принцип инверсии зависимостей

Если код и зависит от интерфейса, то такая зависимость, как правило, минимальна и ненавязчива. Изменять код не нужно, если только не изменяется сам интерфейс, а это обычно происходит намного реже, чем в коде, который зависит от интерфейса. Наличие интерфейса позволяет править существующие классы или же вводить новые классы, реализующие этот интерфейс, не оказывая никакого влияния на использующий его код.

По этой причине лучше иметь зависимость от интерфейса или абстрактного класса, чем от конкретных классов. При наличии зависимости от менее изменчивых объектов уменьшается вероятность того, что конкретные изменения приведут к массовой перекомпиляции.

До сих пор мы приняли ряд мер, чтобы избежать перекомпиляции класса AddOpportunityFormHandler после видоизменения классов, от которых он зависит. Благодаря этому ускоряется компоновка, но это лишь полдела. Мы можем также ускорить компоновку кода, который зависит от класса AddOpportunityFormHandler. Обратимся вновь к структуре пакетов, приведенной на рис. 7.5.

Класс AddOpportunityFormHandler является единственным общедоступным выходным (нетестируемым) классом в пакете OpportunityProcessing. Любые классы в других пакетах, которые зависят от него, должны быть перекомпилированы после изменений в нем. Эту зависимость мы можем разорвать, используя *извлечение средства реализации* или же *извлечение интерфейса* из класса AddOpportunityFormHandler. В этом

случае классы из других пакетов будут зависеть уже от соответствующих интерфейсов. Благодаря такому разрыву зависимостей мы, по существу, предохраняем всех пользователей данного пакета от перекомпиляции при внесении в него любых изменений.

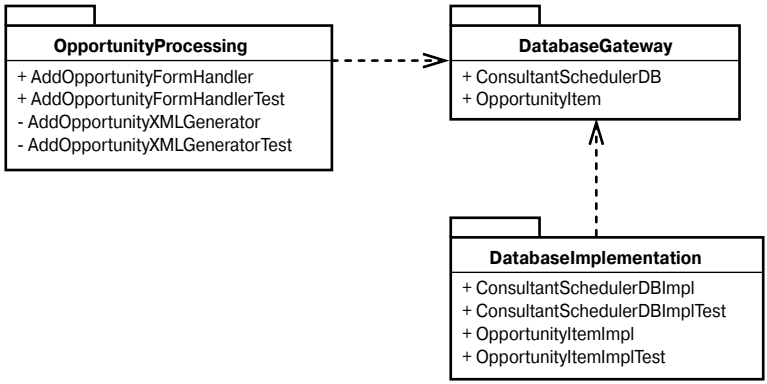


Рис. 7.5. Структура пакетов

Итак, мы можем разорвать зависимости и распределить классы по разным пакетам, чтобы упростить их компоновку. И сделать это очень даже стоит. Ведь благодаря этому заметно ускоряется перекомпоновка и выполнение тестов, а следовательно, и ответная реакция системы по ходу ее разработки. Как правило, это означает сокращение ошибок и всяческих неприятностей, хотя и дается небесплатно. Появление дополнительных интерфейсов и пакетов связано с определенными концептуальными издержками. Справедлива ли такая цена в сравнении с иной альтернативой? Да, справедлива. Иногда дополнительные интерфейсы и пакеты несколько замедляют поиск отдельных элементов, но все равно работать с ними намного удобнее и проще.

После ввода дополнительных интерфейсов и пакетов в проект для разрывания зависимостей время перекомпоновки всей системы немного увеличивается, поскольку появляется больше файлов для компиляции. Но в среднем время рациональной компоновки системы может заметно сократиться в зависимости от того, что именно *требуется* перекомпилировать.

Приступая к оптимизации среднего времени компоновки, вы получаете в конечном итоге участки кода, с которыми очень легко работать. И хотя отдельно скомпилировать и протестировать небольшую совокупность классов не всегда оказывается просто, не следует все же забывать, что для данной совокупности классов это нужно сделать только один раз, пожиная в дальнейшем плоды своих трудов.

Резюме

Способы, рассмотренные в этой главе, можно использовать для ускорения времени компоновки небольших групп классов, но это лишь малая доля того, чего можно добиться, используя интерфейсы и пакеты для управления зависимостями. В книге Роберта К. Мартина, *Быстрая разработка программ. Принципы, примеры, практика* (Издательский

дом “Вильямс”, 2004) представлены другие методы ускорения времени компоновки и управления зависимостями, которые следует взять на вооружение каждому разработчику программного обеспечения.

Глава 8

Как ввести новое свойство

Этот вопрос относится к самым абстрактным в данной книге, поскольку его решение очень зависит от конкретной предметной области. И по этой причине я поначалу не хотел включать его рассмотрение в данную книгу. Тем не менее, независимо от применяемого подхода к проектированию или конкретных ограничений, накладываемых на него, существует ряд способов, позволяющих упростить задачу ввода нового свойства в программу.

Итак, начнем обсуждение данного вопроса с контекста. Одна из самых больших трудностей работы с унаследованным кодом заключается в отсутствии тестов вокруг большей части такого кода. Более того, разместить их по местам не так-то просто. В связи с этим у многих разработчиков возникает искушение прибегнуть к способам, представленным в главе 6. Эти способы (почкования и охвата) можно использовать для ввода кода без тестов, но такой подход связан с целым рядом опасностей: как явных, так и неявных. Прежде всего при почковании и охвате мы не видоизменяем код существенно, и поэтому он не становится от этого лучше хотя бы на время. Кроме того, существует опасность дублирования кода. Если добавляемый код дублирует код, существующий на непроверенных участках, то он может просто залежаться и испортиться. Более того, мы можем даже не осознавать того, что дублируем код до тех пор, пока не зайдем слишком далеко, внося в него изменения. И еще одна опасность связана с боязнью и отказом от изменения кода. Боязнь возникает в связи с тем, что мы просто не в состоянии изменить конкретный фрагмент кода и упростить его для удобства дальнейшей работы с ним, а отказ вызван тем, что целые фрагменты кода не становятся лучше в результате внесенных изменений. Такая боязнь мешает принятию разумного решения. Об этом напоминают жалкие остатки почкования и охвата в коде.

Как известно, трудностям лучше противостоять смело, чем уклоняться от них. Если код можно подвергнуть тестированию, значит, стоит воспользоваться способами, представленными в этой главе, для удачного продвижения вперед в работе над кодом. О способах размещения тестов по местам речь пойдет в главе 13, а о преодолении трудностей, связанных с зависимостями, — в главах 9 и 10.

Разместив тесты по местам, мы оказываемся в лучшем положении для ввода новых свойств. Ведь мы опираемся на прочное основание.

Разработка посредством тестирования

Самым эффективным способом из тех, что я знаю для ввода новых свойств в программное обеспечение, является *разработка посредством тестирования* (TDD). Суть этого способа такова: мы придумываем сначала метод, который поможет нам решить часть стоящей перед нами задачи, а затем пишем для него контрольный пример непрохождения теста. Сам метод еще не существует, но если мы сумеем написать для него тест, то будем лучше понимать, каким именно должен быть код для этого метода.

Разработка посредством тестирования выполняется в такой последовательности.

1. Написание контрольного примера непрохождения теста.
2. Подготовка к компиляции.

3. Подготовка к прохождению.
4. Исключение дублирования кода.
5. Повтор.

Рассмотрим следующий пример. Допустим, что мы работаем над финансовым приложением и нам требуется класс, в котором нужно применить эффективный математический аппарат для проверки возможности продажи определенных товаров. Для этой цели нам потребуется класс Java, в котором выполняются расчеты так называемого первого статистического момента относительно определенной точки. У нас еще нет метода, который должен это делать, но мы знаем, что можем написать контрольный пример для данного метода. Если нам известен порядок расчетов, то мы знаем, что для данных в коде теста ответ должен быть равен `-0,5`.

Написание контрольного примера непрохождения теста

Ниже приведен тест для проверки нужных нам функций.

```
public void testFirstMoment() {
    InstrumentCalculator calculator = new InstrumentCalculator();
    calculator.addElement(1.0);
    calculator.addElement(2.0);

    assertEquals(-0.5, calculator.firstMomentAbout(2.0), TOLERANCE);
}
```

Подготовка к компиляции

Итак, мы написали тест, но он не компилируется, поскольку у нас нет метода `firstMomentAbout` (первый относительный момент) в классе `InstrumentCalculator` (калькулятор инструментов). Но мы можем ввести его как пустой метод. А поскольку нам требуется, чтобы тест не прошел, то мы должны возвратить значение `NaN` двойной длины (типа `double`), которое явно не равно предполагаемому значению `-0,5`.

```
public class InstrumentCalculator
{
    double firstMomentAbout(double point) {
        return Double.NaN;
    }
    ...
}
```

Подготовка к прохождению

Разместив тест на месте, мы готовы написать код, чтобы подготовить тест к прохождению.

```
public double firstMomentAbout(double point) {
    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
```

```
        double element = ((Double)(it.next())).doubleValue();
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

Это слишком большое количество кода, который требуется написать для прохождения теста при разработке посредством тестирования. Как правило, он пишется более мелкими фрагментами. Но если вы уверены в применяемом алгоритме, то можете писать код и такими крупными фрагментами.

Исключение дублирования кода

Имеется ли в данном коде дублирование? Нет, не имеется. Следовательно, вы можем переходить к следующему контрольному примеру.

Написание контрольного примера непрохождения теста

Благодаря написанному только что коду тест проходит, но он годится не для всех случаев. Так, если в операторе `return` случайно произойдет деление на нуль, то что нам делать в этом случае? Что нужно возвратить, если нет элементов? В таком случае нам нужно сгенерировать исключение. Результаты не будут иметь для нас особого значения до тех пор, пока в нашем списке элементов не появятся данные.

Следующий тест имеет особое назначение. Он не проходит, если исключение в связи с неверным основанием (`InvalidBasisException`) не сгенерировано, и проходит, если не генерируется ни это, ни любое другое исключение. Если выполнить этот тест, то он не пройдет, поскольку при делении на нуль в методе `firstMomentAbout` генерируется исключение в связи с арифметической операцией (`ArithmeticException`).

```
public void testFirstMoment() {
    try {
        new InstrumentCalculator().firstMomentAbout(0.0);
        fail("ожидалось исключение InvalidBasisException");
    }
    catch (InvalidBasisException e) {
    }
}
```

Подготовка к компиляции

Для этой цели нам придется изменить объявление метода `firstMomentAbout`, чтобы в нем генерировалось исключение `InvalidBasisException`.

```
public double firstMomentAbout(double point)
        throws InvalidBasisException {

    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
```

```
        double element = ((Double)(it.next())).doubleValue();
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

Но этот код все равно не компилируется. Ошибки, сообщаемые компилятором, указывают на то, что нам нужно генерировать исключение, если оно перечислено в объявлении. Поэтому мы должны написать код, определяющий условия для генерирования исключения.

```
public double firstMomentAbout(double point)
        throws InvalidBasisException {

    if (element.size() == 0)
        throw new InvalidBasisException("элементы отсутствуют");

    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)(it.next())).doubleValue();
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

Подготовка к прохождению

Теперь все наши тесты проходят.

Исключение дублирования кода

В данном случае дублирование кода отсутствует.

Написание контрольного примера непрохождения теста

Следующий фрагмент кода, который нам требуется написать, относится к методу, рассчитывающему второй статистический момент относительно определенной точки. На самом деле код для расчета этого момента является лишь разновидностью кода для расчета первого статистического момента. Ниже приведен тест, который позволяет нам перейти к написанию такого кода. В данном случае ожидается возврат значения `0,5`, а не `-0,5`, и поэтому нам нужно написать новый тест для метода `secondMomentAbout` (второй относительный момент), который еще не существует.

```
public void testSecondMoment() throws Exception {
    InstrumentCalculator calculator = new InstrumentCalculator();
    calculator.addElement(1.0);
    calculator.addElement(2.0);

    assertEquals(0.5, calculator.secondMomentAbout(2.0), TOLERANCE);
}
```

Подготовка к компиляции

Для подготовки теста к компиляции придется добавить объявление метода secondMomentAbout. Это можно было бы сделать таким же образом, как для и метода firstMomentAbout, но ведь код для расчета второго статистического момента лишь незначительно отличается от кода для расчета первого статистического момента.

Следующая строка кода в методе firstMomentAbout:

```
numerator += element - point;
```

должна быть заменена такой строкой в методе secondMomentAbout:

```
numerator += Math.pow(element - point, 2.0);
```

В целом, для такого рода расчетов имеется общий шаблон. Расчеты *n*-го статистического момента проводятся с помощью следующего выражения:

```
numerator += Math.pow(element - point, N);
```

Приведенная выше строка кода в методе firstMomentAbout оказывается вполне пригодной для расчетов, поскольку выражение element - point в ней равнозначно выражению Math.pow(element - point, 1.0).

В данный момент у нас два варианта выбора. Обратив внимание на общность обоих упомянутых выше методов, мы можем написать сначала общий метод, воспринимающий конкретную точку, относительно которой выполняются расчеты статистического момента, а также значение N, определяющее порядок этого момента, и затем заменить каждый пример применения метода firstMomentAbout(double) вызовом нового общего метода. Конечно, мы могли бы это сделать, но тогда при вызове данного метода пришлось бы указывать значение N, а нам бы не хотелось, чтобы клиенты предоставляли при этом произвольное значение N. Но похоже, что мы уклонились немного в сторону, поэтому остановимся пока что на этом, чтобы завершить то, с чего мы начали, а именно: подготовить тест к компиляции. К обобщению методов мы можем вернуться позднее, если сочтем это по-прежнему актуальным.

Для подготовки к компиляции нам достаточно скопировать метод firstMomentAbout и переименовать его в метод secondMomentAbout.

```
public double secondMomentAbout(double point)
        throws InvalidBasisException {

    if (elements.size() == 0)
        throw new InvalidBasisException("элементы отсутствуют");

    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)(it.next())).doubleValue();
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

Подготовка к прохождению

Данный код не проходит тестирование. В таком случае мы можем вернуться к подготовке теста к прохождению, внеся в код следующие изменения.

```
public double secondMomentAbout(double point)
        throws InvalidBasisException {

    if (elements.size() == 0)
        throw new InvalidBasisException("элементы отсутствуют");

    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)(it.next())).doubleValue();
        numerator += Math.pow(element - point, 2.0);
    }
    return numerator / elements.size();
}
```

Возможно, вас несколько смутило такое вырезание, копирование и вставка кода, но в дальнейшем мы постараемся исключить дублирование кода в методе `secondMomentAbout`. Несмотря на то, что мы пишем в данном примере совершенно новый код, копирование нужного фрагмента кода и его видоизменение в новом методе оказывается достаточно эффективным в контексте унаследованного кода. Нередко приходится вводить новые свойства в особенно скверный код, и тогда нам легче понять результаты своих модификаций кода, если мы поместим их в новом месте и сравним их рядом со старым кодом. В дальнейшем можем исключить дублирование кода, чтобы аккуратно свести новый код в отдельный класс, или же просто отказаться от модификации кода и попытаться выполнить ее иначе, зная, что в нашем распоряжении по-прежнему имеется старый код для изучения и анализа.

Исключение дублирования кода

А теперь, когда оба теста проходят, мы можем перейти к следующей стадии: исключению дублирования кода. Как же нам сделать это?

Для этого можно, в частности, извлечь все тело метода `secondMomentAbout`, переименовать его на `nthMomentAbout` и дополнить его параметром `N`.

```
public double secondMomentAbout(double point)
        throws InvalidBasisException {
    return nthMomentAbout(point, 2.0);
}

private double nthMomentAbout(double point, double n)
        throws InvalidBasisException {

    if (elements.size() == 0)
        throw new InvalidBasisException("элементы отсутствуют");

    double numerator = 0.0;
```

```
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)(it.next())).doubleValue();
        numerator += Math.pow(element - point, n);
    }
    return numerator / elements.size();
}
```

Если мы теперь выполним наши тесты, то обнаружим, что они по-прежнему проходят. Далее мы можем вернуться к методу `firstMomentAbout` и заменить его тело вызовом метода `nthMomentAbout`.

```
public double firstMomentAbout(double point)
        throws InvalidBasisException {
    return nthMomentAbout(point, 1.0);
}
```

Эта последняя стадия исключения дублирования очень важна. Ведь мы можем быстро и просто ввести новые свойства в код, копируя целые блоки кода. Но если мы не исключим в дальнейшем дублирование кода, то добавим немало хлопот себе на стадии разработки и модификации программного обеспечения и другим на стадии его сопровождения. С другой стороны, при наличии всех необходимых тестов на местах мы можем без особого труда исключить дублирование кода. Мы убедились в этом на рассматриваемом здесь примере, но в данном случае тесты оказались в нашем распоряжении лишь потому, что мы с самого начала пользовались разработкой посредством тестирования. В унаследованном коде особое значение имеют тесты, которые мы пишем и размещаем вокруг существующего кода по ходу разработки посредством тестирования. Разместив их по местам, мы можем свободно писать любой код, который требуется для ввода нового свойства, зная, что его можно добавить к остальному коду, не усугубляя положение.

Разработка посредством тестирования и унаследованный код

Одно из самых ценных свойств разработки посредством тестирования состоит в том, что она позволяет нам поочередно сосредоточивать основное внимание на решении отдельных задач. Мы либо пишем код, либо реорганизуем его, но никогда не делаем и то, и другое одновременно.

Такое разделение особенно ценно в работе с унаследованным кодом, поскольку оно позволяет нам писать новый код независимо от старого.

Написав новый код, мы можем реорганизовать его, исключив любое дублирование нового и старого кода.

Для унаследованного кода разработка посредством тестирования выполняется в следующей расширенной последовательности.

0. *Подготовка к тестированию класса, в который требуется внести изменения.*

1. Написание контрольного примера непрохождения теста.

2. Подготовка к компиляции.

3. Подготовка к прохождению. (*При этом желательно не изменять существующий код.*)

4. Исключение дублирования кода.

5. Повтор.

Программирование по разности

Разработка посредством тестирования никак не связана с объектной ориентацией. На самом деле в примере из предыдущего раздела приведен фрагмент процедурного кода, заключенный в оболочку класса. В объектно-ориентированном программировании у нас имеется другая возможность: использовать наследование, чтобы вводить новые свойства, не видоизменяя класс непосредственно. После ввода нового свойства мы можем точно определить, как нам его интегрировать.

Основной способ сделать это называется *программированием по разности*. Это довольно старый способ, широко обсуждавшийся и применявшийся еще в 1980-е годы, но в 1990-е годы он утратил свою популярность, как только в сообществе ООП было замечено, что наследование может вызывать серьезные осложнения, если пользоваться им чрезмерно. Но это совсем не означает, что мы должны отказаться от наследования. С помощью тестов мы можем без особого труда переходить к другим структурам, если наследование вызывает какие-то осложнения.

Покажем принцип программирования по разности на конкретном примере. Допустим, что мы протестировали класс Java, называемый `MailForwarder` (пересылка почты) и являющийся составной частью программы на языке Java, управляющей списками рассылки. У этого класса имеется метод под названием `getFromAddress` (получить адрес отправителя). В коде это выглядит следующим образом:

```
private InternetAddress getFromAddress(Message message)
        throws MessagingException {

    Address [] from = message.getFrom ();
    if (from != null && from.length > 0)
        return new InternetAddress (from [0].toString ());
    return new InternetAddress (getDefaultFrom());
}
```

Назначение данного метода — извлечь адрес отправителя полученного почтового сообщения и возвратить его, чтобы затем воспользоваться им в качестве адреса отправителя сообщения, пересылаемого по списку получателей.

Он используется только в одном месте, т.е. в следующих строках кода из метода под названием `forwardMessage` (переслать сообщение):

```
MimeMessage forward = new MimeMessage (session);
forward.setFrom (getFromAddress (message));
```

Как нам поступить далее, если потребуется поддержка анонимных списков рассылки в качестве нового требования к программе? Члены этих списков могут отправлять сообщения, но в качестве адреса отправителя в их сообщениях должен быть задан конкретный адрес электронной почты, исходя из значения переменной экземпляра `domain` класса `MessageFowarder` (пересылка сообщений). Ниже приведен контрольный пример непрохождения теста для такого изменения в программе (когда тест выполняется, переменной `expectedMessage` (ожидаемое сообщение) присваивается сообщение, пересылаемое объектом класса `MessageFowarder`).

```
public void testAnonymous () throws Exception {
    MessageForwarder forwarder = new MessageForwarder();
    forwarder.forwardMessage (makeFakeMessage());
```

```
    assertEquals ("anon-members@" + forwarder.getDomain(),
    expectedMessage.getFrom ()[0].toString());
}
```

Далее мы выполняем подклассификацию согласно рис. 8.1.

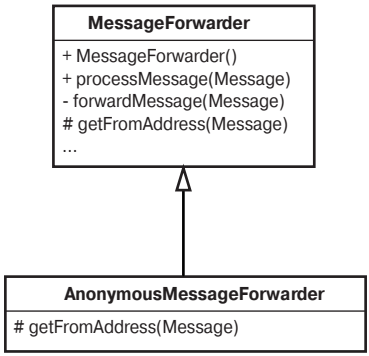


Рис. 8.1. Подклассификация класса `MessageFowarder`

В данном случае метод `getFromAddress` сделан защищенным, а не частным в классе `MessageFowarder`. Затем он был переопределен в классе `AnonymousMessageForwarder` (пересылка анонимных сообщений). В этом классе он выглядит следующим образом:

```
protected InternetAddress getFromAddress(Message message)
        throws MessagingException {
    String anonymousAddress = "anon-" + listAddress;
    return new InternetAddress(anonymousAddress);
}
```

Что же это нам дает? Мы решили поставленную перед нами задачу, но ввели новый класс в систему для очень простого поведения. А стоило ли выполнять подклассификацию целого класса пересылки сообщений ради одного лишь изменения в нем адреса отправителя? В долгосрочной перспективе, пожалуй, и не стоило, но в то же время это позволило нам быстро пройти тест. А если этот тест проходит, то мы можем быть уверены в том, что новое поведение сохранится, если мы решим внести изменения в структуру кода.

```
public void testAnonymous () throws Exception {
    MessageForwarder forwarder = new AnonymousMessageForwarder();
    forwarder.forwardMessage (makeFakeMessage());
    assertEquals ("anon-members@" + forwarder.getDomain(),
        expectedMessage.getFrom ()[0].toString());
}
```

Что-то уж слишком просто. В чем же загвоздка? А в том, что если пользоваться данным способом слишком часто и не уделять внимания главным элементам структуры кода, то он начнет быстро деградировать. Для того чтобы увидеть, что может при этом произойти, рассмотрим еще одно изменение. Нам нужно не только пересылать сообщения по списку рассылки получателей, но и отправлять их слепую копию (bcc) ряду других получателей, которых нельзя ввести в официальный список рассылки. Таких получателей назовем внесписковыми.

Такое изменение выглядит достаточно простым. Ведь мы можем вновь выполнить подклассификацию класса `MessageForwarder` и переопределить его метод обработки, чтобы отправлять сообщения по данному месту назначения, как показано на рис. 8.2.

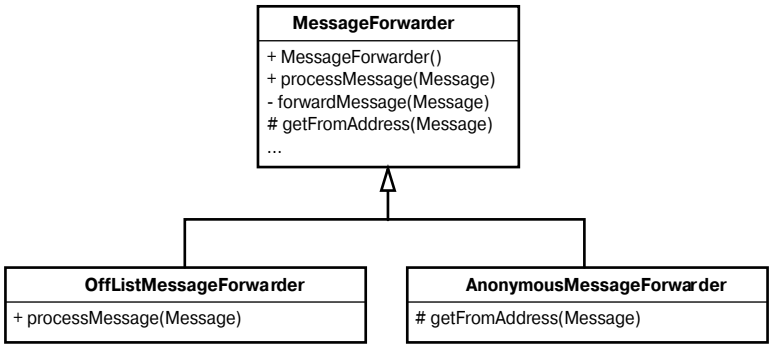


Рис. 8.2. Подклассификация по двум разностям

Такой вариант может оказаться вполне работоспособным, за одним исключением. Что, если в классе `MessageForwarder` нам требуется реализовать две функции — отправку всех сообщений внесписковым получателям и анонимную пересылку сообщений?

Это одна из самых больших проблем, возникающих в связи со слишком широким применением наследования. Если мы поместим свойства в отдельные классы, то они станут доступными нам лишь по очереди.

Как же развязать этот узел? Для этого можно, в частности, отложить на время ввод свойства внесписковых получателей и реорганизовать код, чтобы сделать его более ясным. К счастью, мы написали тест и поместили его в нужном месте, а теперь можем воспользоваться им, чтобы проверить, сохраняется ли поведение при переходе к другому алгоритму.

Свойство анонимной пересылки сообщений мы могли бы реализовать и без подклассификации, сделав его одним из вариантов конфигурации программы. Для этого можно, например, изменить конструктор класса, чтобы он воспринимал совокупность свойств.

```
Properties configuration = new Properties();
configuration.setProperty("anonymous", "true");
MessageForwarder forwarder = new MessageForwarder(configuration);
```

Можем ли мы подготовить наш тест к прохождению? Рассмотрим тест еще раз.

```
public void testAnonymous () throws Exception {
    MessageForwarder forwarder = new AnonymousMessageForwarder();
    forwarder.forwardMessage (makeFakeMessage());
    assertEquals ("anon-members@" + forwarder.getDomain(),
        expectedMessage.getFrom ()[0].toString());
}
```

В настоящий момент этот тест проходит. В классе `AnonymousMessageForwarder` переопределяется метод `getFrom` из класса `MessageForwarder`. А что, если изменить метод `getFrom` в классе `MessageForwarder` следующим образом?

```
private InternetAddress getFromAddress(Message message)
        throws MessagingException {

    String fromAddress = getDefaultFrom();
```

```
    if (configuration.getProperty("anonymous").equals("true")) {
        fromAddress = "anon-members@" + domain;
    }
    else {
        Address [] from = message.getFrom ();
        if (from != null && from.length > 0) {
            fromAddress = from [0].toString ();
        }
    }
    return new InternetAddress (fromAddress);
}
```

Теперь в классе `MessageForwarder` имеется метод `getFrom`, способный обрабатывать как анонимные, так и обычные сообщения. Для проверки этого факта мы можем закомментировать переопределение метода `getFrom` в классе `AnonymousMessageForwarder` и посмотреть, проходят ли тесты в этом случае.

```
public class AnonymousMessageForwarder extends MessageForwarder
{
/*
    protected InternetAddress getFromAddress(Message message)
            throws MessagingException {
        String anonymousAddress = "anon-" + listAddress;
        return new InternetAddress(anonymousAddress);
    }
*/
}
```

Конечно, они проходят.

Класс `AnonymousMessageForwarder` нам уже не понадобится, и поэтому мы можем его удалить. Для этого нам нужно отыскать в коде все места, где создается класс `AnonymousMessageForwarder`, и заменить вызов его конструктора вызовом конструктора, воспринимающего упомянутую выше совокупность свойств.

Эту совокупность свойств можно использовать и для ввода нового свойства в программу. Для этого достаточно выбрать свойство, активизирующее обработку сообщений для внесписковых получателей.

Можно ли считать дело сделанным? Не совсем. Мы внесли в код метода `getFrom` в классе `MessageForwarder` некоторый беспорядок, но, поскольку у нас имеются соответствующие тесты, мы можем очень быстро извлечь этот метод и немного поправить его. В настоящий момент он выглядит следующим образом.

```
private InternetAddress getFromAddress(Message message)
        throws MessagingException {

    String fromAddress = getDefaultFrom();
    if (configuration.getProperty("anonymous").equals("true")) {
        fromAddress = "anon-members@" + domain;
    }
    else {
        Address [] from = message.getFrom ();
        if (from != null && from.length > 0)
            fromAddress = from [0].toString ();
```

```
    }
    return new InternetAddress (fromAddress);
}
```

А после реорганизации кода он выглядит следующим образом.

```
private InternetAddress getFromAddress(Message message)
        throws MessagingException {

    String fromAddress = getDefaultFrom();
    if (configuration.getProperty("anonymous").equals("true")) {
        from = getAnonymousFrom();
    }
    else {
        from = getFrom(Message);
    }
    return new InternetAddress (from);
}
```

Теперь код выглядит чуть более ясным, но в то же время свойства рассылки анонимных сообщений и обработки сообщений для внесписковых получателей сведены в класс `MessageForwarder`. Плохо ли это с точки зрения *принципа единственной ответственности*? Возможно, и плохо — все зависит от того, насколько крупным получается код, связанный с такой ответственностью, и насколько запутанным он оказывается по сравнению с остальным кодом. В данном случае определить анонимность списка не так уж и сложно. Этому способствует выбранный нами подход к совокупности свойств. А что мы будем делать, если таких свойств окажется много и код в классе `MessageForwarder` начнет засоряться условными операторами? В таком случае можно воспользоваться отдельным классом вместо совокупности свойств. В частности, мы можем создать класс под названием `MailingConfiguration` (конфигурация рассылки), чтобы хранить в нем совокупность свойств (рис. 8.3).

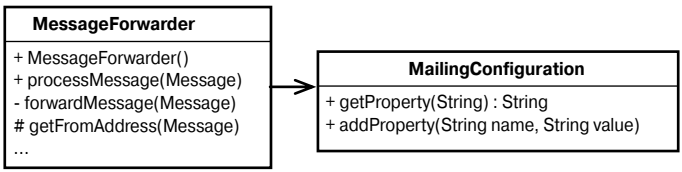


Рис. 8.3. Делегирование полномочий классу `MailingConfiguration`

Такая мера кажется действенной, но не является ли она чрезмерной? Ведь класс `MailingConfiguration`, по-видимому, выполняет те же функции, что и совокупность свойств.

Что, если переместить метод `getFromAddress` в класс `MailingConfiguration`? Класс `MailingConfiguration` мог бы тогда воспринимать сообщение и определять возвращаемый адрес отправителя. Если же конфигурация программы настроена на анонимность сообщений, то он мог бы возвращать адрес отправителя для анонимной рассылки сообщений. В противном случае он мог бы извлечь первый адрес из сообщения и возвратить этот адрес. Тогда структура кода выглядела бы так, как показано на рис. 8.4. Обратите внимание на то, что нам уже не требуется метод для получения и задания свойств. Теперь класс `MailingConfiguration` поддерживает функции более высокого уровня.

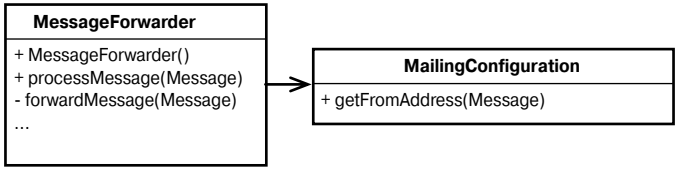


Рис. 8.4. Перенос поведения в класс `MailingConfiguration`

После этого мы могли бы приступить к вводу других методов в класс `MailingConfiguration`. Так, если нам потребовалось бы реализовать свойство обработки сообщений для внесписковых получателей, мы могли бы ввести метод под названием `buildRecipientList` (составить список получателей) в класс `MailingConfiguration` и предоставить классу `MessageForwarder` возможность использовать его так, как показано на рис. 8.5.

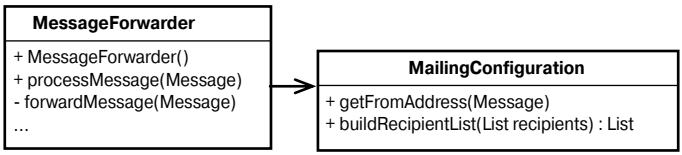


Рис. 8.5. Перенос дополнительного поведения в класс `MailingConfiguration`

При таких изменениях имя рассматриваемого здесь класса не вполне соответствует его функциям. Конфигурация обычно носит довольно пассивный характер, а этот класс активно создает и видоизменяет данные для объектов класса `MessageForwarder` по их запросу. Для такого класса более подходящим будет имя `MailingList` (список рассылки), если в системе отсутствует класс с аналогичным именем. Объекты класса `MessageForwarder` обращаются к спискам рассылки для определения адресов отправителей и составления списков получателей. Можно сказать, что ответственность за определение характера изменений в сообщениях лежит на списке рассылки. На рис. 8.6 показана структура кода после переименования упомянутого класса.

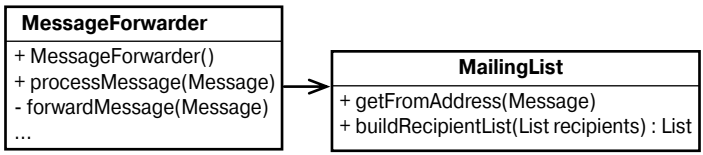


Рис. 8.6. Переименование класса `MailingConfiguration` *в класс* `MailingList`

Среди многих видов эффективной реорганизации кода самым эффективным считается переименование класса (Rename Class). Такая реорганизация кода изменяет точку зрения программистов на код и позволяет им обнаружить в нем те возможности, о которых они прежде и не подозревали.

Программирование по разности — полезный способ, позволяющий быстро вносить изменения в код и использовать тесты для перехода к более ясной конструкции. Но для того чтобы делать это правильно, нам придется принять во внимание два скрытых препятствия. Первым из них является нарушение *принципа подстановки Лискова* (LSP).

Принцип подстановки Лискова

Использование наследования может привести к появлению некоторых едва заметных ошибок. Рассмотрим следующий код.

```
public class Rectangle
{
    ...
    public Rectangle(int x, int y, int width, int height) { … }
    public void setWidth(int width) { ... }
    public void setHeight(int height) { ... }
    public int getArea() { ... }
}
```

В этом коде имеется класс `Rectangle` (прямоугольник). Можем ли мы создать подкласс с именем `Square` (квадрат)?

```
public class Square extends Rectangle
{
    ...
    public Square(int x, int y, int width) { ... }
   ...
}
```

Подкласс `Square` наследует методы `setWidth` (задать ширину) и `setHeight` (задать высоту) от класса `Rectangle`. Какой же должна получиться площадь квадрата в результате выполнения следующего кода?

```
Rectangle r = new Square();
r.setWidth(3);
r.setHeight(4);
```

Если площадь равна 12, то можно ли считать такую фигуру квадратом? Мы могли бы переопределить методы `setWidth` и `setHeight`, чтобы сохранить в подклассе `Square` форму квадрата, или же видоизменить переменные ширины и высоты в этих методах, но это привело бы к не совсем логичным результатам. Так, если задать высоту 4 и ширину 3, то вместо ожидаемой площади 12 будет получена площадь 16 прямоугольника.

Это классический пример нарушения принципа подстановки Лискова. Объекты подклассов должны быть подставлены в коде вместо объектов их суперклассов. В противном случае в коде возникают скрытые ошибки.

Принцип подстановки Лискова подразумевает, что клиенты отдельного класса должны иметь возможность использовать объекты его подкласса, даже не зная о том, что они на самом деле являются объектами подкласса. Каких-то действенных способов полностью исключить нарушение принципа подстановки Лискова не существует. Соответствие класса данному принципу зависит от клиентов этого класса и их ожиданий. Но для соблюдения этого принципа можно руководствоваться следующими эмпирическими правилами.

1. Избегайте, по возможности, переопределения конкретных методов.
2. Если вы все же переопределяете конкретные методы, то проверьте, сможете ли вы вызвать переопределяемый метод в переопределяющем методе.

Но в приведенном выше примере с классом `MessageForwarder` мы не делали ничего подобного. В действительности, мы поступили совсем наоборот: переопределили конкретный метод в подклассе (`AnonymousMessageForwarder`). И что же в этом особенного?

Дело в том, что когда мы переопределяем конкретные методы, как, например, метод `getFromAddress` класса `MessageForwarder` в классе `AnonymousMessageForwarder`, то можем изменить поведение некоторого кода, использующего объекты класса `MessageForwarder`. Если же ссылки на класс `MessageForwarder` размещены по всему приложению и одна из них настроена на обращение к классу `AnonymousMessageForwarder`, то пользователи такой ссылки могут посчитать, что они обращаются к классу `MessageForwarder`, который извлекает адрес получателя из обрабатываемого им сообщения и использует его для обработки других сообщений. Имеет ли для пользователей данного класса значение, какой именно адрес отправителя он использует: упомянутый выше или же другой специальный адрес? Это зависит от конкретного применения. Но в целом код становится более запутанным, когда мы слишком часто переопределяем конкретные методы. Кто-нибудь может заметить в коде ссылку на класс `MessageForwarder`, проанализировать этот класс и посчитать, что код в этом классе служит для выполнения метода `getFromAddress`, не имея даже представления о том, что ссылка на самом деле указывает на класс `AnonymousMessageForwarder` и что в данном случае используется метод `getFromAddress` именно этого класса. Если бы мы действительно хотели сохранить наследование, то могли бы сделать абстрактными класс `MessageForwarder` и его метод `getFromAddress`, предоставив подклассам возможность подставлять конкретные тела своих методов. На рис. 8.7 показано, как должна выглядеть структура кода после таких изменений.

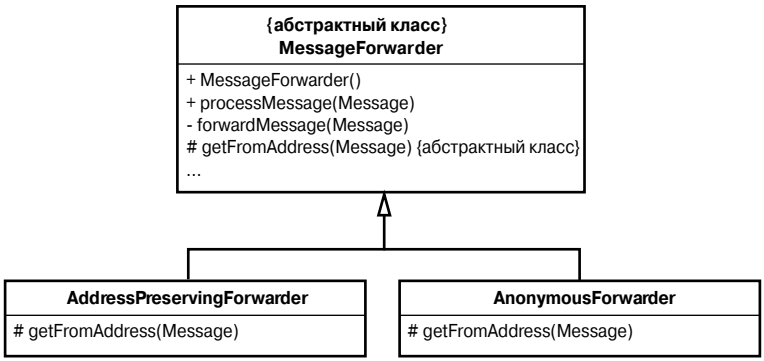


Рис. 8.7. Нормализованная иерархия

Такую иерархию классов можно назвать *нормализованной*. В нормализованной иерархии ни у одного из классов нет более одной реализации метода. Иными словами, ни у одного из классов нет метода, переопределяющего конкретный метод, наследуемый из суперкласса. В этом случае, анализируя функции абстрактного класса, можно обнаружить в нем абстрактные методы, каждый из которых реализуется в одном из подклассов данного класса. В нормализованной иерархии вам не нужно беспокоиться о переопределении в подклассах поведения, наследуемого ими от их суперклассов.

Следует ли делать это постоянно? Несколько переопределений конкретных методов никогда не помешают, если они не нарушают принцип подстановки Лискова. Но всякий

раз рекомендуется подумать о том, насколько классы отдаляются от нормализованной формы, и иногда обращаться к ней перед тем, как разделять ответственность.

Программирование по разности позволяет нам быстро вносить изменения в систему. При этом мы можем точно определить новое поведение с помощью написанных нами тестов и перейти, если потребуется, к более подходящим структурам. Тесты позволяют сделать такой переход довольно быстро.

Резюме

Способы, описанные в этой главе, можно использовать для ввода новых свойств в любой код, подлежащий тестированию. На тему разработки посредством тестирования за последние годы написано немало литературы. В частности, рекомендуются следующие книги: *Экстремальное программирование: разработка через тестирование* Кента Бека, издательство "Питер", 2003, а также *Test-Driven Development: A Practical Guide* (Разработка посредством тестирования. Практическое руководство) Дэйва Эстела (Dave Astel), Prentice Hall Professional Technical Reference, 2003.

Глава 9

Класс нельзя ввести в средства тестирования

Это тяжелый случай. Данная книга оказалась бы гораздо более краткой, если бы получать экземпляр класса в средствах тестирования было бы всегда просто. К сожалению, сделать это зачастую оказывается очень трудно.

Ниже перечислены четыре самых типичных затруднений, которые возникают в данном случае.

1. Объекты класса не могут быть созданы просто.
2. Средства тестирования нелегко компонуются вместе с находящимся в них классом.
3. Конструктор класса, который требуется использовать, дает скверные побочные эффекты.
4. В конструкторе класса происходит нечто очень важное и требующее распознавания.

Все эти затруднения будут рассмотрены в этой главе на целом ряде примеров, представленных на разных языках программирования. Разрешить каждое из этих затруднений нельзя каким-то одним способом. Однако изучение всех этих примеров поможет вам поближе ознакомиться с целым арсеналом способов разрыва зависимостей и научиться применять их попеременно в зависимости от конкретной ситуации.

Пример раздражающего параметра

Когда мне требуется внести изменения в унаследованный код, я обычно приступаю к делу с изрядной долей оптимизма. Сам не знаю, почему. Я изо всех сил стараюсь быть реалистом, но полностью избавиться от оптимизма все же не могу. При первом взгляде на унаследованный код, я говорю себе (или своему коллеге): “Похоже, что никаких осложнений не предвидится. Мы просто создадим объект `flumoux` класса `Floogle`, и дело с концом”. На словах все оказывается просто, но когда дело доходит до анализа кода класса `Floogle` (или как там он еще называется), то сразу же становится ясно, что придется добавить метод в одном месте и изменить метод в другом месте, ну и, конечно, ввести данный класс в средства тестирования. И в этот момент у меня начинают возникать сомнения: “Похоже, что простейший конструктор этого класса воспринимает три параметра. Но, — говорю я с оптимизмом, — построить его, возможно, будет не так уж и трудно”.

Итак, рассмотрим небольшой пример и посмотрим, оправдан ли мой оптимизм или же его следует просто приписать действию защитного механизма.

Допустим, что в коде системы выписывания счетов имеется непроверенный класс Java под названием `CreditValidator` (средство проверки кредитоспособности).

```
public class CreditValidator
{
    public CreditValidator(RGHConnection connection,
```

```
                                CreditMaster master,
                                String validatorID) {
            ...
    }

    Certificate validateCustomer(Customer customer)
            throws InvalidCredit {
    ...
    }

...
}
```

Одним из видов ответственности этого класса является уведомление о наличии действующего кредита у покупателей. Если у покупателей имеется такой кредит, то мы возвращаемся к кредитному документу, который сообщает нам сумму их кредита. В противном случае данный класс генерирует исключение.

Наша задача в данном примере — добавить новый метод в рассматриваемый здесь класс. Этот метод будет называться `getValidationPercent`. Его основное назначение — сообщить нам количество (в процентах) успешных вызовов метода `validateCustomer` (проверить покупателя), сделанных нами в течение срока действия объекта `validator` класса `CreditValidator`.

С чего же нам начать?

Когда нам требуется создать объект в средствах тестирования, для этого зачастую оказывается достаточно просто попробовать сделать это. После этого мы можем тщательно проанализировать, почему это сделать легко, или же наоборот, трудно, а до тех пор нам ничто не мешает создать тестовый класс JUnit, ввести в него приведенный ниже код и скомпилировать его.

```
public void testCreate() {
    CreditValidator validator = new CreditValidator();
}
```

Самый лучший способ выяснить, насколько хлопотным окажется получение экземпляра класса в средствах тестирования, состоит в том, чтобы просто попытаться это сделать. Для этого напишите контрольный пример и попытайтесь создать в нем объект. Компилятор сообщит вам, что именно вам потребуется, чтобы получить работоспособный код.

Данный тест является тестом на построение. Такие тесты выглядят не совсем обычно. Когда я пишу тест на построение, то обычно не ввожу в его код утверждение, а просто пытаюсь создать объект. В дальнейшем, когда я, наконец-то, могу построить объект в средствах тестирования, то, как правило, я избавляюсь от такого теста или же переименовываю его, чтобы проверить с его помощью что-нибудь более существенное.

Но вернемся к нашему примеру. Мы пока еще не ввели аргументы в конструктор, и поэтому компилятор сообщает нам, что у класса `CreditValidator` отсутствует стандартный конструктор. Просматривая код, мы обнаруживаем, что в качестве аргументов нам требуется указать объекты классов `RGHConnection` (жесткое соединение), `CreditMaster` (владелец кредита) и пароль. У каждого из этих классов имеется только один конструктор. Они выглядят следующим образом.

```
public class RGHConnection
{
    public RGHConnection(int port, String Name, string passwd)
            throws IOException {
        ...
    }
}

public class CreditMaster
{
    public CreditMaster(String filename, boolean isLocal) {
    ...
    }
}
```

При построении класса `RGHConnection` устанавливается соединение с сервером. С помощью этого соединения мы получаем из сервера все отчеты, необходимые для проверки кредитоспособности покупателя.

Другой класс, `CreditMaster`, предоставляет нам информацию об определенных правилах поведения, которой мы пользуемся, принимая решение относительно кредитования. При построении класса `CreditMaster` подобная информация загружается из файла и хранится в оперативной памяти.

Похоже, что ввести этот класс в средства тестирования будет очень легко, не так ли? Не будем спешить с выводами. Мы действительно можем написать тест, но достаточно ли этого?

```
public void testCreate() throws Exception {
    RGHConnection connection = new RGHConnection(DEFAULT_PORT,
"admin", "rii8ii9s");
    CreditMaster master = new CreditMaster("crm2.mas", true);
    CreditValidator validator = new CreditValidator(
connection, master, "a");
}
```

Установление соединений с сервером с помощью объектов класса `RGHConnection` в данном тесте оказывается далеко не самой лучшей идеей, поскольку для этого требуется немало времени, да и сервер не всегда работает. С другой стороны, класс `CreditMaster` не вызывает подобных затруднений. Когда мы создаем объект класса `CreditMaster`, он загружает свой файл довольно быстро. Кроме того, файл находится в состоянии "только для чтения", и поэтому мы можем не беспокоиться о том, что тесты испортят его.

По-настоящему серьезным препятствием на пути к созданию объекта `validator` класса `CreditValidator` служит класс `RGHConnection`. Это и есть тот самый *раздражающий параметр*. Если мы сумеем создать определенного рода фиктивный объект типа `RGHConnection`, заменяющий собой реальный объект, с которым взаимодействует класс `CreditValidator`, то сможем обойти все препятствия, связанные с установлением соединения. С этой целью рассмотрим методы, предоставляемые классом `RGHConnection` (рис. 9.1).

Похоже, что в классе `RGHConnection` имеется целый ряд методов, непосредственно связанных с механизмом установления соединения: `connect` (соединить), `disconnect` (разъединить) и `retry` (повторить), а также методы более прикладного характера: `RFDIReportFor` и `ACTIOReportFor`. При написании нового метода в классе `CreditValidator` нам потребуется вызов метода `RFDIReportFor`, чтобы получить всю

необходимую нам информацию. Как правило, вся эта информация поступает из сервера, но поскольку мы должны всячески избегать установления соединения с ним, то вынуждены сами предоставить вместо этого некий фиктивный объект.

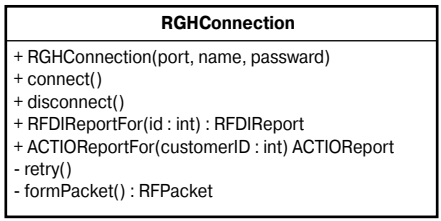


Рис. 9.1. Класс `RGHConnection`

В данном случае для создания фиктивного объекта лучше всего воспользоваться *извлечением интерфейса*. Если у вас имеется инструментальное средство тестирования, поддерживающее реорганизацию кода, то оно, скорее всего, поддерживает также извлечение интерфейса. В противном случае вы должны просто запомнить, что это нетрудно сделать и вручную.

После извлечения интерфейса мы получим в конечном итоге структуру, аналогичную приведенной на рис. 9.2.

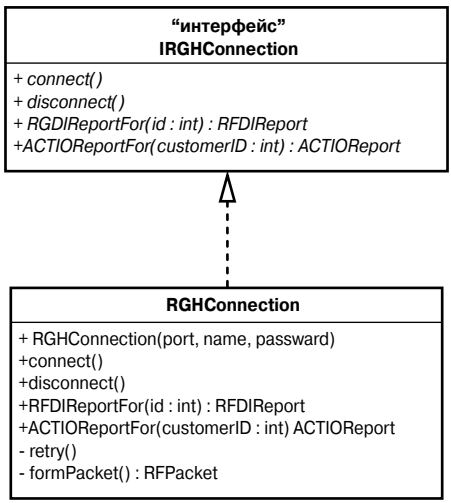


Рис. 9.2. Структура класса `RGHConnection` *после извлечения интерфейса*

Теперь мы можем приступить к написанию тестов, создав небольшой фиктивный класс, предоставляющий нам все необходимые отчеты.

```
public class FakeConnection implements IRGHConnection
{
    public RFDIReport report;

    public void connect() {}
    public void disconnect() {}
```

```
    public RFDIReport RFDIReportFor(int id) { return report; }
    public ACTIOReport ACTIOReportFor(int customerID) { return null; }
}
```

Имея в своем распоряжение такой класс, мы можем написать тест, подобный приведенному ниже.

```
void testNoSuccess() throws Exception {
    CreditMaster master = new CreditMaster("crm2.mas", true);
    IRGHConnection connection = new FakeConnection();
    CreditValidator validator = new CreditValidator(
connection, master, "a");

    connection.report = new RFDIReport(...);

    Certificate result = validator.validateCustomer(new Customer(...));

    assertEquals(Certificate.VALID, result.getStatus());
}
```

Класс `FakeConnection` (фиктивное соединение) выглядит несколько странно. Как часто вам приходится писать методы, вообще не имеющие тела или же просто возвращающие пустое значение после вызова? Более того, в данном классе есть общедоступная переменная, которую может установить кто угодно и где угодно. На первый взгляд, такой класс нарушает все правила, но на самом деле это не совсем так. Для классов, которые мы используем, чтобы сделать возможным тестирование, правила несколько отличаются. Ведь код в классе `FakeConnection` не является выходным. Такой код никогда не будет выполняться в полноценно работающем приложении — он нужен только для средств тестирования.

А теперь, когда мы можем создать объект `validator` класса `CreditValidator`, напишем метод `getValidationPercent`. Ниже приведен тест для него.

```
void testAllPassed100Percent() throws Exception {
    CreditMaster master = new CreditMaster("crm2.mas", true);
    IRGHConnection connection = new FakeConnection("admin", "rii8ii9s");
    CreditValidator validator = new CreditValidator(
                          connection, master, "a");

    connection.report = new RFDIReport(...);
    Certificate result = validator.validateCustomer(new Customer(...));
    assertEquals(100.0, validator.getValidationPercent(), THRESHOLD);
}
```

Тестовый код по сравнению с выходным кодом

Тестовый код не обязательно должен соответствовать тем же стандартам, что и выходной код. Как правило, я не обращаю особого внимания на нарушение инкапсуляции, делая переменные общедоступными, если это упрощает написание тестов. Тем не менее тестовый код должен быть ясным. Он должен быть простым для понимания и изменения.

Обратите внимание на методы `testNoSuccess` (тест не прошел) и `testAllPassed-100Percent` (тест на полную кредитоспособность прошел полностью) в рассматриваемом здесь примере. Содержат ли они какой-нибудь дублированный код? Да, содержат. Первые три строки кода дублируются. Они должны быть извлечены и размещены в общем месте, т.е. в методе `setUp()` упоминаемого здесь тестового класса.

Данный тест проверяет, подтверждается ли почти полностью кредитоспособность покупателя при получении одного действующего кредитного документа.

Этот тест вполне справляется со своей задачей, но при написании кода для метода `getValidationPercent` обращает на себя внимание одно любопытное обстоятельство. Оказывается, что в методе `getValidationPercent` вообще не предполагается использование класса `CreditMaster`. Зачем же тогда создавать объект этого класса и передавать его классу `CreditValidator`? Может быть это нам и не нужно. Ведь мы могли бы создать в нашем тесте объект класса `CreditValidator` следующим образом:

```
CreditValidator validator = new CreditValidator(connection, null, "a");
```

По реакции разработчиков на подобную строку кода можно судить о качестве системы, которую они проектируют. Так, если они отреагируют на эту строку кода следующим образом: "Передача пустого значения в конструкторе — как раз то, что нам нужно. Мы делаем это постоянно в своей системе", то, скорее всего, их система будет спроектирована далеко не самым лучшим образом. Ведь им придется проверять пустое значение везде, где только оно может появиться, используя немало условного кода, чтобы прояснить ситуацию и решить, что делать дальше. А если они отреагируют так: "Как, передавать пустое значение по всей системе? Неужели нельзя придумать ничего получше?", то этой последней категории разработчиков (или, по крайней мере, тем, кто еще читает данную книгу и не захлопнул ее, забросив в сердцах на полку) я просто скажу следующее: помните, что мы делаем это только в тестах. Самое худшее, что может произойти, — это попытка использовать данную переменную в некотором коде. В данном случае во время выполнения кода Java будет сгенерировано исключение. А поскольку средства тестирования перехватывают все исключения, генерируемые в тестах, то мы сможем очень быстро выяснить, используется ли вообще данный параметр.

Передача пустого значения

Если при написании тестов объекту требуется параметр, который трудно построить как объект, то вместо этого следует рассмотреть возможность передачи пустого значения. Если этот параметр используется по ходу выполнения теста, то код сгенерирует исключение, а средства тестирования перехватят такое исключение. Если же необходимо поведение, которое действительно требует объекта, то такой объект можно построить и передать в качестве параметра в данной точке программы.

Передача пустого значения является очень удобным способом в некоторых языках программирования. Она отлично подходит для Java, C# и практически любого языка, в котором генерируется исключение, если во время выполнения используются ссылки на пустое значение. Но делать это в C и C++ не рекомендуется, если не известно, что во время выполнения будут выявлены ошибки указания на пустое значение. Иначе тесты будут в лучшем случае аварийно завершаться самым непредсказуемым образом, а в худшем случае — тесты просто окажутся тайно и безнадежно неверными. Во время выполнения они разрушат данные в памяти, и вы даже не узнаете об этом.

Когда я программирую на Java, зачастую начинаю работу, прежде всего, с теста, аналогичного приведенному ниже, заполняя параметры по мере надобности.

```
public void testCreate() {
    CreditValidator validator = new CreditValidator(null, null, "a");
}
```

В связи этим очень важно не забывать о следующем: не передавайте пустое значение в выходном коде, если только у вас нет иного выхода. Я знаю, что в некоторых библиотеках

такая передача предполагается, но при написании нового кода имеются альтернативы и получше. Если же у вас возникает искушение использовать пустое значение в выходном коде, то найдите места, где возвращаются и передаются пустые значения, и рассмотрите возможность для иного способа их передачи, например *шаблона пустого объекта*.

Шаблон пустого объекта

Шаблон пустого объекта — это способ избежать использования пустых значений в программе. Так, если имеется метод, возвращающий данные работника по его идентификационному номеру, то что мы должны возвратить, если работника с таким идентификационным номером не существует?

```
for(Iterator it = idList.iterator(); it.hasNext(); ) {
    EmployeeID id = (EmployeeID)it.next();
    Employee e = finder.getEmployeeForID(id);
    e.pay();
}
```

В этом случае у нас имеются два варианта. Во-первых, мы могли бы организовать генерирование исключения, чтобы вообще ничего не возвращать, но тогда клиентам пришлось бы явно обрабатывать ошибку. И во-вторых, мы могли бы возвратить пустое значение, но тогда клиентам пришлось бы явно проверять пустое значение.

Но имеется и третий вариант. Если в приведенном выше коде никак не учитывается отсутствующий для оплаты работник, то, может быть, имеет смысл ввести класс `NullEmployee` (пустой работник). У экземпляра объекта класса `NullEmployee` нет ни Ф.И.О., ни адреса работника, и когда ему предписывается оплатить труд такого работника, то он ничего не делает.

Пустые объекты могут быть полезными в подобном контексте, поскольку они избавляют клиентов от явной проверки ошибок. Но несмотря на все преимущества пустых объектов, с ними нужно обходиться очень аккуратно. Так, ниже приведен пример очень неудачного подсчета оплаченных работников.

```
int employeesPaid = 0;
for(Iterator it = idList.iterator(); it.hasNext(); ) {
    EmployeeID id = (EmployeeID)it.next();
    Employee e = finder.getEmployeeForID(id);
    e.pay();
    mployeesPaid++; // программная ошибка!
}
```

Если будут возвращены пустые данные о любом из работников, то подсчет оплаченных работников окажется неверным.

Пустые объекты полезны, в частности, тогда, когда клиента не интересует, является ли та или иная операция успешной. И зачастую мы можем ловко подвести наш проект именно под такой случай.

Передача пустого значения и *извлечение интерфейса* являются двумя подходами к разрешению проблемы раздражающего параметра. Но иногда полезным оказывается еще один подход. Если зависимость от параметра, вызывающего раздражение, не закодирована жестко в его конструктор, то мы можем воспользоваться *подклассификацией и переопределением метода*, чтобы избавиться от такой зависимости. В данном случае это вполне возможно. Так, если метод `connect` используется в конструкторе класса `RGHConnection`

для установления соединения, то мы могли бы разорвать зависимость, переопределив метод `connect` в тестовом подклассе. Подклассификация и переопределение метода оказывается весьма полезным способом разрыва зависимостей, но прежде чем воспользоваться им, мы должны убедиться в том, что не изменяем поведение, которое мы проверяем.

Пример скрытой зависимости

Некоторые классы выглядят обманчиво простыми. Мы просматриваем их, находим конструктор, который нам требуется использовать, но как только мы попытаемся вызвать его, то сразу же наталкиваемся на препятствие. Одним из самых распространенных препятствий в подобных случаях является *скрытая зависимость*, которая означает, что в конструкторе используется некоторый ресурс, к которому мы не можем получить доступ из средств тестирования. В подобной ситуации мы оказываемся в следующем примере неудачно построенного класса C++, управляющего списком рассылки.

```
class mailing_list_dispatcher
{
public:
                mailing_list_dispatcher ();
    virtual     ~mailing_list_dispatcher;

    void        send_message(const std::string& message);
    void        add_recipient(const mail_txm_id id,
                              const mail_address& address);
    ...

    private:
        mail_service     *service;
        int              status;
};
```

Это лишь часть конструктора класса, распределяющая объект `mail_service` (почтовая служба), используя оператор `new` в списке инициализации конструктора. Такой стиль программирования явно неудачный. Ведь в конструкторе выполняется много детальных операций с объектом `mail_service` и, к тому же, используется магическое число `12`, которое вообще не понятно что означает.

```
mailing_list_dispatcher::mailing_list_dispatcher()
: service(new mail_service), status(MAIL_OKAY)
{
    const int client_type = 12;
    service->connect();
    if (service->get_status() == MS_AVAILABLE) {
        service->register(this, client_type, MARK_MESSAGES_OFF);
        service->set_param(client_type, ML_NOBOUNCE | ML_REPEATOFF);
    }
    else
        status = MAIL_OFFLINE;
    ...
}
```

Мы можем создать экземпляр данного класса в тесте, но это мало что нам даст. Прежде всего, нам нужно прикомпоновать почтовые библиотеки и настроить почтовую систему на регистрацию сообщений. А если мы воспользуемся функцией `send_message` в своих тестах, то фактически отправим почтовое сообщение на деревню дедушке. Протестировать такие функции в автоматическом режиме очень трудно, если не установить специальный почтовый ящик и периодически связываться с ним, ожидая поступления сообщений. Это было бы уместно для тестирования всей системы в целом, но если нам нужно лишь ввести некоторые тестируемые функции в класс, то стоит ли городить такой огород. Как же нам создать тест для простого объекта, чтобы ввести ряд новых функций?

Основная трудность в данном случае заключается в том, что зависимость от объекта `mail_service` является скрытой в конструкторе `mailing_list_dispatcher` (диспетчер списка рассылки). Если найти какой-нибудь способ заменить объект `mail_service` фиктивным объектом, то, распознав такой объект, мы могли бы получить ответную реакцию при изменении класса.

Для этой цели подходит, в частности, способ, называемый *параметризацией конструктора*. С его помощью мы овеществляем зависимость, имеющуюся в конструкторе, передавая ее в этом конструкторе.

Вот как выглядит код конструктора после его параметризации.

```
mailing_list_dispatcher::mailing_list_dispatcher(mail_service *service)
: status(MAIL_OKAY)
{
    const int client_type = 12;
    service->connect();
    if (service->get_status() == MS_AVAILABLE) {
        service->register(this, client_type, MARK_MESSAGES_OFF);
        service->set_param(client_type, ML_NOBOUNCE | ML_REPEATOFF);
    }
    else
        status = MAIL_OFFLINE;
    ...
}
```

Единственное отличие состоит в том, что объект `mail_service` создается за пределами класса и передается ему. На первый взгляд, такое усовершенствование кажется незначительным, но на самом деле оно дает нам невероятные возможности для достижения поставленной цели. В частности, мы можем воспользоваться *извлечением интерфейса*, чтобы создать интерфейс для объекта `mail_service`. В качестве одного из средств реализации такого интерфейса может служить выходной класс, который действительно отправляет почту. С другой стороны, им может стать фиктивный класс, распознающий результаты наших действий во время его тестирования и позволяющий нам убедиться в том, что они произошли на самом деле.

Параметризация конструктора — очень удобный способ овеществления зависимостей, имеющихся в конструкторе, хотя о нем вспоминают очень редко. Одним из камней преткновения для разработчиков служит следующее обстоятельство: они нередко считают, что для передачи нового параметра им придется изменить все клиенты данного класса, но на самом деле это не так. В данной ситуации мы можем поступить следующим образом. Сначала мы извлекаем тело конструктора в новый метод, назвав его `initialize`. В отличие от других видов извлечения метода, такое извлечение представляет собой довольно

безопасную попытку без тестирования, поскольку, выполняя его, мы можем *сохранить сигнатуры*.

```
void mailing_list_dispatcher::initialize(mail_service *service)
{
    status = MAIL_OKAY;
    const int client_type = 12;
    service.connect();
    if (service->get_status() == MS_AVAILABLE) {
        service->register(this, client_type, MARK_MESSAGES_OFF);
        service->set_param(client_type, ML_NOBOUNCE | ML_REPEATOFF);
    }
    else
    status = MAIL_OFFLINE;
    ...
}
mailing_list_dispatcher::mailing_list_dispatcher(mail_service *service)
{
    initialize(service);
}
```

Далее мы можем предоставить конструктор с исходной сигнатурой. Если в тестах этот конструктор может вызываться параметризированным объектом `mail_service`, то со стороны клиентов — с исходной сигнатурой. Им совсем не обязательно знать о том, что в конструкторе произошли какие-то изменения.

```
mailing_list_dispatcher::mailing_list_dispatcher()
{
    initialize(new mail_service);
}
```

Подобную реорганизацию кода еще проще осуществить в таких языках, как C# и Java, поскольку мы можем вызывать в них одни конструкторы из других конструкторов.

Так, если бы мы делали нечто подобное на языке C#, то получившийся в итоге код выглядел бы следующим образом:

```
public class MailingListDispatcher
{
    public MailingListDispatcher()
    : this(new MailService())
    {}

    public MailingListDispatcher(MailService service) {
        ...
    }
}
```

Зависимости, скрытые в конструкторах, можно разрешить самыми разными способами. Нередко для этой цели мы пользуемся *извлечением и переопределением получателя*, *извлечением и переопределением фабричного метода* и *заменой переменной экземпляра*, но я лично предпочитаю как можно чаще пользоваться *параметризацией конструктора*. Если объект создается в конструкторе и сам не имеет никаких зависимостей построения, то применить параметризацию конструктора не составляет большого труда.

Пример пятна построения объекта

Параметризация конструктора считается одним из самых простых способов разрыва скрытых зависимостей в конструкторе, и поэтому в своей практике программирования я часто обращаюсь, прежде всего, к ней. Если же внутри конструктора строится большое количество объектов или же осуществляется доступ к большому количеству глобальных переменных, то в конечном итоге получается довольно крупный список параметров. В худшем случае конструктор создаст сначала несколько объектов, а затем воспользуется ими для создания других объектов, как в приведенном ниже фрагменте кода.

```
class WatercolorPane
{
public:
    WatercolorPane(Form *border, WashBrush *brush, Pattern *backdrop)
    {
        ...
        anteriorPanel = new Panel(border);
        anteriorPanel->setBorderColor(brush->getForeColor());
        backgroundPanel = new Panel(border, backdrop);

        cursor = new FocusWidget(brush, backgroundPanel);
        ...
    }
    ...
}
```

Если нам нужно распознать курсор, то мы вряд ли сможем это сделать. Ведь объект курсора вложен в пятно построения объекта. Мы могли бы попытаться вынести сначала весь код, используемый для создания курсора, за пределы класса, а затем клиент мог бы создать курсор и передать его в качестве аргумента. Но это не очень надежно, если у нас нет подходящих тестов на местах, и к тому же, может доставить немало хлопот клиентам данного класса.

Если у нас имеется инструментальное средство реорганизации кода, безопасно извлекающее методы, то мы можем воспользоваться *извлечением и переопределением фабричного метода* из кода конструктора, хотя такой способ оказывается работоспособным не во всех языках программирования. Так, в Java и C# нам удастся это сделать, а в C++ не допускаются вызовы виртуальных функций в конструкторах для разрешения виртуальных функций, определяемых в производных классах. Да и вообще, это не самая лучшая идея. Ведь в функциях из производных классов нередко допускается возможность использовать переменные из их базового класса. А до тех пор, пока конструктор базового класса не будет полностью завершен, существует вероятность того, что переопределенная функция, которую он вызывает, может получить доступ к неинициализированной переменной.

Другая возможность состоит в *замене переменной экземпляра*. В этом случае мы вводим в класс метод установки, или так называемый установщик, который позволяет нам поменять объект на другой экземпляр после его построения.

```
class WatercolorPane
{
public:
    WatercolorPane(Form *border, WashBrush *brush, Pattern *backdrop)
```

```
    {
        ...
        anteriorPanel = new Panel(border);
        anteriorPanel->setBorderColor(brush->getForeColor());
        backgroundPanel = new Panel(border, backdrop);

       cursor = new FocusWidget(brush, backgroundPanel);
       ...
    }
    void supersedeCursor(FocusWidget *newCursor)
    {
        delete cursor;
        cursor = newCursor;
    }
}
```

Такую реорганизацию кода в C++ следует выполнять очень осторожно. Ведь когда мы заменяем объект, то должны избавиться от старого объекта, а это зачастую означает, что нам придется прибегнуть к оператору удаления, чтобы вызвать его деструктор и разрушить данный объект в оперативной памяти. Делая это, мы должны ясно понимать, что именно деструктор выполняет и не разрушит ли он заодно все, что передается конструктору данного объекта. Если мы не примем меры предосторожности при очистке оперативной памяти, то можем внести ряд едва заметных программных ошибок.

Во многих языках программирования замена переменной экземпляра выполняется довольно просто. Ниже приведен результат, зарегистрированный в Java. Для того чтобы избавиться от объекта, на который ссылается переменная `cursor`, нам не нужно делать ничего особенного; в конечном итоге обо всем позаботится "сборщик мусора". Но мы должны быть очень внимательны, чтобы не пользоваться заменой переменной экземпляра в выходном коде. Если объекты, которые мы пытаемся заменить, управляют другими ресурсами, то мы можем вызвать серьезные осложнения в управлении ресурсами.

```
void supersedeCursor(FocusWidget newCursor) {
    cursor = newCursor;
}
```

А теперь, когда у нас имеется заменяющий метод, мы можем попытаться создать объект `FocusWidget` (реквизит окна для фокуса) вне класса и передать его объекту после построения последнего. Для распознавания нам потребуется *извлечение интерфейса* или же *извлечение средства реализации* из класса `FocusWidget` и создание фиктивного объекта, передаваемого этому классу. Безусловно, создать такой объект будет проще, чем объект `FocusWidget` в конструкторе.

```
TEST(renderBorder, WatercolorPane)
{
    ...
    TestingFocusWidget *widget = new TestingFocusWidget;
    WatercolorPane pane(form, border, backdrop);

    pane.supersedeCursor(widget);

    LONGS_EQUAL(0, pane.getComponentCount());
}
```

Я лично предпочитаю не пользоваться *заменой переменной экземпляра*, если у меня есть другой выход, поскольку в этом случае слишком велика вероятность появления осложнений в управлении ресурсами. Но иногда я все же пользуюсь этим способом в C++. Я, конечно, предпочел бы *извлечение и переопределение фабричного метода*, но этого не допускают конструкторы C++. Именно по этой причине я и пользуюсь заменой переменной экземпляра.

Пример раздражающей глобальной зависимости

Многие годы программисты жаловались на отсутствие достаточного количества многократно используемых компонентов на рынке, но постепенно ситуация меняется к лучшему. На рынке уже появилось немало коммерчески и свободно доступных структур, или так называемых каркасов, но, в целом, многие из них не находят широкого применения, поскольку их приходится дополнять собственным кодом. Такие структуры нередко обеспечивают жизненный цикл приложения, а нам приходится писать код, заполняющий в них пробелы. Подобная ситуация характерна для всех разновидностей структур — от ASP.NET до Java Struts. Даже среды тестирования xUnit ведут себя подобным образом: мы пишем тестовые классы, а xUnit вызывает их и отображает результаты тестирования.

Структуры решают многие проблемы и оказывают нам заметную поддержку, когда мы приступаем к работе над проектом, но это не тот вид многократного использования, на который мы на самом деле рассчитывали раньше при разработке программного обеспечения. Когда мы обнаруживаем некоторый класс или ряд классов, подходящих для нашего приложения, мы просто используем их в очередной раз по старинке. Конечно, было бы неплохо делать это регулярно, но, откровенно говоря, такого рода многократное использование кажется просто несерьезным, если нельзя даже извлечь произвольный класс из обыкновенного приложения и скомпилировать его отдельно в средствах тестирования, не приложив для этого немало труда (не сочтите это за ворчание старого программиста).

Существенно затруднить создание и использование классов в среде тестирования способны многие разновидности зависимости, но самые большие трудности возникают в связи с глобальными переменными. В простейших случаях мы можем воспользоваться *параметризацией конструктора*, *параметризацией метода*, а также *извлечением и переопределением вызова*, чтобы избавиться от такой зависимости, но иногда зависимость от глобальных переменных приобретает настолько обширный характер, что это зло оказывается проще побороть в корне. Ниже приведен пример подобной ситуации, где представлен один из основных классов из приложения на языке Java, регистрирующего разрешения на строительство в государственном органе.

```
public class Facility
{
    private Permit basePermit;

    public Facility(int facilityCode, String owner, PermitNotice notice)
            throws PermitViolation {

    Permit associatedPermit =
        PermitRepository.getInstance().findAssociatedPermit(notice);
    if (associatedPermit.isValid() && !notice.isValid()) {
        basePermit = associatedPermit;
    }
```

```
    else if (!notice.isValid()) {
        Permit permit = new Permit(notice);
        permit.validate();
        basePermit = permit;
    }
    else
        throw new PermitViolation(permit);
    }
    ...
}
```

Нам нужно создать объект класса `Facility` (строительный объект) в средствах тестирования, и поэтому мы начинаем с попытки построить этот объект в средствах тестирования.

```
public void testCreate() {
    PermitNotice notice = new PermitNotice(0, "a");
    Facility facility = new Facility(Facility.RESIDENCE, "b", notice);
}
```

Этот тест компилируется нормально, но когда мы приступаем к написанию дополнительных тестов, сразу наталкиваемся на препятствие. В конструкторе используется класс `PermitRepository` (архив разрешений), который требуется инициализировать конкретным рядом разрешений, чтобы правильно настроить наши тесты. Этим коварным и раздражающим препятствием является следующий оператор в конструкторе:

```
Permit associatedPermit =
    PermitRepository.getInstance().findAssociatedPermit(notice);
```

Такое препятствие мы могли бы обойти параметризацией конструктора, но в данном приложении этот случай не единичный. В нем насчитывается еще 10 классов приблизительно с такой же самой строкой кода. Она находится в конструкторах, регулярных и статических методах. Можно себе представить, сколько времени потребуется на устранение данного препятствия в базе кода.

Если вам приходилось изучать шаблоны проектирования, то вы, вероятно, распознаете в данной ситуации пример *шаблона проектирования одиночки*. Метод `getInstance` (получить экземпляр) класса `PermitRepository` является статическим методом, возвращающим единственный экземпляр класса `PermitRepository`, который может существовать в данном приложении. Поле, в котором хранится этот экземпляр, также является статическим и находится в классе `PermitRepository`.

В Java шаблон одиночки служит в качестве одного из механизмов для создания глобальных переменных. Как правило, пользоваться глобальными переменными не рекомендуется по двум причинам. В частности, они делают код непрозрачным. Анализируя код, мы должны понять, как и на что он действует. Так, если нам нужно понять, на что может воздействовать приведенный ниже код Java, то мы должны обратить внимание лишь на два места в этом коде.

```
Account example = new Account();
example.deposit(1);
int balance = example.getBalance();
```

Нам известно, что объект счета (`Account`) может оказывать воздействие на параметры, передаваемые конструктору `Account`, но в данном случае ничего не передается. Объекты `Account` могут также оказывать воздействие на объекты, передаваемые методам в качестве

параметров, но в данном случае не передается ничего из того, что могло быть измениться, кроме целого значения, возвращаемого методом `getBalance` (получить баланс) и присваиваемого целочисленной переменной `balance`. Именно на это значение и могут повлиять приведенные выше операторы.

Когда мы пользуемся глобальными переменными, данная ситуация меняется в корне. Анализируя использование такого класса, как `Account`, мы можем не иметь даже понятия, получает ли он доступ или же модифицирует переменные, объявленные в каком-нибудь другом месте программы. Нечего и говорить, насколько это затрудняет понимание программ.

Самое трудное для тестирования кода в подобной ситуации — выяснить, какие именно глобальные переменные используются классом, чтобы установить их в подходящее для тестирования состояние. И это нужно сделать перед каждым тестом, если он требует иной настройки. Такая работа, откровенно говоря, не из приятных. Мне приходилось выполнять ее в десятках систем, чтобы подготовить их к тестированию, и никогда она не вызывала у меня особого энтузиазма.

Но вернемся к нашему примеру. Класс `PermitRepository` представляет собой одиночку, поскольку подделать его особенно трудно. Основное назначение шаблона одиночки — сделать невозможным создание более чем одного экземпляра одиночки в приложении. Возможно, это и приемлемо для выходного кода, но не годится для тестирования, поскольку каждый тест должен быть полностью изолирован как отдельное мини-приложение от других тестов. Поэтому для выполнения кода, содержащего одиночки, в средствах тестирования нам придется ослабить свойство одиночки. И вот как мы это сделаем.

Прежде всего, следует ввести статический метод в класс одиночки. Этот метод позволит нам заменить статический экземпляр в одиночке. Назовем этот метод `setTestingInstance` (задать тестовый экземпляр).

```
public class PermitRepository
{
    private static PermitRepository instance = null;

    private PermitRepository() {}

    public static void setTestingInstance(PermitRepository newInstance)
    {
        instance = newInstance;
    }

    public static PermitRepository getInstance()
    {
        if (instance == null) {
            instance = new PermitRepository();
        }
        return instance;
    }

    public Permit findAssociatedPermit(PermitNotice notice) {
        ...
    }
    ...
}
```

А теперь, когда у нас имеется метод установки, мы можем создать тестовый экземпляр класса `PermitRepository` и задать его. Для этой цели напишем в нашей тестовой среде следующий код:

```
public void setUp() {
    PermitRepository repository = new PermitRepository();
    ...
    // добавить здесь разрешения в архив
    ...
    PermitRepository.setTestingInstance(repository);
}
```

Ввод статического установщика — это не единственный способ разрешения рассматриваемой здесь ситуации. Имеется и другой способ. Мы можем ввести в класс одиночки следующий метод `resetForTesting()` (сбросить для тестирования):

```
public class PermitRepository
{
    ...
    public void resetForTesting() {
        instance = null;
    }
    ...
}
```

Вызывая этот метод в нашей тестовой среде, мы можем создавать новые экземпляры одиночек для каждого теста. Одиночка будет переинициализироваться для каждого теста. Такой алгоритм вполне подходит в том случае, если общедоступные методы в одиночке позволяют устанавливать его состояние каждый раз, когда это требуется для тестирования. Если же у одиночки нет таких общедоступных методов или же он пользуется некоторыми внешними ресурсами, оказывающими воздействие на его состояние, в таком случае лучше выбрать ввод статического установщика. Для установки одиночки в правильное состояние можно выполнить его подклассификацию, переопределить методы для разрыва зависимостей и ввести общедоступные методы в подкласс.

Будет ли такой прием вполне работоспособным? Не совсем. Когда программисты применяют *шаблон проектирования одиночки*, они нередко делают частным конструктор класса одиночки, и для этого у них имеются веские основания. Ведь это самый очевидный способ гарантировать, что никто за пределами данного класса не сможет создать еще один экземпляр одиночки.

В данный момент у нас возникло противоречие между двумя целями проектирования. С одной стороны, нам нужно гарантировать наличие в системе только одного экземпляра класса `PermitRepository`, а с другой — нам требуется тестировать классы в системе независимо один от другого. Можем ли мы достичь обеих целей?

Сделаем небольшое отступление. Зачем вообще в системе нужен единственный экземпляр класса? Ответ на этот вопрос зависит от конкретной системы, поэтому ниже приведены наиболее общие варианты такого ответа.

1. **Мы моделируем реальный мир, в котором моделируемый объект присутствует только в одном экземпляре.** К этому типу относятся некоторые системы управления аппаратурой. Программисты создают класс для каждой монтажной платы, которой требуется управлять. Они считают, что если имеется только одна плата

каждого типа, то ей должен соответствовать одиночка. Это же относится и к базам данных. В государственном органе имеется лишь один архив разрешений на строительство, поэтому объектом, обеспечивающим доступ к архиву, должен быть одиночка.

2. **Если создаются два экземпляра одного и того же объекта, то возможны серьезные осложнения.** Подобная ситуация опять же характерна для систем управления аппаратурой. Представьте себе, что случайно созданы два контроллера, управляющие стержнями атомного реактора, и две разные части программы управляют одними и теми же стержнями, независимо одна от другой.
3. **Если создаются два экземпляра объекта, то они используют слишком много ресурсов.** Такое случается часто. В качестве ресурсов служат как физические параметры, например, дисковое пространство или потребление оперативной памяти, так и абстрактные параметры, в том числе количество лицензий на программное обеспечение.

Упомянутые выше причины являются основными для получения единственного экземпляра объекта, но они еще не объясняют, зачем программисты пользуются одиночками. Программисты нередко создают одиночки, потому что им требуются глобальные переменные. Они считают, что иначе им было бы неудобно передавать переменную в те места программы, где она требуется.

Если нам требуется одиночка по последней причине, то у нас нет никаких оснований хранить свойство одиночки. Мы можем сделать конструктор защищенным, общедоступным или же с областью действия в пределах пакета, имея при этом приличную, тестируемую систему. В других случаях по-прежнему целесообразно исследовать иные альтернативы. В частности, мы можем ввести, если требуется, другой вид защиты, а также проверить систему компоновки и осуществить в ней поиск во всех исходных файлах, чтобы убедиться, что метод `setTestingInstance` не вызывается нетестированным кодом. То же самое мы можем сделать и с проверками во время выполнения. Если метод `setTestingInstance` вызывается во время выполнения, то мы можем выдать аварийное предупреждение или приостановить работу системы и подождать вмешательства со стороны оператора. Следует признать, что во многих языках, предшествовавших ООП, появление одиночек было невозможным, а разработчики все равно ухитрялись проектировать довольно надежные системы. Ведь в конечном итоге все сводится в ответственному проектированию и программированию.

Если нарушение свойства одиночки не вызывает серьезных осложнений, то можно руководствоваться групповым правилом. Например, каждый член группы разработчиков должен понимать, что в приложении имеется один экземпляр базы данных и другого быть не должно.

Для ослабления свойства одиночки в классе `PermitRepository` мы можем сделать его конструктор общедоступным. Это нас вполне устроит, если общедоступные методы в классе `PermitRepository` позволят нам делать все, что нам нужно, чтобы подготовить хранилище к тестированию. Так, если в классе `PermitRepository` имеется метод `addPermit` (добавить разрешение), позволяющий заполнить этот класс любыми разрешениями, которые могут потребоваться для наших тестов, то, возможно, будет достаточно создать хранилища и использовать их в наших тестах. В других случаях у нас может и не быть доступа, который нам требуется, а еще хуже, если одиночка будет делать нечто такое, чего нельзя допустить в средствах тестирования, например, обращение к базе данных в фоновом режиме. В подобных случаях мы можем выполнить *подклассификацию и переопределение метода* и создать производные классы, чтобы упростить тестирование.

Вернемся, однако, к примеру системы регистрации разрешений на строительство. Помимо метода и переменных, превращающих класс `PermitRepository` в одиночку, у нас имеется следующий метод:

```
public class PermitRepository
{
    ...
    public Permit findAssociatedPermit(PermitNotice notice) {
     // открыть базу данных разрешений на строительство
    ...

    // выбрать, используя переменные в заметке
    ...

    // проверить наличие лишь одного совпадающего
    // разрешения, иначе выдать сообщение об ошибке
    ...

    // возвратить совпавшее разрешение
    ...
    }
}
```

Если же нам требуется избежать обращения к базе данных, то мы можем выполнить подклассификацию класса `PermitRepository` следующим образом:

```
public class TestingPermitRepository extends PermitRepository
{
    private Map permits = new HashMap();

    public void addAssociatedPermit(PermitNotice notice, permit) {
        permits.put(notice, permit);
    }

    public Permit findAssociatedPermit(PermitNotice notice) {
        return (Permit)permits.get(notice);
    }
}
```

Сделав это, мы можем частично сохранить свойство одиночки. А поскольку мы используем подкласс класса `PermitRepository`, то можем сделать конструктор данного класса защищенным, а не общедоступным. Это позволит воспрепятствовать созданию более чем одного экземпляра класса `PermitRepository`, хотя допускает создание подклассов.

```
public class PermitRepository
{
    private static PermitRepository instance = null;

    protected PermitRepository() {}

    public static void setTestingInstance(PermitRepository newInstance)
    {
        instance = newInstance;
```

```
    }

    public static PermitRepository getInstance()
    {
        if (instance == null) {
            instance = new PermitRepository();
        }
        return instance;
    }

    public Permit findAssociatedPermit(PermitNotice notice)
    {
    ...
    }
    ...
}
```

В большинстве случаев мы можем воспользоваться *подклассификацией и переопределением метода* подобным образом, чтобы получить фиктивного одиночку прямо на месте. А в других случаях зависимости приобретают настолько обширный характер, что проще воспользоваться *извлечением интерфейса* из одиночки и затем заменить все ссылки в приложении на имя данного интерфейса. Для этого придется немало потрудиться, но мы можем сделать *упор на компилятор*, чтобы внести необходимые изменения в код. Вот как будет выглядеть класс `PermitRepository` после извлечения.

```
public class PermitRepository implements IPermitRepository
{
    private static IPermitRepository instance = null;

    protected PermitRepository() {}

    public static void setTestingInstance(IPermitRepository newInstance)
    {
        instance = newInstance;
    }

    public static IPermitRepository getInstance()
    {
        if (instance == null) {
            instance = new PermitRepository();
        }
        return instance;
    }

    public Permit findAssociatedPermit(PermitNotice notice)
    {
        ...
    }
    ...
}
```

Интерфейс `IPermitRepository` будет иметь сигнатуры для всех общедоступных нестатических методов из класса `PermitRepository`.

```
public interface IPermitRepository
{
    Permit findAssociatedPermit(PermitNotice notice);
    ...
}
```

Если вы программируете на языке, поддерживающем инструментальное средство реорганизации кода, то можете выполнить извлечение данного интерфейса автоматически. А если вы программируете на языке, не поддерживающем такое средство, то вам, возможно, проще будет воспользоваться *извлечением средства реализации*.

Вся эта реорганизация кода называется *вводом статического установщика*. Таким способом мы можем разместить тесты по местам, несмотря на обширный характер глобальных зависимостей. К сожалению, это мало помогает обойти глобальные зависимости. Если же приходится решать эту проблему, то лучше воспользоваться *параметризацией метода* и *параметризацией конструктора*. Эти виды реорганизации кода позволяют заменить глобальную ссылку на временную переменную в методе или же на поле в объекте. Недостаток параметризации метода заключается в том, что в конечном итоге получается слишком много методов, отвлекающих внимание разработчиков, когда они пытаются разобраться в классах. А недостаток параметризации конструктора состоит в том, что в каждом объекте, использующем в настоящий момент глобальную переменную, в итоге появляется дополнительное поле. Это поле приходится передавать его конструктору, а следовательно, классу, создающему данный объект, требуется также доступ к его экземпляру. Если же такое дополнительное поле потребуется многим объектам, то это существенно повлияет на объем оперативной памяти, используемой приложением, хотя нередко указывает и на другие недостатки проектирования.

Рассмотрим самый худший случай. Допустим, что у нас приложение с несколькими сотнями классов, во время выполнения которого создаются тысячи объектов, причем каждому из них требуется доступ к базе данных. Даже без анализа этого приложения с первого взгляда становится ясно, что если оно делает нечто более полезное, чем доступ к базе данных, то его функции можно разделить таким образом, чтобы одни классы занимались сохранением и выборкой данных, а другие — выполняли иные обязанности. Если мы очень постараемся разделить обязанности в таком приложении, то зависимости будут локализованы, и тогда нам, возможно, не придется обращаться к базе данных буквально в каждом объекте. Одни объекты будут заполняться данными, выбираемыми из базы данных, а другие — проводить расчеты по данным, предоставляемым их конструкторами.

В качестве упражнения выберите крупное приложение и попробуйте найти в нем глобальную переменную. Как правило, глобальные переменные доступны глобально, но они используются не глобально, а в относительно малом числе мест. Подумайте, как предоставить этот объект тем объектам, которым он требуется, если он не может быть глобальной переменной? Как реорганизовать код такого приложения? Имеются ли виды ответственности, которые можно разделить по рядам классов, чтобы сократить область действия глобальной переменной?

Если вы обнаружите глобальную переменную, которая используется буквально в каждом месте программы, то это означает, что в коде отсутствует какое-либо многоуровневое представление. Подробнее об этом речь пойдет в главах 15 и 17.

Пример ужасных зависимостей включения

C++ был первым моим языком ООП, и я должен признаться, что очень горд тем, что досконально изучил его особенности, достоинства и недостатки. Этот язык занял в свое время господствующее положение в программировании, поскольку он позволял совершенно прагматично решать многие неприятные проблемы. В частности, его дополнительные возможности позволяли использовать только свойства языка C для повышения эффективности программ, работавших на медленных машинах, а также писать и компилировать код C в виде подмножества C++, чтобы постепенно перейти от процедурного программирования на C к ООП.

Несмотря на то, что C++ стал на какое-то время весьма распространенным языком программирования, он в конце концов уступил в своей популярности Java и ряду других более новых языков. У него было преимущество обратной совместимости с C, но у других языков было еще большее преимущество простоты программирования. Программирующие на C++ неоднократно убеждались в том, что стандартные возможности этого языка далеко не идеальны для сопровождения программ, и поэтому им приходилось выходить за рамки этих возможностей, чтобы сохранить систему гибкой и удобной для изменений.

Особые трудности унаследования кода C в C++ связаны с согласованием разных частей программы. Если в Java и C# классу из одного файла требуется использовать класс из другого файла, то мы используем оператор `import` или `using`, чтобы сделать доступным определение данного класса. Компилятор находит этот класс и проверяет, был ли он уже скомпилирован. Если нет, то он компилируется. А если он скомпилирован, то компилятор читает краткий фрагмент данных из скомпилированного файла, получая ровно столько информации, сколько ему требуется, чтобы убедиться в том, что все методы, необходимые исходному классу, находятся в данном классе.

Такого рода оптимизация обычно отсутствует у компиляторов C++. Если в C++ одному классу требуется знать о другом классе, то объявление этого класса (как правило, в другом файле) текстуально включается в тот файл, где требуется его использовать. Этот процесс может оказаться существенно замедленным. Компилятору приходится повторно анализировать объявление данного класса и формировать внутреннее его представление всякий раз, когда он находит такое объявление. Более того, механизм включения допускает злоупотребления: один файл может включать в себя другой файл, а тот еще один файл и т.д. В тех проектах, где разработчики не избежали подобных злоупотреблений, нередко можно обнаружить небольшие файлы, включающие постепенные переходы к десяткам тысяч строк кода. Разработчики таких приложений удивляются, почему компоновка занимает у них столько времени. Но поскольку включения рассредоточены по всей системе, очень трудно выявить один конкретный файл и понять, почему он так долго компилируется.

Вам может показаться, что я сильно разочаровался в C++, но это не так. Этот язык очень важен, и на нем написано невероятное количество кода, но программировать на нем нужно очень аккуратно, чтобы написанный код работал как следует.

Работая с унаследованным кодом C++, зачастую бывает очень трудно получить экземпляр класса в средствах тестирования. Одним из самых первых препятствий, на которые мы при этом наталкиваемся, служит заголовочная зависимость. Какой заголовочный файл нам требуется, чтобы создать класс отдельно в средствах тестирования?

Ниже приведена часть объявления крупного класса C++ под названием `Scheduler` (планировщик). У него имеется свыше 200 методов, но в данном примере объявления показаны только пять из них. Помимо того, этот класс крупный, в нем имеются очень жес-

ткие и запутанные зависимости от многих других классов. Как же нам подвергнуть класс Scheduler тестированию?

```
#ifndef SCHEDULER_H
#define SCHEDULER_H

#include "Meeting.h"
#include "MailDaemon.h"
...
#include "SchedulerDisplay.h"
#include "DayTime.h"

class Scheduler
{
public:
        Scheduler(const string& owner);
        ~Scheduler();

    void addEvent(Event *event);
    bool hasEvents(Date date);
    bool performConsistencyCheck(string& message);
    ...
};

#endif
```

Среди прочего, в классе Scheduler используются объекты классов Meeting (совещание), MailDaemon (почтовый демон), Event (событие), SchedulerDisplay (отображение планировщика) и Date (дата). Если нам требуется проверить объекты класса Scheduler, то проще всего попытаться построить один из них в том же каталоге, но в другом файле под названием SchedulerTests (тесты планировщика). А зачем размещать тесты в том же каталоге? При наличии препроцессора это зачастую оказывается просто легче сделать. Если бы пути к включаемым файлам не использовались в проекте последовательно, то нам пришлось бы немало потрудиться, чтобы создать тесты в других каталогах.

```
#include "TestHarness.h"
#include "Scheduler.h"

TEST(create,Scheduler)
{
    Scheduler scheduler("fred");
}
```

Если мы создадим файл и попытаемся построить объект планировщика в тесте, то сразу же столкнемся с проблемой включения. Для того чтобы скомпилировать класс Scheduler, нам необходимо убедиться в том, что компилятору и компоновщику известно обо всех объектах, которые требуются классу Scheduler, а также обо всех других объектах, которые требуются этим объектам, и т.д. Правда, система компоновки выдаст нам немалое количество сообщений об ошибках и подробно укажет на все эти объекты.

В простейших случаях файл Scheduler.h включает в себя все, что требуется для создания объекта класса Scheduler, но в некоторых случаях в заголовочный файл включа-

ется не все, что для этого требуется. И тогда нам придется самим предоставить ряд дополнительных включений, чтобы создать и использовать объект класса `Scheduler`.

Мы могли бы просто скопировать все директивы `#include` из исходного файла для класса `Scheduler`, но дело в том, что не все они могут нам понадобиться. Поэтому лучше всего вводить эти директивы по очереди и по ходу дела решать, нужны ли нам эти конкретные зависимости. Мы можем нередко избежать их, вводя упреждающие объявления.

В идеальном случае было бы проще всего включить все файлы, которые нам понадобятся, и подождать до тех пор, пока не появятся ошибки компоновки, но это привело бы только к путанице. При наличии длинной цепочки переходных зависимостей нам пришлось бы включить больше файлов, чем действительно требуется. Даже если цепочка зависимостей окажется не слишком длинной, все равно останутся зависимости от объектов, с которыми очень трудно работать в средствах тестирования. В данном примере одну из таких зависимостей представляет класс `SchedulerDisplay`. Он не показан здесь, но на самом деле к нему осуществляется доступ в конструкторе класса `Scheduler`. От такой зависимости мы можем избавиться следующим образом:

```
#include "TestHarness.h"
#include "Scheduler.h"

void SchedulerDisplay::displayEntry(const string& entyDescription)
{
}

TEST(create,Scheduler)
{
    Scheduler scheduler("fred");
}
```

В приведенном выше фрагменте кода мы ввели альтернативное объявление `Scheduler Display::displayEntry`. К сожалению, после этого нам придется получить в данном файле отдельно скомпонованные контрольные примеры. Для каждого метода из класса `SchedulerDisplay` мы можем иметь лишь одно объявление в программе, и поэтому нам потребуется отдельная программа для тестирования объектов планировщика.

Правда, мы можем повторно использовать для этой цели некоторые фиктивные объекты. Вместо размещения определений классов, подобных приведенному выше встроенному объявлению класса `SchedulerDisplay` в тестовом файле, мы можем поместить их в отдельный заголовочный файл, чтобы использовать его в целом ряде тестовых файлов.

```
#include "TestHarness.h"
#include "Scheduler.h"
#include "Fakes.h"

TEST(create,Scheduler)
{
    Scheduler scheduler("fred");
}
```

Если вы попробуете хотя бы пару раз получить экземпляр объекта C++ в средствах тестирования, то данная процедура покажется вам простой и механической, но у нее имеются и серьезные недостатки. Во-первых, вам придется создать отдельную программу, а во-вторых, вы на самом деле не разрываете зависимости на уровне языка программирования, и поэтому код не становится от этого яснее. Более того, дублированные объявления,

размещаемые в тестовом файле (в данном примере — объявление `SchedulerDisplay::displayEntry`), должны сохраняться до тех пор, пока тесты находятся на своих местах.

Данный способ подходит для тех случаев, когда в крупном классе имеются очень серьезные трудности, связанные с зависимостями. Но это не тот способ, которым можно пользоваться часто или просто. Если класс приходится разделять на большое количество мелких классов, то создание отдельной тестовой программы для такого класса окажется очень полезным. Со временем эта отдельная программа может и не понадобиться по ходу извлечения дополнительных классов для тестирования.

Пример многослойного параметра

Мне нравятся простые конструкторы. Что лучше возможности создать класс, а затем просто набрать вызов его конструктора и получить аккуратный рабочий объект, готовый к применению? Но зачастую объекты создаются трудно. Ведь каждый объект приходится устанавливать в нужное состояние, в котором он готов к дальнейшей работе. Как правило, это означает, что ему требуется предоставить другие объекты, которые также должны быть установлены в правильное состояние. А этим объектам могут потребоваться еще одни объекты, устанавливаемые в соответствующее состояние, и в итоге параметр конструктора тестируемого класса составляется из целой цепочки создаваемых объектов. Все эти объекты наслаиваются как кожура луковицы. Ниже приведен пример проблемы подобного рода. Допустим, что у нас имеется следующий класс, отображающий объект `SchedulingTask` (задача планирования):

```
public class SchedulingTaskPane extends SchedulerPane
{
    public SchedulingTaskPane(SchedulingTask task) {
        ...
    }
}
```

Для создания объекта этого класса нам придется передать ему объект `SchedulingTask`, но для построения объекта `SchedulingTask` нам необходимо воспользоваться одним и только одним конструктором.

```
public class SchedulingTask extends SerialTask
{
    public SchedulingTask(Scheduler scheduler, MeetingResolver resolver)
    {
        ...
    }
}
```

Если мы обнаружим, что нам потребуются дополнительные объекты для создания объектов классов `Scheduler` и `MeetingResolver` (разрешатель совещаний), то это будет стоить нам немалых хлопот. Единственное, что может утешить нас и не довести до полного отчаяния, это тот факт, что должен быть хотя бы один класс, которому не потребуются объекты другого класса в качестве аргументов. Ведь если такой класс отсутствует, то систему вообще нельзя скомпилировать.

Для того чтобы справиться с подобной ситуацией, следует тщательно продумать свои действия. В частности, нам нужно написать тесты, но что нам действительно нужно от па-

раметров, передаваемых в конструкторе? Если нам от них ничего особенного не нужно в наших тестах, то мы можем просто *передать пустое значение*. А если нам требуется некоторое рудиментарное поведение, то мы можем воспользоваться *извлечением интерфейса* или же *извлечением средства реализации* из самой прямой зависимости и использовать полученный таким образом интерфейс для создания фиктивного объекта. В данном случае наблюдается самая прямая зависимость класса `SchedulingTaskPane` (панель задач планирования) от класса `SchedulingTask`. Если нам удастся создать фиктивный объект класса `SchedulingTask`, то мы сможем построить объект класса `SchedulingTaskPane`.

К сожалению, класс `SchedulingTask` наследует от класса под названием `SerialTask` (последовательная задача), и единственной его функцией является переопределение некоторых защищенных методов. А все общедоступные методы находятся в классе `SerialTask`. Можем ли мы воспользоваться извлечением интерфейса из класса `SchedulingTask` или же нам придется совершить это и по отношению к классу `SerialTask`? В Java нам не придется этого делать, поскольку мы можем создать интерфейс для класса `SchedulingTask`, включающий в себя, среди прочего, методы из класса `SerialTask`.

Полученная в итоге иерархия классов должна выглядеть так, как показано на рис. 9.3.

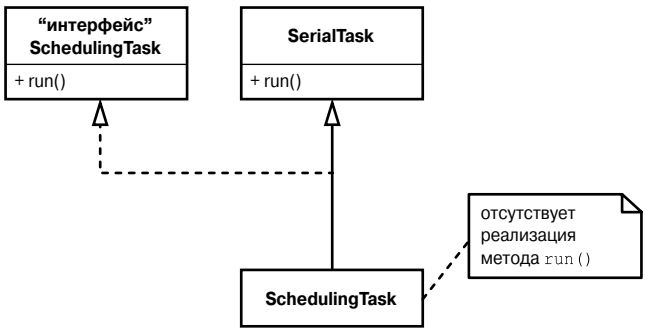


Рис. 9.3. Класс `SchedulingTask` и его иерархия

В данном случае нам повезло в том отношении, что мы пользуемся Java. К сожалению, в C++ нам не удалось бы разрешить эту ситуацию подобным образом. Ведь в этом языке отсутствует отдельная конструкция интерфейса. Интерфейсы, как правило, реализуются в C++ в виде классов, содержащих лишь чистые виртуальные функции. Если бы данный пример был перенесен из C++, то класс `SchedulingTask` стал бы абстрактным, поскольку он наследует чистую виртуальную функцию от исходного класса `SchedulingTask`. Для получения экземпляра класса `SchedulingTask` нам пришлось бы предоставить тело метода `run()` из класса `SchedulingTask`, который делегирует соответствующие полномочия методу `run()` из класса `SerialTask`. К счастью, сделать это не очень сложно. Вот как это выглядит в коде.

```
class SerialTask
{
public:
    virtual void run();
    ...
};

class ISchedulingTask
```

```
{
public:
    virtual void run() = 0;
    ...
};

class SchedulingTask : public SerialTask, public ISchedulingTask
{
public:
    virtual void run() { SerialTask::run(); }
};
```

В любом языке программирования, где у нас имеются средства для создания интерфейсов или же классов, действующих как интерфейсы, мы можем систематически использовать и те и другие для разрывания зависимостей.

Пример совмещенного параметра

Если в конструкторах встречаются параметры, которые нам мешают, то в качестве выхода из этого затруднительного положения мы можем зачастую воспользоваться *извлечением интерфейса* или же *извлечением средства реализации*. Но иногда такой подход оказывается непрактичным. Рассмотрим для примера еще один класс из упоминавшейся ранее системы регистрации разрешений на строительство.

```
public class IndustrialFacility extends Facility
{
    Permit basePermit;

    public IndustrialFacility(int facilityCode, String owner,
             OriginationPermit permit) throws PermitViolation {

        Permit associatedPermit =
                PermitRepository.GetInstance()
                    .findAssociatedFromOrigination(permit);

        if (associatedPermit.isValid() && !permit.isValid()) {
            basePermit = associatedPermit;
        }
        else if (!permit.isValid()) {
            permit.validate();
            basePermit = permit;
       }
       else
            throw new PermitViolation(permit);
    }
   ...
}
```

Нам требуется получить экземпляр данного класса в средствах тестирования, но для этого нам придется преодолеть пару препятствий. Одно из них состоит в том, что мы вновь получаем доступ к одиночке (классу `PermitRepository`). Это препятствие мы можем

обойти, используя способы, представленные ранее в разделе "Пример раздражающей глобальной зависимости". Но прежде нам придется преодолеть другое препятствие: очень трудно создать исходное разрешение (объект класса `OriginationPermit`), которое нам требуется передать конструктору, поскольку у объектов класса `OriginationPermit` ужасно запутанные зависимости. В связи с этим сразу же возникает мысль воспользоваться извлечением интерфейса из класса `OriginationPermit`, чтобы обойти подобную зависимость. Но, к сожалению, сделать это не так-то просто. На рис. 9.4 показана структура иерархии класса `Permit`.

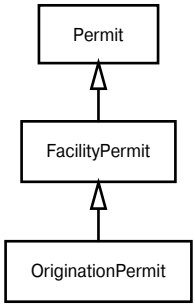


Рис. 9.4. Иерархия класса `Permit`

Конструктор класса `IndustrialFacility` (промышленный строительный объект) воспринимает объект класса `OriginationPermit` и обращается к объекту класса `PermitRepository`, чтобы получить ассоциированное разрешение. Воспользуемся методом из класса `PermitRepository`, воспринимающим объект класса `OriginationPermit` и возвращающим объект класса `Permit`. Если в архиве найдено ассоциированное разрешение, то оно будет сохранено в поле объекта `Permit`. В противном случае в этом поле будет сохранен объект класса `OriginationPermit`. Мы могли бы создать интерфейс для класса `OriginationPermit`, но это мало что бы нам дало. Кроме того, мы могли бы присвоить интерфейс `IOriginationPermit` полю `Permit`, но и это не очень бы помогло. В Java интерфейсы не могут наследовать от классов. Поэтому наиболее очевидное решение заключается в том, чтобы создать интерфейсы вниз по иерархии и преобразовать поле `Permit` в поле `IPermit`. На рис. 9.5 показано, как это будет выглядеть в схематическом виде.

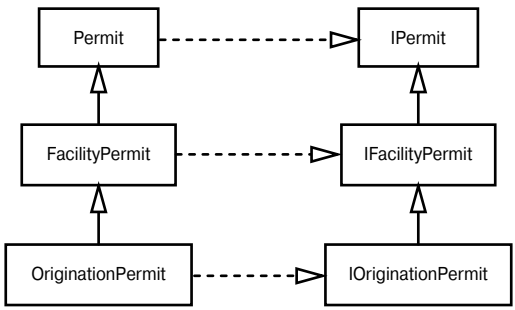


Рис. 9.5. Иерархия класса `Permit` с извлеченными интерфейсами

Пожалуй, для этого потребуется немало нелепой работы, а в итоге код получится не особенно привлекательным. Интерфейсы отлично подходят для разрывания зависимостей, но когда мы доходим до взаимно однозначного соответствия классов и интерфейсов, то структура кода становится слишком громоздкой. Прошу понять меня правильно: к такой структуре можно перейти, когда просто нет иного выхода, но когда имеются другие возможности, мы должны их непременно рассмотреть. К счастью, они имеются.

Извлечение интерфейса — это лишь один из многих способов разрывания зависимости от параметра. В одних случаях целесообразно задаться вопросом, почему такая зависимость — это плохо, в других — создание параметра затруднительно, в третьих — параметр дает неприятный побочный эффект (он может обусловить обращение к файловой системе или базе данных), а в четвертых — код параметра может выполняться слишком долго. Когда мы пользуемся извлечением интерфейса, то можем обойти все эти препятствия, но за счет грубого разрывания связи с классом. Если же осложнения вызывают только отдельные части класса, то мы можем применить другой подход и просто разорвать связь именно с ними.

Проанализируем класс `OriginationPermit` более внимательно. Он не нужен нам для тестирования, поскольку он незаметно обращается к базе данных, когда мы предписываем ему проверить себя на достоверность.

```
public class OriginationPermit extends FacilityPermit
{
    ...
    public void validate() {
    // установить соединение с базой данных
    ...

    // запросить информацию проверки достоверности
    ...

    // установить признак проверки достоверности
    ...

    // закрыть базу данных
    ...
    }
}
```

Нам такие действия в тесте не нужны, поскольку нам пришлось бы ввести фиктивные данные в базу данных и тем самым сильно огорчить администратора базы данных. Ведь у него и без того немало хлопот, да и работа у него, прямо скажем, не из легких.

Другой подход заключается в использовании *подклассификации и переопределения метода*. Так, мы можем создать сначала класс под названием `FakeOriginationPermit` (подделка исходного разрешения), предоставляющий методы, упрощающие изменение признака проверки достоверности, а затем переопределить в подклассах метод `validate` (проверить на достоверность) и установить признак проверки достоверности так, как нам будет нужно, в ходе тестирования класса `IndustrialFacility`. Ниже приведен первый подходящий для этой цели тест.

```
public void testHasPermits() {
    class AlwaysValidPermit extends FakeOriginationPermit
    {
```

```
        public void validate() {
            // установить признак проверки достоверности
            becomeValid();
        }
    };

    Facility facility = new IndustrialFacility(Facility.HT_1, "b",
              new AlwaysValidPermit());
    assertTrue(facility.hasPermits());
}
```

Во многих языках программирования классы можно создавать "на ходу" подобно приведенному выше методу. Это очень удобно для тестирования, хотя в выходном коде я стараюсь делать это как можно реже. Подобным способом можно очень просто составлять контрольные примеры для особых случаев.

Подклассификация и переопределение метода помогает нам разорвать зависимости от параметров, но иногда вынесение методов за скобки в классе не совсем подходит для этой цели. В рассмотренном выше примере нам повезло, поскольку ненужные нам зависимости были обособлены в методе `validate`. В особо тяжелых случаях эти зависимости настолько переплетаются с нужной нам логикой, что приходится сначала извлекать методы. При наличии инструментального средства реорганизации кода сделать это будет нетрудно. В противном случае могут пригодиться некоторые способы, рассматриваемые в главе 22.

Глава 10

Метод нельзя выполнить в средствах тестирования

Размещение тестов по местам для внесения изменений в код иногда оказывается не совсем простым делом. Если вы можете получить экземпляр класса отдельно в средствах тестирования, то считайте, что вам просто повезло. Но такая удача сопутствует разработчикам очень редко. Если же вы испытываете затруднения при вводе класса в средства тестирования, обратитесь к материалу главы 9.

Получить экземпляр класса — зачастую лишь полдела, а далее нужно написать тесты для методов, в которые требуется внести изменения. Иногда мы можем сделать это, вообще не получая экземпляр класса. Так, если в методе не особенно используются данные экземпляра, то мы можем применить *раскрытие статического метода*, чтобы получить доступ к коду. Если же метод оказывается довольно длинным и трудным в обращении, то лучше воспользоваться *выносом объекта метода*, чтобы перенести код в тот класс, экземпляр которого нам легче получить.

К счастью, объем работы по написанию тестов для методов обычно невелик. Ниже перечислены препятствия, которые могут встретиться нам на этом пути.

- Метод бывает недоступным для тестирования. Он может оказаться частным или же представлять иные трудности для доступа.
- Вызов метода может оказаться затруднительным, поскольку трудно построить параметры, которые ему требуются.
- Метод может давать неприятные побочные эффекты (видоизменение базы данных, запуск крылатой ракеты и пр.), в связи с чем его нельзя выполнить в средствах тестирования.
- Требуется распознавание объекта, используемого в методе.

В остальной части этой главы рассматриваются примеры, демонстрирующие разные способы преодоления перечисленных выше препятствий и затруднений, а также некоторые компромиссные решения.

Пример скрытого метода

Допустим, что нам требуется внести изменения в метод из определенного класса, но этот метод частный. Что мы должны для этого сделать?

Прежде всего, мы должны выяснить, можем ли мы протестировать общедоступный метод. Если да, то сделать это стоит по двум причинам. Во-первых, мы избавим себя от хлопот, пытаясь найти способ доступа к частному методу. А во-вторых, тестируя общедоступные методы, мы тем самым гарантированно проверяем метод в том виде, в каком он используется в коде. Это поможет нам ввести нашу работу с этим методом в какое-то определенное русло. В унаследованном коде нередко встречаются методы сомнительного

качества, беспорядочно разбросанные внутри классов. Объем работы по реорганизации кода, которую приходится выполнять, чтобы сделать частный метод полезным для всех вызывающих его частей программы, может оказаться довольно большим. Конечно, неплохо иметь самые общие методы, полезные для многих вызывающих частей программы, но дело в том, что у каждого метода должно быть лишь столько функций, сколько требуется для поддержки вызывающих его частей программы, и в то же время он должен быть достаточно ясным для понимания, чтобы в него можно было легко внести изменения. Если мы протестируем частный метод общедоступными методами, которые им пользуются, то можем не опасаться того, что сделаем его слишком общим. Если же метод должен однажды стать общедоступным, то первый же его пользователь вне его класса должен написать контрольные примеры, точно разъясняющие, что этот метод делает и как им правильно пользоваться.

Все это, конечно, замечательно, но нам иногда требуется просто написать контрольный пример для частного метода, вызов которого глубоко скрыт в классе. Ведь нам нужна конкретная ответная реакция и тесты, объясняющие, как пользоваться этим методом, хотя вполне возможно, что его тестирование общедоступными методами в соответствующем классе просто затруднительно.

Как же нам написать тесты для частного метода? Это едва ли не самый распространенный вопрос, возникающий по поводу тестирования кода. Правда, на него существует очень прямой ответ: если нам требуется протестировать частный метод, мы должны сделать его общедоступным. Если же нас беспокоит, как сделать метод общедоступным, то зачастую это означает, что соответствующий класс выполняет слишком много функций, и нам придется с этим что-то делать. Ниже приведены некоторые причины, по которым нас может беспокоить, как сделать метод общедоступным.

1. Метод представляет собой утилиту, которая не особенно заботит клиентов.
2. Если клиенты пользуются данным методом, то они могут отрицательно повлиять на результаты выполнения других методов из того же класса.

Первая причина не очень серьезна. Наличие дополнительного общедоступного метода в интерфейсе класса в данном случае вполне простительно, хотя мы должны постараться выяснить, не лучше ли перенести такой метод в другой класс. Вторая причина представляется несколько более серьезной, хотя для ее устранения имеется следующее действенное средство: частные методы можно перенести в новый класс. Их можно сделать общедоступными в новом классе и создать его внутренний интерфейс в исходном классе. Благодаря этому методы становятся тестируемыми, а структура кода улучшается.

Я, конечно, понимаю, что такой совет может показаться слишком голословным, но его нельзя все же назвать совершенно бесполезным. Ведь непреложным остается тот факт, что удачная структура кода тестируется, а нетестируемая структура кода неудачна. Ответы на вопросы, возникающие в подобных случаях, следует искать, пользуясь способами, приведенными в главе 20. Но если на местах находится не очень много тестов, то нам, возможно, придется продвигаться в работе с кодом более осторожно и приложить дополнительные усилия, чтобы разделить его на удобные для проверки части.

Покажем, как разрешать подобные затруднения на конкретном примере. Ниже приведена часть объявления класса в C++.

```
class CCAImage
{
private:
    void setSnapRegion(int x, int y, int dx, int dy);
    ...
```

```
public:
    void snap();
    ...
};
```

Класс `CCAImage` служит для получения снимков в системе защиты. Невольно возникает вопрос: почему класс изображения отвечает за получение снимков? Дело в том, что это унаследованный код. У данного класса имеется метод `snap()`, управляющий фотокамерой с помощью низкоуровневого интерфейса API на языке C для получения снимка, хотя это снимок особого рода. В течение одного вызова метода `snap()` выполняются разные действия с фотокамерой, в результате чего делается снимок, который затем помещается в другой части буфера изображения, хранящегося в данном классе. Логика принятия решения относительно места расположения каждого снимка отличается своей динамичностью — все зависит от характера движения объекта съемки, фиксируемого на снимке. В зависимости от того, как объект съемки движется, метод `snap()` может повторно вызвать метод `setSnapRegion`, чтобы определить место для размещения текущего снимка в буфере изображения. К сожалению, интерфейс API фотокамеры изменился, и поэтому нам придется внести соответствующие изменения в метод `setSnapRegion` (задать область фиксации изображения). Что же мы должны для этого сделать?

Прежде всего, мы могли бы просто сделать данный метод общедоступным. Но, к сожалению, это имело бы ряд весьма отрицательных последствий. Класс `CCAImage` опирается на ряд переменных, определяющих текущее местоположение области фиксации. Если попытаться вызвать метод `setSnapRegion` в выходном коде за пределами метода `snap()`, то это привело бы к серьезным осложнениям в работе следящей системы фотокамеры.

Итак, мы столкнулись с серьезной проблемой. Но прежде чем искать ее решение, следует уяснить, как она возникла. Истинная причина того, что мы не можем как следует протестировать класс изображения, заключается в том, что у него слишком много видов ответственности. В идеальном случае было бы неплохо разделить этот класс на более мелкие подклассы, используя способы, представленные в главе 20, но нам придется очень внимательно проанализировать, сможем ли мы вообще реорганизовать код подобным образом прямо сейчас. Конечно, было бы неплохо сделать это сразу, но все зависит от того, на какой стадии выпуска обновленной версии программы мы находимся, сколько у нас имеется времени и насколько мы можем рисковать.

Если мы не можем себе позволить разделение ответственности прямо сейчас, то есть ли у нас возможность написать тесты для метода, который мы изменяем? К счастью, такая возможность у нас есть. И вот как мы могли бы поступить.

Прежде всего, нам следует изменить метод `setSnapRegion` с частного на защищенный.

```
class CCAImage
{
protected:
    void setSnapRegion(int x, int y, int dx, int dy);
    ...
public:
    void snap();
   ...
};
```

Затем мы можем выполнить подклассификацию класса `CCAImage`, чтобы получить доступ к данному методу.

```
class TestingCCAImage : public CCAImage
{
public:
    void setSnapRegion(int x, int y, int dx, int dy)
    {
        // вызвать метод setSnapRegion из суперкласса
        CCAImage::setSnapRegion(x, y, dx, dy);
    }
};
```

В современных компиляторах C++ для автоматического делегирования можно также воспользоваться следующим объявлением `using` в тестовом подклассе:

```
class TestingCCAImage : public CCAImage
{
public:
    // раскрыть все реализации метода setSnapRegion из класса CCAImage
    // в виде общедоступного интерфейса. Делегировать его вызовы
    // классу CCAImage.
        using CCAImage::setSnapRegion;
    }
```

После этого мы можем вызвать метод `setSnapRegion` из класса `CCAImage` непосредственно в тесте, хотя и косвенно. Но насколько это целесообразно? Прежде нам не хотелось делать метод общедоступным, но ведь мы теперь совершаем нечто подобное: превращая метод в защищенный, мы делаем его более доступным.

Откровенно говоря, я предпочел бы выбрать более надежный путь, расположив тесты на местах. Конечно, такое изменение позволяет нам нарушить инкапсуляцию. Когда мы размышляем над тем, как работает код, мы должны принимать во внимание тот факт, что метод `setSnapRegion` можно теперь вызвать и в подклассах. И хотя это не так существенно, тем не менее, может оказаться достаточной причиной для того, чтобы побудить нас к полной реорганизации кода в следующий раз, когда дело коснется изменений в классе `CCAImage`. В частности, мы могли бы разделить виды ответственности в классе `CCAImage` по разным подклассам и сделать их тестируемыми.

Нарушение защиты от доступа

Во многих более новых, чем C++, языках ООП мы можем использовать рефлексию или специальные полномочия для доступа к частным переменным во время выполнения. Несмотря на очевидные удобства, такое свойство на самом деле имеет несколько искусственный характер. Оно оказывается очень полезным для разрыва зависимостей, но в своих проектах я стараюсь не размещать тесты с доступом к частным переменным. Такого рода ухищрение только мешает разработчикам замечать, насколько неудачным оказывается их код. Насколько бы это ни показалось садистским, но страдания, которые нам доставляет работа с унаследованным кодом, могут послужить очень сильным стимулом к его изменению. Конечно, мы могли бы уклониться от этих страданий, но если речь не идет о первопричинах, классах, слишком перегруженных разными видами ответственности, и запутанных зависимостях, то тем самым мы только откладываем на время неизбежную расплату. Когда всем станет ясно, что код получился неудачным, то затраты на его улучшение покажутся до смешного незначительными.

Пример “полезного” свойства языка

Разработчики языков программирования обычно стараются сделать все возможное, чтобы облегчить участь тех, кто на этих языках программирует, хотя задача у них не из простых. Ведь им приходится соблюдать тонкий баланс между простотой программирования и вопросами безопасности и надежности. Некоторые свойства языка, на первый взгляд, вполне способны урегулировать все эти вопросы, но как только мы попытаемся протестировать код, то сразу же сталкиваемся с суровой реальностью.

Ниже приведен фрагмент кода C#, воспринимающего совокупность файлов выгружаемых из веб-клиента. Все они выбираются по очереди, после чего в данном коде возвращается список потоков, связанных с файлами, обладающими особыми свойствами.

```
public void IList getKSRStreams(HttpFileCollection files) {
    ArrayList list = new ArrayList();
    foreach(string name in files) {
        HttpPostedFile file = files[name];
        if (file.FileName.EndsWith(".ksr") ||
            (file.FileName.EndsWith(".txt")
                    && file.ContentLength > MIN_LEN)) {
            ...
            list.Add(file.InputStream);
        }
    }
    return list;
}
```

Нам бы хотелось внести некоторые изменения в этот фрагмент кода, а возможно, и немного реорганизовать его, но написать тесты для этого кода не так-то просто. В частности, нам хотелось бы создать объект `HttpFileCollection` (совокупность HTTP-файлов) и заполнить его объектами `HttpPostedFile` (отправленные HTTP-файлы), но это просто невозможно. Во-первых, у класса `HttpPostedFile` отсутствует конструктор. И во-вторых, этот класс герметичен. В C# это означает, что мы не можем создать экземпляр класса `HttpPostedFile` и выполнить его подклассификацию. Класс `HttpPostedFile` входит в состав библиотеки .NET. Во время выполнения его экземпляры создаются другим классом, но у нас нет к нему доступа. Беглый взгляд на класс `HttpFileCollection` обнаруживает в нем аналогичные затруднения: отсутствие общедоступных конструкторов и возможности создавать порожденные классы.

Для чего же компания Microsoft предоставила этот класс в наше распоряжение? Ведь для того чтобы воспользоваться им, мы приобрели соответствующую лицензию и регулярно обновляем ее. Я далек от мысли обвинять компанию Microsoft в каком-то злом умысле против программистов, тем более, что аналогичные препятствия для подклассификации мы испытываем, работая с продукцией компании Sun. Так, ключевое слово `final` в Java служит для обозначения классов, особенно восприимчивых с точки зрения безопасности. Ведь если бы кому-нибудь удалось создать подкласс класса `HttpPostedFile` или даже такого класса, как `String`, то он мог бы написать некоторый злонамеренный код и передать его коду, использующему эти классы. Такая опасность весьма реальна, но свойства `sealed` и `final` неприятны тем, что сковывают наши действия.

Что же мы можем предпринять, чтобы написать тесты для метода `getKSRStreams` (получить потоки клавишного приемопередатчика)? Мы не можем, в частности, вос-

пользоваться *извлечением интерфейса* или же *извлечением средства реализации*. Классы `HttpPostedFile` и `HttpFileCollection` нам неподвластны, поскольку они являются библиотечными классами и не подлежат изменению. На первый взгляд, единственным подходящим в данной ситуации способом является *адаптация параметра*.

Правда, в данном случае нам немного повезло, поскольку мы можем обращаться к совокупности выгружаемых файлов в цикле. К тому же, у герметичного класса `HttpFileCollection`, используемого в рассматриваемом здесь фрагменте кода, имеется негерметичный суперкласс `NameObjectCollectionBase` (база совокупности именованных объектов). Поэтому мы можем выполнить его подклассификацию и передать объект полученного в итоге подкласса методу `getKSRStreams`. Такое изменение окажется вполне безопасным и простым, если мы сделаем *упор на компилятор*.

```
public void LList getKSRStreams(OurHttpFileCollection files) {
    ArrayList list = new ArrayList();
    foreach(string name in files) {
        HttpPostedFile file = files[name];
        if (file.FileName.EndsWith(".ksr") ||
            (file.FileName.EndsWith(".txt")
                    && file.ContentLength > MAX_LEN)) {
            ...
            list.Add(file.InputStream);
        }
    }
    return list;
}
```

В приведенном выше фрагменте кода `OurHttpFileCollection` является подклассом класса `NameObjectCollectionBase`, а тот — абстрактным классом, связывающим строки с объектами.

Итак, мы обошли одно препятствие, но преодолеть следующее препятствие будет посложнее. Нам потребуются объекты класса `HttpPostedFile`, чтобы выполнить метод `getKSRStreams` в тесте, но мы не можем их создать. А что нам, собственно, от них нужно? Нам нужен класс, предоставляющий пару следующих свойств: `FileName` (имя файла) и `ContentLength` (длина содержимого). Для отделения класса `HttpPostedFile` мы можем воспользоваться *заключением интерфейса API в оболочку*. С этой целью нам достаточно извлечь интерфейс (`IHttpPostedFile`) и написать оболочку (`HttpPostedFileWrapper`).

```
public class HttpPostedFileWrapper : IHttpPostedFile
{
    public HttpPostedFileWrapper(HttpPostedFile file) {
        this.file = file;
    }

    public int ContentLength {
        get { return file.ContentLength; }
    }
    ...
}
```

А поскольку теперь у нас имеется интерфейс, мы можем также создать класс для тестирования.

```
public class FakeHttpPostedFile : IHttpPostedFile
{
    public FakeHttpPostedFile(int length, Stream stream, ...) { ... }

    public int ContentLength {
        get { return length; }
    }
}
```

Если теперь мы сделаем *упор на компилятор* и внесем изменения в выходной код, то сможем обратиться к объектам `HttpPostedFileWrapper` (оболочка отправленных HTTP-файлов) или `FakeHttpPostedFile` (подделка отправленных HTTP-файлов) с помощью интерфейса `IHttpPostedFile`, даже не зная, какой из них на самом деле используется.

```
public IList getKSRStreams(OurHttpFileCollection) {
    ArrayList list = new ArrayList();
    foreach(string name in files) {
        IHttpPostedFile file = files[name];
        if (file.FileName.EndsWith(".ksr") ||
            (file.FileName.EndsWith(".txt"))
                    && file.ContentLength > MAX_LEN)) {
            ...
            list.Add(file.InputStream);
        }
    }
    return list;
}
```

В данном случае досаждает лишь необходимость циклического обращения к исходному классу `HttpFileCollection` в выходном коде, заключение в оболочку каждого содержащегося в нем объекта класса `HttpPostedFile` и последующий ввод этого объекта в новую совокупность, передаваемую методу `getKSRStreams`. Но это цена, которую приходится платить за безопасность.

Очень легко поверить в то, что свойства `sealed` и `final` являются явной ошибкой и что их вообще не следовало вводить в языки программирования. Но на самом деле ошибку совершаем мы, когда допускаем зависимость от библиотек, которые нам неподвластны, и тем самым навлекаем на себя неприятности.

Когда-нибудь в основных языках программирования, возможно, будут предоставлены особые полномочия доступа для тестирования, а до тех пор такими механизмами, как `sealed` и `final`, следует пользоваться весьма экономно. Когда же потребуются классы, в которых применяются подобные механизмы, их целесообразно изолировать внутри некоторой оболочки, чтобы получить достаточную свободу для внесения изменений в код. Дополнительные сведения о способах решения подобной проблемы приведены в главе 14.

Пример необнаруживаемого побочного эффекта

Теоретически в написании теста для проверки функций отдельного фрагмента кода нет ничего плохого. Мы получаем экземпляр класса, вызываем его методы и проверяем результаты, которые они возвращают. Что же здесь может оказаться неверным? Ничего,

если бы объект, который мы создаем, не был связан с какими-нибудь другими объектами. Если он используется другими объектами, а сам не пользуется ничем другим, то мы также можем воспользоваться им в своих тестах, действующих аналогично остальной части нашей программы. Но, к сожалению, объекты, не пользующиеся другими объектами, встречаются очень редко.

Программы зачастую рассчитаны на автономное выполнение. В них нередко используются объекты с методами, которые вообще ничего не возвращают. Вызывающей части программы ничего не известно, что происходит внутри таких методов. Одни объекты вызывают методы других объектов, а к чему это приведет — неизвестно.

Рассмотрим пример класса, для которого характерно подобное поведение.

```
public class AccountDetailFrame extends Frame
        implements ActionListener, WindowListener
{
    private TextField display = new TextField(10);
    ...
    public AccountDetailFrame(...) { ... }

    public void actionPerformed(ActionEvent event) {
        String source = (String)event.getActionCommand();
        if (source.equals("project activity")) {
            detailDisplay = new DetailFrame();
            detailDisplay.setDescription(
                getDetailText() + " " + getProjectionText());
            detailDisplay.show();
            String accountDescription
                = detailDisplay.getAccountSymbol();
            accountDescription += ": ";
            ...
            display.setText(accountDescription);
            ...
       }
    }
   ...
}
```

Этот старый класс Java выполняет следующие действия: создает компоненты пользовательского интерфейса, получает от них уведомления с помощью обработчика `actionPerformed` (выполненные действия) и определяет, что нужно и чего не нужно отображать. И все это делается довольно странным образом: сначала составляется подробный текст, который затем отображается в отдельно создаваемом окне. Когда окно выполнит свою функцию, прямо из него собирается соответствующая информация, которая затем немного обрабатывается и сохраняется в отдельных текстовых полях.

Метод данного класса можно было бы попытаться выполнить в средствах тестирования, не это не имеет никакого смысла. Ведь он создал бы окно, отобразил бы в нем некоторую информацию с приглашением для ввода, а затем перешел бы к отображению очередной информации в другом окне. В таком коде отсутствует подходящее место для распознавания того, что он делает.

Что же мы можем предпринять? Прежде всего, мы можем отделить часть кода, не зависящую от графического пользовательского интерфейса, от той части кода, которая от него зависит. А поскольку мы имеем в данном случае дело с кодом Java, то можем воспользо-

ваться одним из доступных в этом языке инструментальных средств реорганизации кода. Для разделения кода в рассматриваемом здесь методе нам нужно выполнить ряд реорганизаций кода типа *извлечения метода*.

С какого же места нам следует начать?

Сам метод представляет собой перехватчик уведомлений, поступающих от многооконной среды. Прежде всего, он получает имя команды из события действия, которое ему передается. Если мы извлечем все тело данного метода, то сможем избавиться от любой зависимости от класса `ActionEvent` (событие действия).

```
public class AccountDetailFrame extends Frame
        implements ActionListener, WindowListener
{
    private TextField display = new TextField(10);
    ...
    public AccountDetailFrame(...) { ... }

    public void actionPerformed(ActionEvent event) {
        String source = (String)event.getActionCommand();
        performCommand(source);
    }

    public void performCommand(String source) {
        if (source.equals("project activity")) {
            detailDisplay = new DetailFrame();
            detailDisplay.setDescription(
                  getDetailText() + " " + getProjectionText());
            detailDisplay.show();
            String accountDescription
                  = detailDisplay.getAccountSymbol();
            accountDescription += ": ";
            ...
            display.setText(accountDescription);
            ...
        }
    }
    ...
}
```

Но этого явно недостаточно для того, чтобы сделать код тестируемым. Поэтому нам предстоит далее извлечь методы для кода доступа к другой рамке. Это позволит сделать рамку `detailDisplay` (подробное отображение) переменной экземпляра класса.

```
public class AccountDetailFrame extends Frame
        implements ActionListener, WindowListener
{
    private TextField display = new TextField(10);
    private DetailFrame detailDisplay;
...
    public AccountDetailFrame(...) { .. }

    public void actionPerformed(ActionEvent event) {
        String source = (String)event.getActionCommand();
```

```
        performCommand(source);
    }

    public void performCommand(String source) {
        if (source.equals("project activity")) {
            detailDisplay = new DetailFrame();
            detailDisplay.setDescription(
                getDetailText() + " " + getProjectionText());
           detailDisplay.show();
           String accountDescription
               = detailDisplay.getAccountSymbol();
           accountDescription += ": ";
           ...
           display.setText(accountDescription);
           ...
       }
    }
    ...
}
```

Теперь мы можем извлечь код, использующий данную рамку, в ряд методов. Как же нам назвать эти методы? Для этого мы должны проанализировать, что именно делает каждый извлеченный фрагмент кода с точки зрения его класса, т.е. что именно он вычисляет для данного класса. Кроме того, мы не должны пользоваться именами, в которых упоминаются компоненты отображения. Конечно, мы можем пользоваться компонентами отображения в таком извлеченном фрагменте кода, но имена должны скрывать этот факт. С учетом этого мы можем создать для каждого фрагмента кода командный или запросный метод.

Разделение команд и запросов

Разделение команд и запросов — это принцип проектирования, впервые сформулированный Бертраном Мейером. Проще говоря, этот принцип требует следующего: метод может быть либо командным, либо запросным, но не тем и не другим одновременно. Командным называется такой метод, который видоизменяет состояние объекта, но не возвращает значение. А запросным называется метод, возвращающий значение, но не видоизменяющий объект.

Почему так важно соблюдать этот принцип? Для этого имеется целый ряд причин, главная из которых — взаимодействие. Если метод является запросным, то мы не должны анализировать его тело, чтобы выяснить, можем ли мы использовать его несколько раз подряд, не вызывая какой-нибудь нежелательный побочный эффект.

Вот как выглядит метод `performCommand` (выполнить команду) после целого ряда извлечений кода.

```
public class AccountDetailFrame extends Frame
    implements ActionListener, WindowListener
{
    public void performCommand(String source) {
        if (source.equals("project activity")) {
            setDescription(getDetailText() + " " + getProjectionText());
            ...
            String accountDescription = getAccountSymbol();
```

```
            accountDescription += ": ";
            ...
            display.setText(accountDescription);
            ...
        }
    }

    void setDescription(String description) {
        detailDisplay = new DetailFrame();
        detailDisplay.setDescription(description);
        detailDisplay.show();
    }

    String getAccountSymbol() {
        return detailDisplay.getAccountSymbol();
    }
    ...
}
```

А теперь, когда мы извлекли весь код, связанный с рамкой `detailDisplay`, мы можем перейти к извлечению кода доступа к компонентам класса `AccountDetailFrame` (рамка подробных данных счета).

```
public class AccountDetailFrame extends Frame
        implements ActionListener, WindowListener {
    public void performCommand(String source) {
        if (source.equals("project activity")) {
            setDescription(getDetailText() + " " + getProjectionText());
            ...
            String accountDescription
                = detailDisplay.getAccountSymbol();
            accountDescription += ": ";
            ...
            setDisplayText(accountDescription);
            ...
        }
    }

    void setDescription(String description) {
        detailDisplay = new DetailFrame();
        detailDisplay.setDescription(description);
        detailDisplay.show();
    }

    String getAccountSymbol() {
        return detailDisplay.getAccountSymbol();
    }

    void setDisplayText(String description) {
        display.setText(description);
    }
    ...
}
```

После этих извлечений мы можем выполнить *подклассификацию и переопределение метода*, а затем протестировать любой код, оставшийся в методе `performCommand`. Так, если мы выполним подклассификацию класса `AccountDetailFrame` подобным способом, то сумеем проверить, будет ли выведен правильный текст по команде "`project activity`" (активность проекта).

```
public class TestingAccountDetailFrame extends AccountDetailFrame
{
    String displayText = "";
    String accountSymbol = "";

    void setDescription(String description) {
    }

    String getAccountSymbol() {
        return accountSymbol;
    }

    void setDisplayText(String text) {
        displayText = text;
    }

}
```

Ниже приведен тест для испытания метода `performCommand`.

```
public void testPerformCommand() {
    TestingAccountDetailFrame frame = new TestingAccountDetailFrame();
    frame.accountSymbol = "SYM";
    frame.performCommand("project activity");
    assertEquals("SYM: basic account", frame.displayText);
}
```

Разделяя зависимости подобным очень строгим образом, т.е. выполняя реорганизацию кода автоматическим извлечением метода, мы можем получить в конечном итоге код, который способен немного обескуражить нас. Например, метод `setDescription` (задать описание), создающий рамку и отображающий ее, выглядит просто безобразно. А что произойдет, если мы вызовем его дважды? Нам придется как-то решать этот вопрос, но подобные грубые извлечения кода — это лишь первая стадия. А в дальнейшем у нас еще будет возможность выяснить, следует ли переместить код создания рамки в более подходящее место.

На какой же стадии мы теперь находимся? Мы начали с одного класса, содержавшего другой класс с одним очень важным методом: `performAction` (выполнить действие). А в конечном счете мы пришли к результату, приведенному на рис. 10.1

В методах `getAccountSymbol` (получить знак счета) и `setDescription` используется поле `detailDisplay` и больше ничего, хотя этого и не видно на блок-схеме, приведенной на рис. 10.1. А в методе `setDisplayText` (задать отображаемый текст) используется только поле `display` типа `TextField` (текстовое поле). И то, и другое свойство можно признать в качестве раздельной ответственности. В таком случае мы можем прийти к результату, приведенному на рис. 10.2.

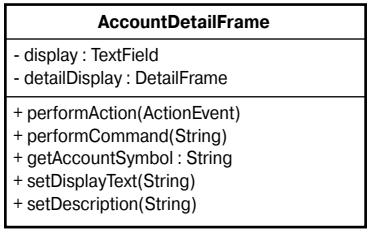


Рис. 10.1. Класс `AccountDetailFrame`

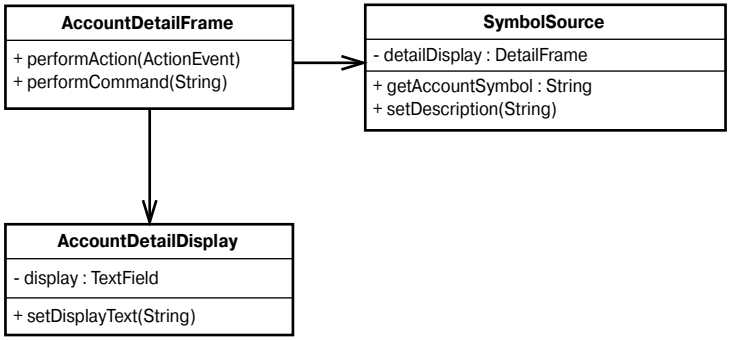


Рис. 10.2. Класс `AccountDetailFrame` *после грубой реорганизации кода*

Это довольно грубая реорганизация исходного кода, но, по крайней мере, она позволяет хоть как-то разделить ответственность. Класс `AccountDetailFrame` привязан к графическому пользовательскому интерфейсу (это подкласс класса `Frame`) и по-прежнему содержит бизнес-логику. При последующей реорганизации кода мы можем выйти за пределы такой логики, но теперь у нас есть, по крайней мере, возможность выполнить метод, содержащий бизнес-логику, в контрольном примере, а это уже шаг вперед.

Класс `SymbolSource` (источник знаков) является конкретным классом, обозначающим решение создать еще один объект класса `Frame` и получить от него полезную информацию. Но в рассматриваемом здесь примере этот класс получил название `SymbolSource`, потому что его основная функция с точки зрения класса `AccountDetailFrame` состоит в получении любой необходимой информации о знаке. И я не удивлюсь, если класс `SymbolSource` станет интерфейсом, когда данное решение изменится.

Шаги, предпринятые нами в данном примере, очень типичны. Если в нашем распоряжении имеется инструментальное средство реорганизации кода, то мы без особого труда извлечем методы из класса и затем приступим к разделению методов на группы, которые затем могут быть перенесены в новые классы. Хорошее инструментальное средство реорганизации кода только поможет нам автоматизировать реорганизацию кода путем извлечения метода, когда это можно сделать безопасно. Но самой опасной частью всей работы может оказаться промежуточное редактирование, выполняемое при последовательном применении данного инструментального средства. Не следует, однако, забывать, что для размещения тестов по местам вполне допускается извлечение методов с неудачными именами или структурой. Ведь самое главное — это безопасность. А после того как тесты окажутся на своих местах, код можно сделать намного более ясным.

Глава 11

Требуются изменения в коде, но неизвестно, какие методы следует тестировать

Допустим, что нам требуется внести изменения в код и написать *характеристические тесты*, чтобы точно определить уже имеющееся поведение. Где же их следует писать? На этот вопрос проще всего ответить следующим образом: тесты следует писать для каждого изменяемого метода. Но достаточно ли этого? Возможно, и достаточно, если код простой и понятный, но с унаследованным кодом ситуация совершенно иная. Изменение в одном месте может повлиять на поведение в каком-нибудь другом месте, и если у нас нет на месте подходящего теста, то мы даже не узнаем об этом.

Когда мне требуется внести изменения в особенно запутанный унаследованный код, то мне зачастую приходится тратить время на выявление места, где я мог бы написать свои тесты. С этой целью я осмысливаю изменения, которые мне предстоит сделать, пытаясь выяснить, на какие объекты они могут повлиять, а те, в свою очередь, — на другие объекты и т.д. В таком осмысливании нет ничего нового, ведь этим программисты занимались с самого зарождения вычислительной техники.

Программистам приходится осмысливать свои программы по целому ряду причин. Но любопытно, что они не особенно откровенничают по этому поводу. Они просто считают данный процесс неотъемлемой частью профессии программиста. К сожалению, это не очень нам помогает, когда мы сталкиваемся с особенно запутанным кодом, который ужасно трудно осмыслить. Мы понимаем, что должны реорганизовать код, чтобы сделать его более понятным, но тогда вновь встает вопрос тестирования кода. Если у нас нет подходящих тестов, то как мы узнаем, правильно ли мы реорганизовали код?

Способы, представленные в этой главе, призваны восполнить образовавшийся пробел. Нам нередко приходится осмысливать свои программы нетривиальным способом, чтобы обнаружить наиболее подходящие места для тестов.

Осмысление воздействий

Об этом обычно не принято много говорить, но всякое функциональное изменение в программном обеспечении связано с определенной цепочкой воздействий. Так, если изменить значение `3` на `4` в приведенном ниже коде C#, то изменится и результат выполнения вызываемого метода `getBalancePoint()` (получить точку равновесия). Кроме того, могут измениться и результаты выполнения методов, вызывающих данный метод, и так далее до определенной границы системы. Несмотря на это, поведение многих частей программы не меняется. Они не дают других результатов просто потому, что не вызывают метод `getBalancePoint()` — как непосредственно, так и косвенно.

```
int getBalancePoint() {
    const int SCALE_FACTOR = 3;
    int result = startingLoad + (LOAD_FACTOR * residual * SCALE_FACTOR);
    foreach(Load load in loads) {
        result += load.getPointWeight() * SCALE_FACTOR;
    }
    return result;
}
```

Поддержка анализа воздействий в интегрированной среде разработки

Иногда у меня возникает мысль, что было бы неплохо, если интегрированная среда разработки (IDE) помогала бы мне увидеть те или иные воздействия в связи с изменениями в унаследованном коде. Предварительно выделив фрагмент кода, я нажал бы оперативную клавишу и получил бы в ответ список всех переменных и методов, на которые могло бы оказать влияние изменение, внесенное мной в унаследованный код.

Возможно, когда-нибудь и будет разработано такое инструментальное средство, а до тех пор у нас имеются все основания для осмысления воздействий в связи с изменениям в коде без помощи подобных средств. Такому навыку несложно обучиться, но намного труднее узнать, насколько правильно он усвоен.

Самый лучший способ понять суть осмысления воздействий — обратиться к конкретному примеру. Ниже приведен класс Java из приложения, манипулирующего кодом C++. Похоже, что в данном примере довольно интенсивно используется предметная область, не так ли? Но знание предметной области не имеет особого значения, когда мы осмысливаем воздействия.

Попробуем выполнить небольшое упражнение, составив список всех объектов, которые можно изменить после создания объекта `CppClass` (класс C++), способного повлиять на результаты, возвращаемые любым из его методов.

```
public class CppClass {
    private String name;
    private List declarations;


    public CppClass(String name, List declarations) {
        this.name = name;
        this.declarations = declarations;
    }

    public int getDeclarationCount() {
        return declarations.size();
    }

    public String getName() {
        return name;
    }

    public Declaration getDeclaration(int index) {
        return ((Declaration)declarations.get(index));
    }
```

```
    public String getInterface(String interfaceName, int [] indices) {
        String result = "class " + interfaceName + " {\npublic:\n";
        for (int n = 0; n < indices.length; n++) {
            Declaration virtualFunction
                 = (Declaration)(declarations.get(indices[n]));
            result += "\t" + virtualFunction.asAbstract() + "\n";
        }
        result += "};\n";
        return result;
    }
}
```

Упомянутый выше список должен допускать следующее.

1. Кто-нибудь может ввести дополнительные элементы в список объявлений после его передачи конструктору. А поскольку этот список доступен по ссылке, изменения в нем способны повлиять на результаты, возвращаемые методами `getInterface` (получить интерфейс), `getDeclaration` (получить объявление) и `getDeclarationCount` (получить число объявлений).
2. Кто-нибудь может изменить один из объектов, содержащихся в списке объявлений, или же заменить один из его элементов, оказав тем самым воздействие на те же самые методы.

Глядя на метод `getName` (получить имя), кто-нибудь заподозрит, что этот метод может возвратить другое значение, если кто-то другой изменит строку `name`, но в Java объекты класса `String` постоянны. После создания таких объектов их значение не меняется. А после создания объекта `CppClass` метод `getName` всегда возвращает одно и то же строковое значение.

Мы можем набросать эскиз в виде упрощенной блок-схемы, показывающей, что изменения в списке объявлений (`declarations`) оказывают влияние на результат, возвращаемый методом `getDeclarationCount` (рис. 11.1).

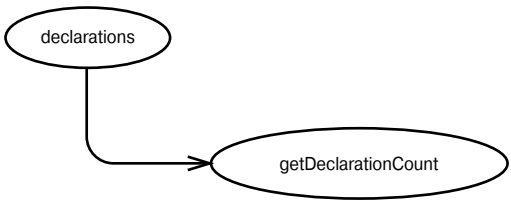


Рис. 11.1. Воздействие списка объявлений `declarations` на метод `getDeclarationCount`

Этот эскиз показывает, что если список объявлений `declarations` каким-то образом изменится, например увеличится, то метод `getDeclarationCount` может возвратить другое значение.

Аналогичный эскиз можно набросать и для вызова метода `getDeclaration(int index)`, как показано на рис. 11.2. Значения, возвращаемые при вызове метода `getDeclaration(int index)`, могут измениться, если по какой-либо причине произойдут изменения в списке объявлений `declarations` или же в самих объявлениях.

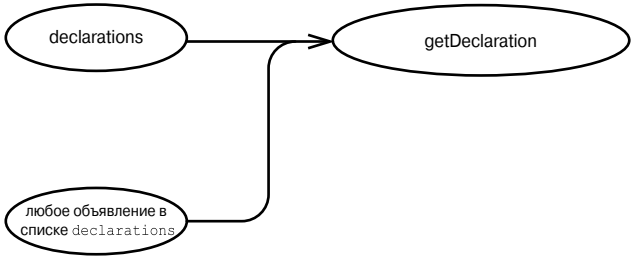


Рис. 11.2. Воздействие списка объявлений `declarations` *и содержащихся в нем объектов на метод* `getDeclarationCount`

На рис. 11.3 приведены аналогичные воздействия на метод `getInterface`. Мы можем объединить все эти эскизы в один общий эскиз, как показано на рис. 11.4.

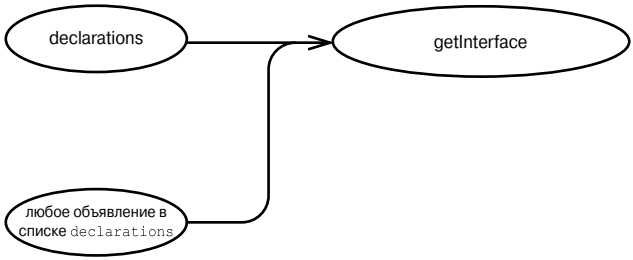


Рис. 11.3. Воздействия на метод `getInterface`

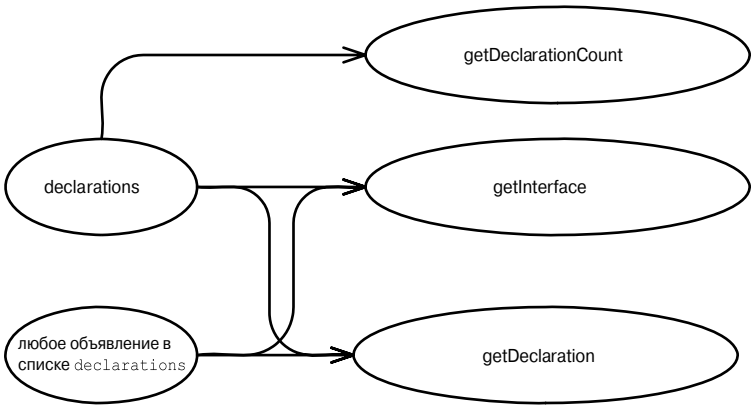


Рис. 11.4. Объединенный эскиз воздействий

Во всех этих блок-схемах не очень много синтаксиса, поэтому они называются *эскизами воздействий*. Они составляются по принципу заключения в отдельный кружок блок-схемы каждой переменной, на которую оказывается воздействие, и каждого метода, возвращаемое значение которого может измениться. В одних случаях переменные оказываются в одном и том же объекте, а в других случаях — в разных объектах. Но для составления подобных эскизов это не так важно: мы просто заключаем в отдельные кружки блок-схемы объекты, которые подлежат изменению, и соединяем их стрелками со всеми остальными

объектами, значения которых могут измениться во время выполнения под воздействием изменений в этих объектах.

Если ваш код хорошо структурирован, то у многих методов в вашей программе будут простые структуры воздействий. В действительности, одним из показателей добротности кода программы служит преставление довольно сложных воздействий на внешнее окружение в виде суммы намного более простых воздействий в коде. Практически все, что можно сделать для упрощения эскиза воздействий, составляемого для отдельного фрагмента кода, делает этот код более понятным и удобным для сопровождения.

А теперь расширим наше представление о системе, из которой происходит рассмотренный выше класс, чтобы представить картину воздействий в более крупном масштабе. Объекты `CppClass` создаются в классе под названием `ClassReader` (считыватель классов). В действительности, мы могли бы определить, что они создаются только в классе `ClassReader`.

```
public class ClassReader {
    private boolean inPublicSection = false;
    private CppClass parsedClass;
    private List declarations = new ArrayList();
    private Reader reader;

    public ClassReader(Reader reader) {
        this.reader = reader;
    }

    public void parse() throws Exception {
        TokenReader source = new TokenReader(reader);
        Token classToken = source.readToken();
        Token className = source.readToken();

        Token lbrace = source.readToken();
        matchBody(source);
        Token rbrace = source.readToken();

        Token semicolon = source.readToken();

        if (classToken.getType() == Token.CLASS
            && className.getType() == Token.IDENT
            && lbrace.getType() == Token.LBRACE
            && rbrace.getType() == Token.RBRACE
            && semicolon.getType() == Token.SEMIC) {
          parsedClass = new CppClass(className.getText(),
                  declarations);
        }
    }
    ...
}
```

Вспомните, что нам известно о классе `CppClass`? Знаем ли мы, что список объявлений вообще не изменяется после создания объекта класса `CppClass`? Представление самого класса `CppClass` ничего нам об этом не говорит. Поэтому нам нужно выяснить, каким

образом заполняется список объявлений. Проанализировав класс CppClass более тщательно, мы обнаружим, что объявления вводятся в этом классе только в одном месте, а именно: в методе matchVirtualDeclaration (согласовать виртуальное объявление), вызываемом методом matchBody (согласовать тело) при синтаксическом анализе.

```
private void matchVirtualDeclaration(TokenReader source)
            throws IOException {
    if (!source.peekToken().getType() == Token.VIRTUAL)
        return;
    List declarationTokens = new ArrayList();
    declarationTokens.add(source.readToken());
    while(source.peekToken().getType() != Token.SEMIC) {
        declarationTokens.add(source.readToken());
    }
    declarationTokens.add(source.readToken());
    if (inPublicSection)
        declarations.add(new Declaration(declarationTokens));
}
```

Похоже, что все изменения в данном списке происходят до создания объекта CppClass. А поскольку мы вводим новые объявления в список без каких-либо ссылок на них, то объявления вообще не изменяются.

А теперь подумаем об объектах, содержащихся в списке объявлений. Метод readToken (читать лексему) класса TokenReader (считыватель лексем) возвращает объекты лексем, которые просто содержат строку и целое, никогда не меняющееся число. Оно не показано в приведенном выше фрагменте кода, но беглый анализ класса Declaration показывает, что ничто другое не может изменить состояние данного класса после его создания, и поэтому мы можем с полной уверенностью сказать, что при создании объекта CppClass его список объявлений и содержимое этого списка не меняются.

А что это знание нам дает? Если бы мы получили неожиданные значения от объекта CppClass, то знали бы, что нам придется проанализировать лишь пару объектов. Как правило, в ходе такого анализа мы можем проследить обратный путь к тем местам, где были созданы подобъекты CppClass, и выяснить, что же происходит. Кроме того, мы можем сделать код более ясным, отметив некоторые ссылки в постоянной класса CppClass с помощью ключевого слова final, имеющегося в Java.

В программах, написанных не очень качественно, нам зачастую очень трудно выяснить, почему анализируемые нами результаты получаются именно такими, а не другими. На данном этапе у нас возникают трудности с отладкой, и поэтому нам приходится осмысливать происходящее в обратном направлении: от следствия к причине, т.е. от проблемы к ее источнику. Работая с унаследованным кодом, мы нередко задаем себе другой вопрос: если мы внесем конкретное изменение, то как оно может повлиять на остальные результаты выполнения программы?

Для ответа на этот вопрос требуется осмысление в прямом направлении от точек внесения изменений. Если вы как следует освоите такое осмысление, то тем самым овладеете основами способа поиска подходящих мест для написания тестов.

Осмысление в прямом направлении

В предыдущем примере мы попытались проследить (от следствия до причины) ряд объектов, оказывающих воздействие на значения в конкретной точке кода. Когда же мы пишем *характеристические тесты*, данный процесс происходит в обратном направлении: мы анализируем ряд объектов и пытаемся выяснить те изменения, которые могут произойти по нисходящей, если эти объекты перестанут действовать. Обратимся к следующему примеру. Ниже приведен класс из файловой системы, действующей в оперативной памяти. У нас нет никаких тестов для этого класса, но нам требуется внести в него некоторые изменения.

```
public class InMemoryDirectory {
    private List elements = new ArrayList();

    public void addElement(Element newElement) {
        elements.add(newElement);
    }

    public void generateIndex() {
        Element index = new Element("index");
        for (Iterator it = elements.iterator(); it.hasNext(); ) {
            Element current = (Element)it.next();
            index.addText(current.getName() + "\n");
        }
        addElement(index);
    }

    public int getElementCount() {
        return elements.size();
    }

    public Element getElement(String name) {
        for (Iterator it = elements.iterator(); it.hasNext(); ) {
            Element current = (Element)it.next();
            if (current.getName().equals(name)) {
                return current;
            }
        }
        return null;
    }
}
```

Класс `InMemoryDirectory` (каталог в оперативной памяти) представляет собой небольшой класс Java. Мы можем создать объект класса `InMemoryDirectory`, добавить в него элементы, сформировать их индекс, а затем получить к ним доступ. Текст в объектах `Elements` содержится как в файлах. При формировании индекса мы создаем элемент под названием `index` и присоединяем имена всех остальных элементов к его тексту.

Один из недостатков класса `InMemoryDirectory` заключается в том, что мы не можем вызвать метод `generateIndex` (формировать индекс) дважды, не испортив все дело. Ведь

если мы попытаемся вызвать метод `generateIndex` дважды, то получим два индексных элемента (второй из них будет фактически указывать на первый как на элемент каталога).

К счастью, класс `InMemoryDirectory` используется в данном приложении очень ограниченно. С его помощью создаются каталоги, которые заполняются элементами, а затем вызывается метод `generateIndex` и каталог передается другим частям приложения для доступа к его элементам. Все эти функции выполняются в настоящий момент исправно, но нам нужно внести в данное приложение изменения таким образом, чтобы его пользователи могли вводить элементы в любой момент в течение времени существования каталога.

В идеальном случае нам бы хотелось, чтобы создание и обновление индекса происходило как побочный эффект ввода элементов. Когда пользователь вводит элемент в первый раз, должен быть создан индексный элемент с именем введенного элемента. А во второй раз тот же самый индексный элемент должен обновиться именем введенного элемента. Написать тесты для проверки нового поведения и удовлетворяющего им кода несложно, но у нас нет тестов для проверки текущего поведения. Как же нам найти место для размещения своих тестов?

В рассматриваемом здесь примере ответ на этот вопрос совершенно ясен: нам требуется ряд тестов, в которых разными способами вызывается метод `addElement` (добавить элемент), формируется индекс, а затем выбираются различные элементы, чтобы проверить их правильность. Как же нам выбрать подходящие для этой цели методы? В данном случае сделать это будет нетрудно, поскольку тесты служат лишь описанием того, как мы предполагаем пользоваться каталогом. Мы могли бы написать их, даже не глядя на код каталога, потому что хорошо себе представляем, для чего предназначен каталог. К сожалению, выявить место в коде для тестирования не всегда оказывается так просто. Для того чтобы продемонстрировать это на практике, я мог бы привести для примера намного более сложный и крупный класс, подобный тем, что нередко встречаются в унаследованных системах, но анализ такого примера настолько бы вам надоел, что вы просто перестали бы читать эту книгу. Поэтому допустим, что рассматриваемый здесь пример отражает достаточно сложный случай и, глядя на код, попробуем осмыслить, что же нам нужно протестировать. Аналогичным образом осмысливаются и более трудные случаи изменения кода.

В рассматриваемом здесь примере нам необходимо, прежде всего, выявить место для внесения изменений. В частности, нам нужно исключить ряд функций из метода `generateIndex` и ввести ряд функций в метод `addElement`. Определив эти методы в качестве точек для внесения изменений, можно приступать к составлению эскизов воздействий.

Начнем с метода `generateIndex`. Откуда он вызывается? Ни один из других методов данного класса не вызывает его. Он вызывается только клиентами. Должны ли мы видоизменить что-нибудь в методе `generateIndex`? Мы создаем новый элемент и вводим его в каталог, и поэтому метод `generateIndex` может воздействовать на совокупность объектов `elements` в данном классе (рис. 11.5).

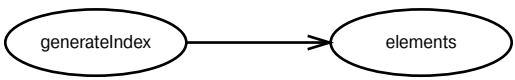


Рис. 11.5. Метод `generateIndex` воздействует на совокупность объектов `elements`

А теперь проанализируем совокупность объектов `elements` и посмотрим, на что она может оказывать влияние. Где еще она используется? По-видимому, она используется в методах `getElement` и `getElementCount` (получить число элементов). Кроме того, совокупность объектов `elements` используется в методе `addElement`, но мы можем и не

учитывать это, поскольку метод `addElement` ведет себя одинаково независимо от действий, которые мы выполняем над совокупностью объектов `elements`: что бы мы с ними ни делали, это никак не повлияет на пользователей метода `addElement` (рис. 11.6).

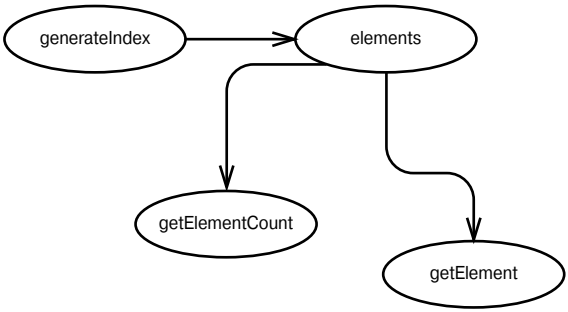


Рис. 11.6. Дополнительные влияния изменений в методе `generateIndex`

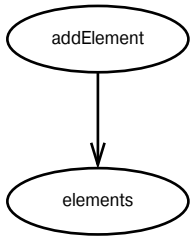


Рис. 11.7. Воздействие метода `addElement` *на совокупность объектов* `elements`

Можно ли считать процесс осмысления изменений в коде завершенным? Нет, нельзя. В качестве точек для внесения изменений в код мы выбрали методы `generateIndex` и `addElement`, и поэтому нам необходимо теперь проанализировать влияние метода `addElement` на совокупность объектов `elements` (рис. 11.7).

Мы, конечно, можем рассмотреть элементы, которые подвергаются такому воздействию, но ведь мы уже сделали это, анализируя воздействие метода `generateIndex` на совокупность объектов `elements`.

Теперь эскиз всех интересующих нас воздействий выглядит так, как показано на рис. 11.8.

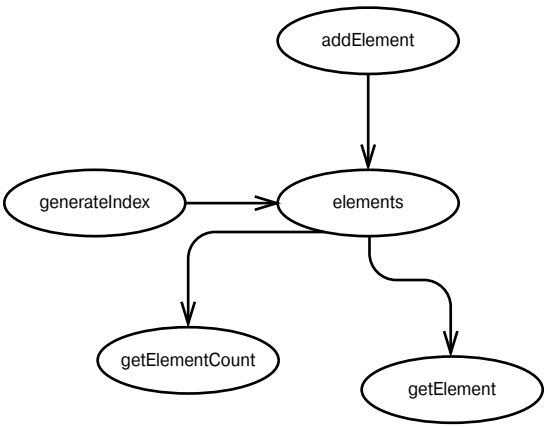


Рис. 11.8. Эскиз воздействий для класса `InMemoryDirectory`

Пользователи класса `InMemoryDirectory` могут ощутить на себе воздействия через методы `generateIndex` и `addElement`. Если мы сумеем написать тесты на месте этих методов, то тем самым покроем ими все воздействия в связи нашими изменениями кода.

Но не упустили ли мы что-нибудь? Как насчет суперклассов и подклассов? Если любые данные в классе `InMemoryDirectory` оказываются общедоступными, защищенными или же доступными в пределах области действия пакета, то метод из подкласса может использовать их неизвестными нам способами. В данном примере переменные экземпляров в классе `InMemoryDirectory` являются частными, и поэтому нам не следует об этом особенно беспокоиться.

При составлении эскизов воздействий непременно убедитесь в том, что вы обнаружили всех клиентов проверяемого вами класса. Если у этого класса имеется суперкласс или подклассы, то могут быть и другие, неучтенные вами клиенты.

Все ли мы учли? Пожалуй, мы совершенно не учли еще одно обстоятельство: мы пользуемся классом `Element` в каталоге, но он не включен в наш эскиз воздействий. Рассмотрим этот класс более внимательно.

Когда мы вызываем метод `generateIndex`, то создаем объект класса `Element` и повторно вызываем в нем метод `addText` (добавить текст). Проанализируем код класса `Element`.

```
public class Element {
    private String name;
    private String text = "";

    public Element(String name) {
        this.name = name;
   }

    public String getName() {
        return name;
    }

    public void addText(String newText) {
        text += newText;
    }

    public String getText() {
        return text;
    }
}
```

К счастью, код данного класса довольно прост. Обведем в эскизе воздействий кружком новый элемент, создаваемый методом `generateIndex`, как показано на рис. 11.9.

Как только новый элемент будет создан и заполнен текстом, метод `generateIndex` введет его в совокупность элементов, а следовательно, он окажет воздействие на эту совокупность (рис. 11.10).

Из проведенного анализа нам уже известно, что метод `addText` оказывает воздействие на совокупность объектов `elements`, которые, в свою очередь, влияют на значения, возвращаемые методами `getElement` и `getElementCount`. Если нам требуется проверить правильность формирования текста, то мы можем вызвать метод `getText` (получить текст) в элементе, возвращаемом методом `getElement`. Только в этих местах нам и требуется написать тесты, чтобы обнаружить влияние внесенных нами изменений.

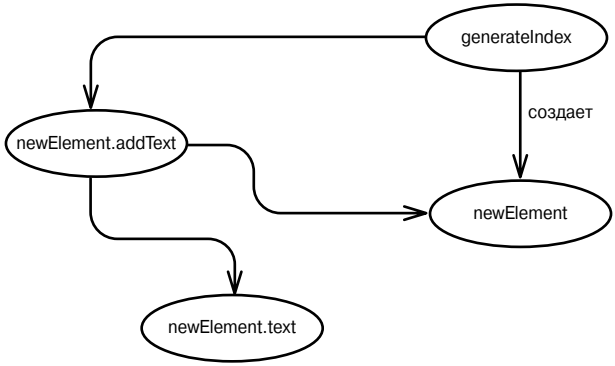


Рис. 11.9. Воздействия с помощью класса `Element`

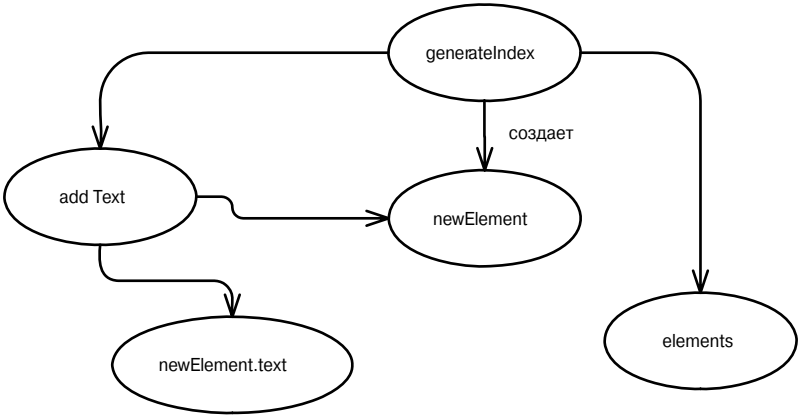


Рис. 11.10. Воздействие метода `generateIndex` *на совокупность объектов* `elements`

Как упоминалось ранее, рассматриваемый здесь пример довольно прост и невелик, хотя и весьма характерен для того осмысления, которое нам требуется для оценки влияния изменений в унаследованном коде. Нам нужно найти места для тестирования и, прежде всего, выяснить, где можно обнаружить влияние внесенных нам изменений и его характер. Если мы точно знаем, где именно можно обнаружить влияние этих изменений, то нам остается только аккуратно выбрать места для написания тестов.

Распространение воздействий

Одни пути распространения воздействий обнаружить проще, чем другие. В примере класса `InMemoryDirectory` из предыдущего раздела нам удалось в конечном итоге дойти до методов, возвращающих значения вызывающей части программы. Даже если я начинаю прослеживание воздействий с точек или мест внесения изменений, то, как правило, я замечаю сначала методы, возвращающие значения. Ведь они распространяют воздействия на вызывающий их код, если только возвращаемые ими значения не используются.

Воздействия могут также распространяться скрытно и незаметно. Так, если объект воспринимает другой объект в качестве параметра, то изменение его состояния отразится обратно на остальной части приложения.

В каждом языке программирования имеются правила обработки параметров, передаваемых методам. По умолчанию ссылки на объекты передаются, как правило, по значению. Это правило характерно для Java и C#. Вместо самих объектов методам в этом случае передаются дескрипторы объектов. В итоге любой метод может изменить состояние объектов через переданные ему дескрипторы этих объектов. В некоторых языках имеются ключевые слова, с помощью которых можно исключить видоизменение состояния объекта, передаваемого методу. Так, в C++ для этой цели служит ключевое слово `const`, указываемое в объявлении параметра конкретного метода.

Самый незаметный способ воздействия одного фрагмента кода на другой — через глобальные или статические данные. Рассмотрим следующий пример.

```
public class Element {
    private String name;
    private String text = "";

    public Element(String name) {
        this.name = name;
    }

    public String getName() {
    return name;
    }

    public void addText(String newText) {
        text += newText;
        View.getCurrentDisplay().addText(newText);
    }

    public String getText() {
        return text;
    }
}
```

Этот класс отличается от класса `Element`, упоминавшегося ранее в примере с классом `InMemoryDirectory`, лишь второй строкой кода в методе `addText`, выделенной полужирным в приведенном выше фрагменте кода. Анализируя сигнатуры методов из данного класса `Element`, мы вряд ли сможем обнаружить влияние отдельных элементов на представление данных. Конечно, скрывать информацию полезно, если только это не та информация, которая нам требуется.

Воздействия распространяются в коде тремя основными путями.

1. Возвращаемые значения, используемые в вызывающей части программы.
2. Видоизменение объектов, передаваемых в качестве параметров, используемых в дальнейшем.
3. Видоизменение статических или глобальных данных, используемых в дальнейшем.

В некоторых языках программирования доступны и другие пути распространения воздействий в коде. Например, программирующие на аспектно-ориентированных языках могут составлять конструкции, называемые аспектами и оказывающие воздействие на поведение кода в других частях системы.

Ниже приведена процедура эвристического анализа, которой я пользуюсь для обнаружения воздействий в коде.

1. Выявить метод, который подлежит изменению.
2. Если метод возвращает значение, то проанализировать в коде те места, откуда он вызывается.
3. Проверить, видоизменяет ли метод какие-либо значения. Если видоизменяет, то проанализировать методы, использующие эти значения, а также методы, вызывающие эти методы.
4. Непременно проверить суперклассы и подклассы, которые бывают пользователями переменных экземпляров и методов, выявленных в ходе данного анализа.
5. Проанализировать параметры, передаваемые методам, выявленным в ходе данного анализа. Проверить, используются ли эти параметры или любые объекты, возвращаемые их методами, в коде, который требуется изменить.
6. Найти глобальные переменные и статические данные, видоизменяемые в любом из методов, выявленных в ходе данного анализа.

Инструментальные средства для осмысления воздействий

Самым действенным средством, имеющимся в нашем арсенале, является наше знание языка, на котором мы программируем. В каждом языке имеются небольшие "брандмауэры", препятствующие распространению воздействий. Если они хорошо нам известны, то мы не упустим возможность воспользоваться ими.

Допустим, что нам требуется внести изменения в представление приведенного ниже класса системы координат. В частности, нам нужно перейти к использованию вектора для хранения значений координат `x` и `y`, поскольку мы хотели бы обобщить класс `Coordinate` до такой степени, чтобы он представлял координаты трехмерного и четырехмерного пространства. В приведенном ниже коде Java нам не нужно выходить за пределы класса, чтобы понять влияние подобного изменения.

```
public class Coordinate {
    private double x = 0;
    private double y = 0;

    public Coordinate() {}
    public Coordinate(double x, double y) {
        this.x = x; this.y = x;
    }
    public double distance(Coordinate other) {
        return Math.sqrt(
```

```
            Math.pow(other.x - x, 2.0) + Math.pow(other.y - y, 2.0));
    }
}
```

Ниже приведен код, на пределы которого нам все же придется заглянуть.

```
public class Coordinate {
    double x = 0;
    double y = 0;

    public Coordinate() {}
    public Coordinate(double x, double y) {
        this.x = x; this.y = x;
    }
    public double distance(Coordinate other) {
        return Math.sqrt(
            Math.pow(other.x - x, 2.0) + Math.pow(other.y - y, 2.0));
    }
}
```

Как видите, разница между двумя приведенными выше фрагментами кода невелика. В первом варианте класса `Coordinate` переменные `x` и `y` являются частными, а во втором его варианте они доступны в области действия пакета. Если мы внесем любые изменения в переменные `x` и `y` в первом варианте, то такие изменения окажут влияние только на клиентов данного класса через функцию `distance` (расстояние) независимо от того, пользуются ли клиенты классом `Coordinate` или же его подклассом. А во втором варианте клиенты из пакета могут получать непосредственный доступ к переменным данного класса. Поэтому мы должны выяснить данное обстоятельство или попытаться сделать эти переменные частными, исключив прямой доступ к ним. Кроме того, переменные экземпляров можно использовать в подклассах класса `Coordinate`, поэтому мы должны выявить их и проверить, используются ли они в методах каких-либо подклассов.

Знать досконально язык, на котором мы программируем, очень важно, поскольку незначительные, на первый взгляд, правила могут сбить нас с толку. Обратимся к следующему примеру на языке C++.

```
class PolarCoordinate : public Coordinate {
public:
           PolarCoordinate();
    double getRho() const;
    double getTheta() const;
};
```

Если в C++ ключевое слово `const` следует после объявления метода, то этот метод не может видоизменить переменные экземпляров объекта. Или все-таки может? Допустим, что суперкласс приведенного выше класса `PolarCoordinate` выглядит следующим образом:

```
class Coordinate {
protected:
    mutable double first, second;
};
```

Если в C++ используется ключевое слово `mutable` в объявлении переменных, то это означает, что такие переменные можно видоизменить в методах типа `const`. Конечно,

такое применение ключевого слова `mutable` кажется из ряда вон выходящим, но когда требуется выяснить, что можно и чего нельзя изменить в программе, которую мы не знаем досконально, нам приходится искать любые воздействия, как бы странно они ни проявлялись. Так, воздействие ключевого слова `const` в коде C++ нельзя принимать на веру без проверки. Это же относится и к тем конструкциям в других языках программирования, которые способны ввести в заблуждение.

Досконально овладейте языком, на котором вы программируете.

Что дает анализ воздействий

Старайтесь анализировать воздействия в коде при всяком удобном случае. Хорошо изучив базу кода, вы иногда замечаете, что вам уже не нужно искать некоторые элементы кода. В таком случае вы начинаете ощущать некую "элементарную доброкачественность" базы кода. В лучших образцах кода сбойных мест оказывается не так уж и много. Благодаря некоторым "правилам", воплощенным в базе кода явным или неявным образом, вам не нужно искать возможные воздействия до умопомрачения. Для того чтобы обнаружить такие правила, достаточно представить себе, как одна часть программы может воздействовать на другую, причем так, как этого еще не наблюдалось в базе кода, а затем признаться себе: "Да нет, это же нелепо". Чем больше таких правил обнаруживается в базе кода, тем легче с ней работать. В неудачном коде очень трудно выявить какие-либо "правила" или же из них имеются сплошные исключения.

"Правила" для базы кода необязательно являются основными постулатами программирования, как, например, "никогда не пользуйтесь защищенными переменными". Напротив, они часто зависят от контекста. В примере класса `CppClass`, приведенном в самом начале этой главы, мы выполнили небольшое упражнение, в котором мы попытались выяснить, что именно может оказать воздействие на пользователей объекта класса `CppClass` после его создания. Ниже приведен фрагмент соответствующего кода.

```
public class CppClass {
    private String name;
    private List declarations;

    public CppClass(String name, List declarations) {
        this.name = name;
        this.declarations = declarations;
    }
    ...
}
```

Как мы уже отмечали, кто-нибудь может видоизменить список объявлений после его передачи конструктору данного класса. Такая возможность как нельзя лучше подходит под упоминавшее выше правило "Да нет, это же нелепо". Если, приступая к анализу класса `CppClass`, мы знаем, что в наше распоряжение предоставляется список, который нельзя изменить, то нам будет намного легче осмысливать воздействия.

В целом, программировать становится легче по мере сужения воздействий в программе. Ведь для того чтобы понять фрагмент кода, нам требуется меньше знать о нем. В крайнем случае мы можем перейти к функциональному программированию на таких языках,

как Scheme и Haskell. Программы, написанные на таких языках, довольно легко понять, хотя эти языки программирования не очень широко распространены. Несмотря на это, ограничивая воздействия в языках ООП, мы в состоянии намного упростить тестирование кода, и для этого не существует никаких препятствий.

Упрощение эскизов воздействий

Эта книга посвящена вопросам упрощения работы с унаследованным кодом, поэтому и не удивительно, что пользы от многих приведенных в ней примеров столько же, сколько и от пролитого молока. В связи с этим мне хотелось бы воспользоваться возможностью продемонстрировать нечто более полезное на примере эскизов воздействий. Возможно, данный пример повлияет на ваш стиль написания кода по ходу работы с ним.

Помните эскиз воздействий для класса `CppClass` (рис. 11.11)?

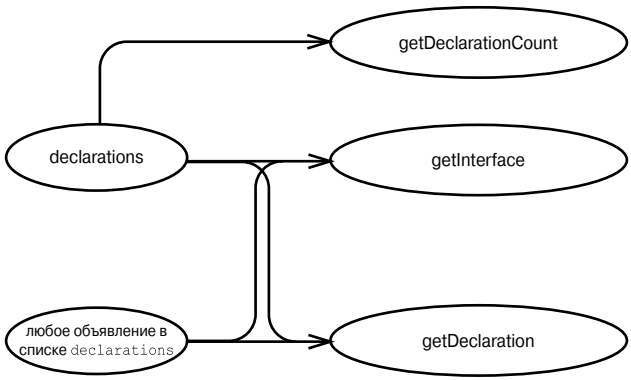


Рис. 11.11. Эскиз воздействий для класса `CppClass`

Похоже, что воздействия в этом эскизе несколько разветвляются. Два фрагмента данных (объявление и список `declarations`) оказывают воздействие на разные методы. Для тестов мы можем выбрать любой подходящий метод. В данном случае лучше всего выбрать метод `getInterface`, поскольку объявления в нем проявляются в несколько большей степени. То, что мы можем распознать методом `getInterface`, не так-то просто распознать методом `getDeclarationCount`. Я не стал бы писать тесты только для метода `getInterface`, если бы охарактеризовывал класс `CppClass`, но было бы непростительно не охватить тестами методы `getDeclaration` и `getDeclarationCount`. Но, что если метод `getInterface` выглядел бы следующим образом:

```
public String getInterface(String interfaceName, int [] indices) {
    String result = "class " + interfaceName + " {\npublic:\n";
    for (int n = 0; n < indices.length; n++) {
        Declaration virtualFunction = getDeclaration(indices[n]);
        result += "\t" + virtualFunction.asAbstract() + "\n";
    }
    result += "};\n";
    return result;
}
```

Отличия в приведенном выше коде незначительны. Теперь метод `getDeclaration` используется в этом коде внутренним образом. Следовательно, эскиз воздействий, приведенный на рис. 11.12, претерпевает изменения, как показано на рис. 11.13.

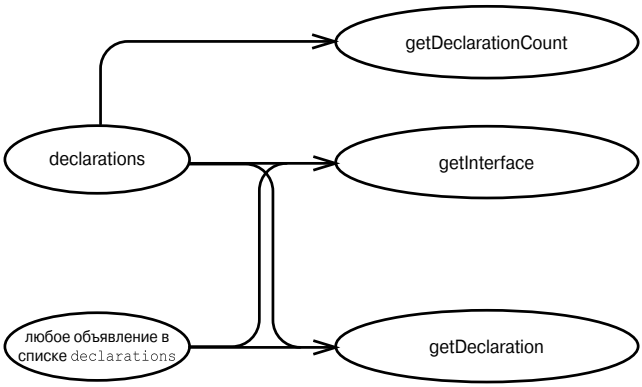


Рис. 11.12. Эскиз воздействий для исходного класса `CppClass`

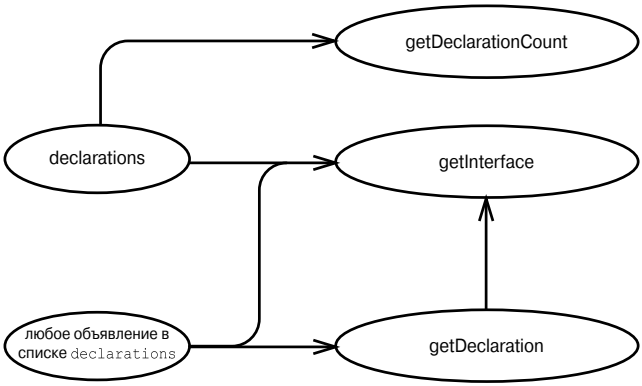


Рис. 11.13. Эскиз воздействий для измененного класса `CppClass`

Столь незначительное, на первый взгляд, изменение имеет весьма существенные последствия. Ведь теперь метод `getDeclaration` используется в методе `getInterface` внутренним образом. А в конечном итоге мы испытываем метод `getDeclaration` всякий раз, когда тестируем метод `getInterface`.

Когда мы удаляем мелкие фрагменты дублированного кода, то зачастую получаем эскизы воздействий с меньшим количеством конечных точек. А это, в свою очередь, упрощает тестирование.

Воздействия и инкапсуляция

Одним из часто упоминаемых преимуществ ООП является инкапсуляция. Всякий раз, когда я демонстрирую разработчикам способы разрыва зависимостей, описанные в этой книге, они, как правило, указывают мне на нарушение инкапсуляции в этих способах. Для многих из них это действительно характерно.

Соблюдать инкапсуляцию очень важно, но *еще важнее* причина, по которой это нужно делать. Ведь инкапсуляция помогает нам осмыслить код. В хорошо инкапсулированном коде оказывается меньше путей для прослеживания кода при попытке понять его. Так, если ввести в конструктор еще один параметр для разрывания зависимости, как это делается при реорганизации кода типа *параметризации конструктора*, то у нас появляется еще один путь для прослеживания кода, когда мы осмысливаем его воздействия. Нарушение инкапсуляции иногда усложняется осмысление кода, но в то же время оно может его упростить, если мы создадим впоследствии хорошо объясняющие код тесты. Имея в своем распоряжении контрольные примеры для тестирования класса, мы можем использовать их для осмысления кода более непосредственно. Кроме того, мы можем написать новые тесты для разрешения любых вопросов, возникающих у нас относительно поведения кода.

Инкапсуляция и покрытие тестами не всегда противоречат один другому, но когда это все же случается, я лично склоняюсь к покрытию тестами. Зачастую это помогает мне в дальнейшем добиться еще большей инкапсуляции.

Инкапсуляция — это не самоцель, а средство для понимания кода.

Когда нам нужно выявить место для написания тестов, мы не должны забывать о том, на что именно могут повлиять вносимые нами изменения. Поэтому мы должны непременно осмыслить связанные с этим воздействия. Сделать это мы можем неформально или же более строго с помощью небольших эскизов, но в любом случае подобное осмысление требует практики. В работе с особенно запутанным кодом это единственный навык, на который мы можем положиться, размещая тесты по местам.

Глава 12

На одном участке требуется внести много изменений, но следует ли разрывать зависимости со всеми классами, имеющими к этому отношение

В некоторых случаях оказывается проще начать с написания тестов для класса. Но в унаследованном коде это зачастую сделать трудно, поскольку очень нелегко разорвать зависимости. Если придерживаться правила вводить изменяемые классы в средства тестирования ради облегчения труда, то самым неприятным препятствием на этом пути может стать изменение, сосредоточенное на одном участке. В систему требуется ввести новое свойство, но оказывается, что для этого придется видоизменить три или четыре тесно связанных класса. И для подготовки каждого из них к тестированию потребуется почти два часа рабочего времени. Конечно, код после это должен стать лучше, но следует ли разрывать все зависимости по отдельности? Возможно, и не следует.

Зачастую код имеет смысл тестировать "на один уровень ниже", чтобы найти место для написания тестов сразу для нескольких изменений. Тесты можно писать в одном общедоступном методе для проверки изменений в целом ряде частных методов или же в интерфейсе одного объекта для проверки взаимодействия нескольких объектов, которые он содержит. Сделав это, мы можем не только проверить внесенные нами изменения, но и создать небольшой "задел" для дополнительной реорганизации кода на данном участке. Структура тестируемого кода может радикально измениться при условии, что тесты точно определяют его поведение.

Тесты более высокого уровня могут оказаться полезными при реорганизации кода. Им нередко отдают предпочтение ради мелкоструктурной проверки каждого класса, поскольку считается, что сделать изменение труднее, если для изменяемого интерфейса написано много мелких тестов. В действительности, вносить изменения нередко оказывается проще, чем ожидалось, поскольку они вносятся сначала в тестах, а затем в коде, постепенно улучшая структуру кода мелкими безопасными приращениями.

Несмотря на всю важность тестов более высокого уровня, они не могут заменить собой блочные тесты. Напротив, они должны стать первым шагом на пути к размещению блочных тестов по местам.

Как же разместить такие "покрывающие" тесты по местам? Прежде всего мы должны выявить место для их написания. Подробнее об этом речь шла в главе 11, где описаны *эскизы воздействий* — эффективное средство выявления места для написания тестов. А в этой главе описывается понятие *точки пересечения* и показывается, как такие точки обнаружи-

ваются. Кроме того, в этой главе описываются самые лучшие точки пересечения, которые только можно найти в коде, — так называемые *точки сужения*, а также показывается, как они обнаруживаются и чем они помогают при написании тестов для покрытия кода, подлежащего изменению.

Точки пересечения

Точка пересечения — это такая точка в программе, где можно обнаружить воздействия в связи с конкретным изменением. В одних приложениях найти подобные точки труднее, чем в других. Так, если отдельные части приложения связаны не самым естественным образом, то найти в нем подходящую точку пересечения будет нелегко. Для этого зачастую требуется осмысление воздействий и разрыв многих зависимостей. Так с чего начинать?

Начинать лучше всего с выявления тех мест, где требуется внести изменения, постепенно прослеживая распространение воздействий наружу из этих точек внесения изменений. Каждая точка, в которой можно обнаружить воздействия, оказывается точкой пересечения, хотя она может оказаться и не самой лучшей точкой пересечения. Поэтому в ходе данного процесса приходится руководствоваться здравым смыслом.

Простой пример

Допустим, что нам требуется видоизменить класс Java под названием `Invoice` (счет-фактура), чтобы изменить порядок расчета затрат. Все затраты рассчитываются в методе `getValue` (получить стоимость) класса `Invoice`.

```
public class Invoice
{
    ...
    public Money getValue() {
        Money total = itemsSum();
        if (billingDate.after(Date.yearEnd(openingDate))) {
            if (originator.getState().equals("FL") ||
                  originator.getState().equals("NY"))
                total.add(getLocalShipping());
            else
                total.add(getDefaultShipping());
        }
        else
             total.add(getSpanningShipping());
        total.add(getTax());
        return total;
    }
    ...
}
```

Нам требуется изменить порядок расчета затрат на транспортировку в Нью-Йорк. Законодательный орган этого штата ввел налог на транспортные операции, в связи с чем затраты приходится относить на счет потребителя. В ходе данного процесса мы извлекаем логику расчета затрат на транспортировку в новый класс под названием `ShippingPricer`

(оценщик затрат на транспортировку). По завершении код должен выглядеть следующим образом:

```
public class Invoice
{
    public Money getValue() {
        Money total = itemsSum();
        total.add(shippingPricer.getPrice());
        total.add(getTax());
        return total;
    }
}
```

Все эти операции, выполнявшиеся раньше в методе `getValue`, теперь выполняются в классе `ShippingPricer`. Нам придется изменить также конструктор класса `Invoice`, чтобы создать объект класса `ShippingPricer`, которому известно о датах выписки счетов-фактур.

Для обнаружения точек пересечения нам придется начать прослеживание воздействий в прямом направлении от точек внесения изменений. Выполнение метода `getValue` будет давать другой результат. Оказывается, что метод `getValue` не используется ни в одном из методов класса `Invoice`. Он используется в методе `makeStatement` (составить отчет) другого класса под названием `BillingStatement` (отчет о выписке счетов), как показано на рис. 12.1.

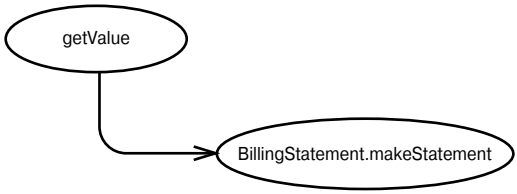


Рис. 12.1. Воздействие метода `getValue` на метод `BillingStatement.makeStatement`

Нам предстоит также видоизменить конструктор, и поэтому мы должны проанализировать код, который от этого зависит. В данном случае мы собираемся создать в конструкторе новый объект `ShippingPricer`. Этот объект оценщика будет оказывать воздействие только на методы, в которых он используется, а среди них он используется он только в методе `getValue`. Это воздействие показано на рис. 12.2.

Приведенные выше эскизы мы можем объединить так, как показано на рис. 12.3.

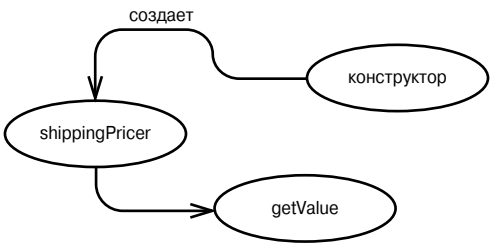


Рис. 12.2. Воздействия на метод `getValue`

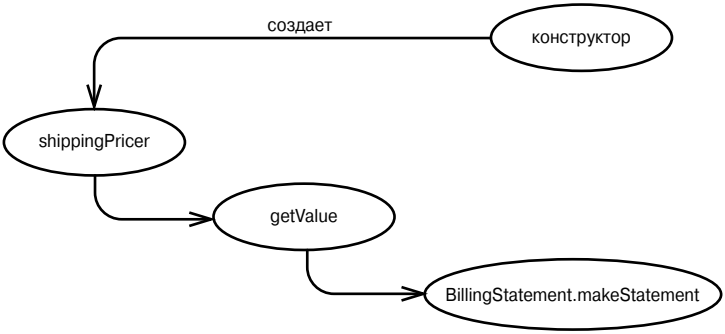


Рис. 12.3. Цепочка воздействий

Где же находятся искомые точки пересечения? В действительности мы можем воспользоваться любым из кружков на блок-схеме, приведенной на рис. 12.3, в качестве точки пересечения при условии, что у нас имеется доступ к тому, что они представляют. Мы могли бы, например, протестировать переменную `shippingPricer`, но поскольку это частная переменная в классе `Invoice`, у нас нет к ней доступа. Даже если бы переменная `shippingPricer` была доступна для тестирования, она оказалась бы довольно узкой точкой пересечения. Мы можем распознать результат изменений в конструкторе (создать объект `shippingPricer`) и убедиться в том, что этот объект делает именно то, что он и должен делать, но мы не можем воспользоваться им, чтобы не изменить метод `getValue` в худшую сторону.

Мы могли бы написать тесты, испытывающие метод `makeStatement` класса `BillingStatement` и проверяющие возвращаемое им значение, чтобы убедиться в правильности внесенных нами изменений. Но все же лучше написать тесты, испытывающие метод `getValue` в классе `Invoice` и проверяющие возвращаемое им значение, поскольку это менее трудоемкая задача. Конечно, было бы неплохо подвергнуть тестированию класс `BillingStatement`, но в настоящий момент в этом нет никакой необходимости. Если бы в дальнейшем нам потребовалось внести изменения в класс `BillingStatement`, то мы могли бы протестировать его.

Как правило, точки пересечения целесообразно выбирать рядом с точками внесения изменений в код по двум причинам. Первой из них является безопасность. Каждый шаг, разделяющий точку внесения изменений и точку пересечения, подобен шагу в логическом аргументе. По сути, мы заявляем следующее: "Тестирование в данной точке возможно, поскольку этот объект оказывает влияние на тот, а тот — на другой объект, оказывающий, в свою очередь, влияние на тестируемый объект". Чем больше шагов в таком аргументе, тем труднее убедиться в его правоте. Иногда единственным надежным способом оказывается написание тестов в точке пересечения и возврат к точке внесения изменений, чтобы немного изменить код и проверить, проходит или же не проходит тест. К такому способу можно прибегнуть лишь иногда, но не пользоваться им постоянно. Отдаленность точек пересечения вредна еще и по той причине, что для них обычно труднее подготовить тесты. Хотя это правило подтверждается не всегда и зависит от конкретного кода. Главное затруднение, опять же, заключается в числе шагов, разделяющих точку внесения изменений и точку пересечения. Для того чтобы убедиться в том, что тест покрывает некоторый отдаленный фрагмент функции кода, нередко приходится сильно "напрягать мозги".

В приведенном выше примере изменения, которые нам требуются в классе `Invoice`, вероятно, лучше всего протестировать именно там. Мы можем создать объект класса `Invoice` в средствах тестирования, настроить его разными способами и вызвать метод `getValue`, чтобы точно определить его поведение по ходу внесения изменений.

Точки пересечения более высокого уровня

Как правило, лучшей точкой пересечения для внесения изменений служит общедоступный метод в изменяемом классе. Такие точки пересечения несложно обнаружить и использовать, но иногда они оказываются не самым лучшим вариантом выбора. Для того чтобы проиллюстрировать это положение, расширим немного пример класса `Invoice`.

Допустим, что, помимо изменения порядка расчета транспортных расходов для объектов класса `Invoice`, нам требуется видоизменить класс под названием `Item` (товар), чтобы он содержал поле для хранения наименования грузоотправителя. Кроме того, в структуре класса `BillingStatement` требуется разделение по грузоотправителям. Соответствующая блок-схема структуры кода на унифицированном языке моделирования (UML) приведена на рис. 12.4.

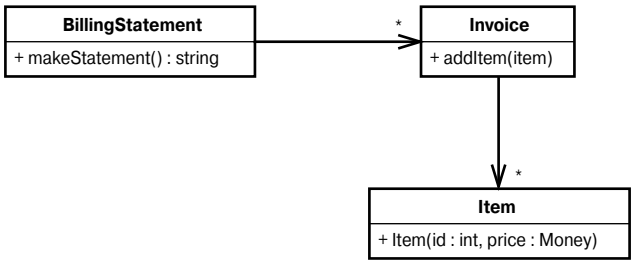


Рис. 12.4. Расширенный пример системы выписки счетов

Если ни для одного из этих классов нет соответствующих тестов, то мы могли бы начать с написания тестов для каждого класса в отдельности и внести требуемые изменения в код. Такой подход вполне работоспособен, но намного эффективнее было бы попытаться найти точку пересечения более высокого уровня, чтобы охарактеризовать в ней данный участок кода. У такого подхода имеется двойное преимущество — нам пришлось бы разрывать меньше зависимостей и в то же время мы могли бы зажать в программные тиски более крупный фрагмент кода. При наличии тестов, характеризующих данную группу классов, у нас имеется больший задел для реорганизации кода. В частности, мы можем изменить структуру классов `Invoice` и `Item`, используя в качестве инварианта тесты, имеющиеся для класса `BillingStatement`. Ниже приведен подходящий пример теста, который можно для начала использовать, чтобы охарактеризовать сообща классы `BillingStatement`, `Invoice` и `Item`.

```
void testSimpleStatement() {
    Invoice invoice = new Invoice();
    invoice.addItem(new Item(0,new Money(10));
    BillingStatement statement = new BillingStatement();
    statement.addInvoice(invoice);
    assertEquals("", statement.makeStatement());
}
```

Такой тест позволяет нам выяснить, какая именно стоимость получается в классе `BillingStatement` для счета-фактуры с одной позицией, а затем изменить тест, чтобы использовать эту стоимость. После этого мы могли бы ввести дополнительные тесты, чтобы выяснить, каким образом происходит форматирование отчета о выписке счетов для различных сочетаний счетов-фактур и их позиций. При написании контрольных примеров для испытания тех участков кода, где предполагается ввести швы, мы должны соблюдать особую осторожность.

Почему класс `BillingStatement` оказывается в данном случае идеальной точкой пересечения? Это единственная точка, которой мы можем воспользоваться, чтобы обнаружить воздействия после изменений в группе классов. На рис. 12.5 представлен эскиз воздействий для тех изменений, которые мы собираемся внести в код.

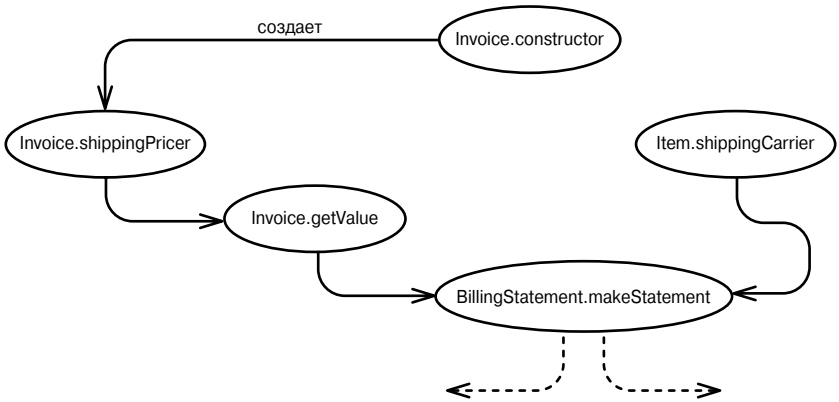


Рис. 12.5. Эскиз воздействий для примера системы выписки счетов

Обратите внимание на то, что все воздействия обнаруживаются через метод `makeStatement`. Это не всегда легко сделать, но именно в данном месте все эти воздействия можно, по крайней мере, обнаружить. Такое место в структуре кода называется *точкой сужения*. В этой точке происходит сужение в *эскизе воздействий*, т.е. она представляет собой такое место, где имеется возможность написать тесты, чтобы покрыть ими обширный ряд изменений. Если в структуре кода удается обнаружить точку сужения, то работа с таким кодом намного упрощается.

Но в отношении точек сужения следует помнить следующее: они определяются точками внесения изменений в код. Ряд изменений в классе может послужить подходящим местом для точки сужения, даже если у такого класса имеется много клиентов. Для того чтобы проиллюстрировать это положение, расширим наше представление о системе выписки счетов, приведенной на рис. 12.6.

В классе `Item` имеется не упоминавшийся ранее метод `needsReorder` (требуется повторный заказ). Класс `InventoryControl` (учет товарных запасов) вызывает этот метод всякий раз, когда требуется выяснить, следует ли разместить заказ. Меняет ли это наш эскиз воздействий для тех изменений, которые нам требуется внести? Нисколько не меняет. Добавление поля `shippingCarrier` (грузоотправитель) в класс `Item` вообще не оказывает никакого влияния на метод `needsReorder`, и поэтому класс `BillingStatement` по-прежнему остается для нас точкой сужения, т.е. тем местом сужения, где мы можем написать тест.

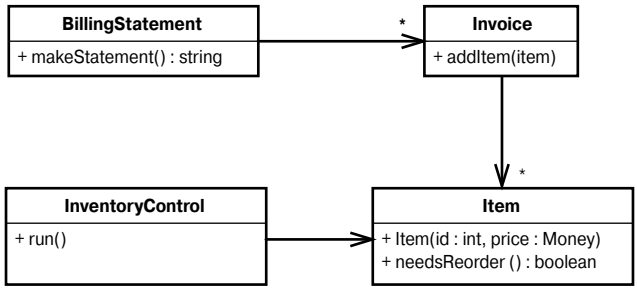


Рис. 12.6. Система выписки счетов с учетом товарных запасов

А теперь рассмотрим несколько иную ситуацию. Допустим, что нам нужно внести еще одно изменение в код, а именно: добавить методы в класс `Item`, чтобы задавать и получать поставщика товара из класса `Item`. Наименование поставщика будет использоваться в классах `InventoryControl` и `BillingStatement`. На рис. 12.7 показано, как это изменение отражается в эскизе воздействий.

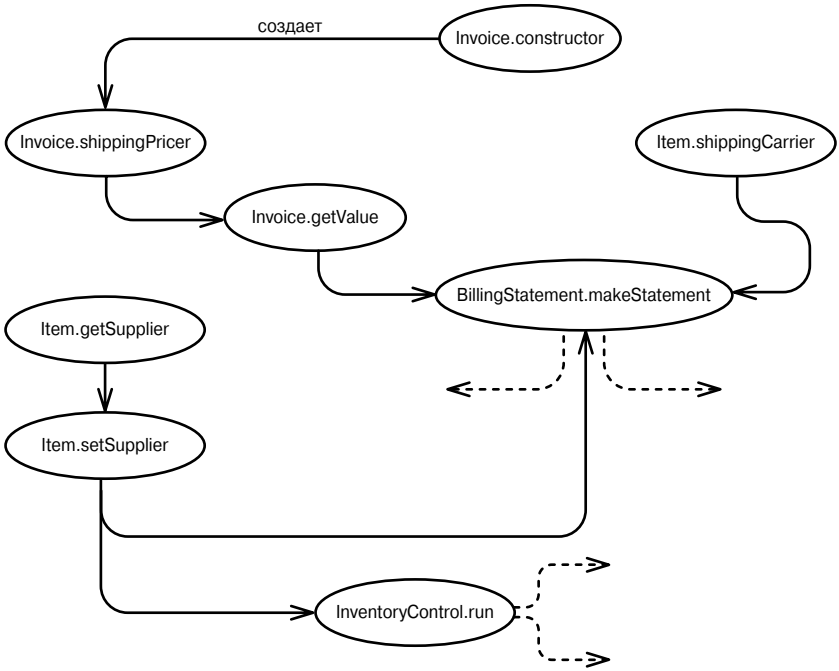


Рис. 12.7. Полный эскиз воздействий для примера системы выписки счетов

Похоже, что ситуация несколько усложнилась. Воздействия в результате наших изменений могут быть обнаружены через метод `makeStatement` класса `BillingStatement` и переменные, на которые воздействует метод `run` класса `InventoryControl`, но теперь единой точки пересечения уже не существует. Тем не менее методы `run` и `makeStatement` вместе взятые можно рассматривать как точку сужения. Сообща они образуют место большего сужения для обнаружения возможных осложнений, чем восемь методов и пере-

менных, которых могут коснуться вносимые в код изменения. Если мы разместим тесты именно в этом месте, то покроем ими значительную часть изменений в коде.

Точка сужения

Точка сужения — это место сужения в эскизе воздействий, где, тестируя пару методов, можно обнаружить изменения во многих методах.

В некоторых программах точки сужения для ряда изменений обнаруживаются очень легко, но зачастую сделать это едва ли вообще возможно. У одного класса или метода могут быть десятки объектов, на которые он способен оказывать непосредственное воздействие, а набросанный начиная с него эскиз воздействий выглядит как крупное дерево с довольно запутанными ветвями. Что же тогда делать? Мы можем, в частности, вернуться к точкам внесения изменений и попытаться вносить их не все сразу, а по очереди, обнаруживая точки сужения только для одного или, в крайнем случае, двух изменений, вносимых одновременно. Если же обнаружить точку сужения вообще не удастся, то можно попытаться написать тесты для отдельных изменений, расположив их как можно ближе.

Еще один способ обнаружения точки сужения состоит в том, чтобы найти нечто общее в использовании части кода, анализируемой по *эскизу воздействий*. Так, у метода или переменной могут быть три пользователя, но это совсем не означает, что он используется тремя разными способами. Допустим, что нам требуется немного реорганизовать метод `needsReorder` класса `Item` из приведенного выше примера. Соответствующий код здесь не приводится, но если набросать эскиз воздействий, то по нему можно обнаружить лишь точку сужения, включающую в себя метод `run` класса `InventoryControl` и метод `makeStatement` класса `BillingStatement`, а место еще большего сужения нам вряд ли удастся отыскать. Будет ли этого достаточно для написания тестов в одном из упомянутых выше классов, но не в другом? В связи с этим возникает другой, главный вопрос: если разделить метод, то удастся ли распознать такое разделение в данном месте? Ответ на этот вопрос зависит от того, как данный метод используется. Если он используется одинаковым образом в объектах со сравнимыми значениями, то тестирования в одном месте, но не в другом, может оказаться достаточно. В этой связи рекомендуется проделать тщательный анализ ситуации вместе с коллегой по работе.

Оценка структуры кода по точкам сужения

В предыдущем разделе речь шла о том, насколько полезными для тестирования могут быть точки сужения, но им можно найти и другое применение. Внимательное изучение мест расположения точек сужения дает некоторое представление о возможных путях улучшения кода.

Что на самом деле представляет собой точка сужения? Она представляет собой естественную границу инкапсуляции. Вместе с точкой сужения мы обнаруживаем узкий канал для распространения всех воздействий крупного фрагмента кода. Так, если метод `BillingStatement.makeStatement` представляет собой точку сужения для совокупности счетов-фактур и их позиций, то мы уже знаем, где искать причину, если отчет о выписке счетов окажется не таким, как предполагалось. Эта причина, скорее всего, связана с классом `BillingStatement` или же со счетами-фактурами и их позициями. С другой стороны, для вызова метода `makeStatement` нам не нужно ничего знать о самих счетах-фактурах и их позициях. Это очень похоже на определение инкапсуляции: нас не должно

интересовать внутреннее устройство, но если все же оно нас заинтересует, то для его понимания нам не обязательно рассматривать внешнее устройство. Когда мне приходится искать точки сужения, то зачастую я начинаю замечать, каким образом можно было бы перераспределить виды ответственности среди классов, чтобы добиться лучшей инкапсуляции.

Обнаружение скрытых классов по эскизам воздействий

Если приходится иметь дело с крупным классом, то иногда для выявления способа его разделения на отдельные части полезными оказываются эскизы воздействий. Рассмотрим небольшой пример кода Java. Допустим, что у нас имеется класс под названием `Parser` (синтаксический анализатор), содержащий метод `parseExpression` (выполнить синтаксический анализ выражения).

```
public class Parser
{
    private Node root;
    private int currentPosition;
    private String stringToParse;
    public void parseExpression(String expression) { .. }
    private Token getToken() { .. }
    private boolean hasMoreTokens() { .. }
}
```

Если набросать эскиз воздействий для данного класса, то можно обнаружить, что метод `parseExpression` зависит от методов `getToken` (получить лексему) и `hasMoreTokens` (содержит дополнительные лексемы), но не зависит напрямую от полей `stringToParse` (анализируемая строка) или `currentPosition` (текущее положение), в отличие от методов `getToken` и `hasMoreTokens`. В данном случае мы имеем дело с естественной границей инкапсуляции, хотя она и не узкая (в двух методах скрываются два фрагмента данных). Мы можем извлечь эти методы и поля в отдельный класс под названием `Tokenizer` (преобразователь в лексемы) и тем самым упростить класс `Parser`.

Это не единственный способ, позволяющий выяснить, как разделить ответственность в классе. Иногда на это указывают имена, как в данном примере, где у нас имеются два метода, в именах которых присутствует слово `Token` (лексема). Анализ имен помогает взглянуть на крупный класс иначе и обнаружить подходящие способы извлечения его компонентов.

В качестве упражнения составьте эскиз воздействий для изменений в крупном классе и, не обращая внимания на имена обведенных кружками компонентов этого класса, просто посмотрите, как они сгруппированы. Можете ли вы провести естественные границы инкапсуляции? Если да, то обратите внимание на обведенные кружками компоненты в пределах этих границ. Придумайте подходящее имя для данной группы компонентов (методов и переменных); оно может стать именем нового класса. Подумайте также, насколько полезным может оказаться любое изменение имен.

Проделайте то же самое со своими коллегами по работе. Обсуждение вопросов присвоения имен может иметь положительные последствия не только для текущей работы, но и в отдаленной перспективе, поскольку помогает группе разработчиков выработать общее представление о проектируемой системе в настоящем и том, какой она может стать в будущем.

Написание тестов в точках сужения — идеальный способ приступить к радикальным изменениям в отдельной части программы. Вы создаете задел, извлекая ряд классов и доводя их до такого состояния, когда вы сможете получать их экземпляры в средствах тестирования. Написав собственные *характеристические тесты*, вы можете уже без опасения вносить любые изменения в код. Таким образом, вы создаете в своем приложении небольшой оазис, где вам удобно и легко работать с кодом. Но будьте осторожны — точки сужения таят в себе скрытые препятствия.

Скрытые препятствия, которые таят в себе точки сужения

При написании блочных тестов следует опасаться ряда крупных неприятностей. Прежде всего, нельзя допускать, чтобы блочные тесты медленно разрастались до масштабов комплексных испытаний системы в миниатюре. Ведь нам требуется протестировать класс, а для этого мы получаем экземпляры нескольких взаимодействующих с ним объектов и передаем их данному классу. Далее мы проверяем некоторые значения, чтобы быть уверенными в том, что вся совокупность объектов хорошо взаимодействует. Но дело в том, что если мы делаем это слишком часто, то в конечном счете блочные тесты получаются крупными, массивными и бесконечно долго выполняющимися.

Главное при написании блочных тестов для нового кода — тестировать классы как можно более независимым образом. Как только вы заметите, что тесты становятся слишком крупными, разделите тестируемый класс на более мелкие независимые части, которые легче тестировать по отдельности. Иногда для этого приходится подделывать остальные объекты, взаимодействующие с тестируемым объектом, поскольку в задачу блочного теста входит проверка поведения одного объекта, а не совместного поведения совокупности объектов. Такое поведение проще проверить, используя фиктивный объект.

Совсем иной оказывается ситуация при написании блочных тестов для существующего кода. Иногда для этого целесообразно отделить часть приложения и наполнить ее тестами. Как только эти тесты окажутся на своих местах, станет намного легче писать более узконаправленные блочные тесты для каждого класса, которого затрагивают вносимые в код изменения. В конечном счете тесты в точках сужения можно удалить за ненадобностью.

Тесты в точках сужения подобны отметкам застолбленного участка земли. Зная, что застолбленный участок принадлежит вам, вы можете свободно реорганизовать его код и написать дополнительные тесты. А со временем вы сумеете удалить тесты в точках сужения и предоставить тестам, написанным для каждого класса, возможность поддерживать вашу разработку.

Глава 13

В код требуется внести изменения, но неизвестно, какие тесты писать

Когда программисты говорят о тестировании, они обычно имеют в виду тесты, с помощью которых они обнаруживают программные ошибки. И зачастую такие тесты оказываются ручными. Написание автоматических тестов для обнаружения программных ошибок в унаследованном коде нередко оказывается не таким эффективным, как обычное испытание кода на работоспособность. Если имеется какой-нибудь способ испытания унаследованного кода вручную, то программные ошибки обнаруживаются в нем очень быстро. Недостаток такого подхода состоит в том, что подобную работу приходится выполнять вручную всякий раз, когда в код вносятся изменения. И откровенно говоря, программисты этого просто не делают. Практически каждая группа разработчиков, с которой мне приходилось работать и в которой применялось ручное тестирование изменений, вносившихся в код, в конце концов сильно отставала от графика выполняемых работ. Уровень доверия среди членов таких групп оказывался не таким высоким, каким он мог бы быть в действительности.

Обнаружение программных ошибок в унаследованном коде обычно не вызывает особых трудностей. Но в стратегическом плане это могут оказаться усилия, направленные в неверное русло. Ведь зачастую лучше принять такие меры, которые помогли бы группе разработчиков начать постоянно писать правильный код. Путь к успеху лежит в сосредоточении усилий на исключении появления программных ошибок в коде.

Автоматические тесты являются очень важным инструментальным средством, но только не для обнаружения программных ошибок — по крайней мере, не напрямую. Как правило, в автоматических тестах должны быть указаны цель, которой мы хотели бы достичь, или хотя бы попытка сохранить уже имеющееся поведение. В ходе разработки тесты, *указывающие* конечную цель, естественным образом превращаются в тесты, *сохраняющие* поведение. Мы обнаруживаем программные ошибки не при первом, а, как правило, при последующем выполнении теста, когда мы изменяем поведение неожиданным для себя образом.

А как все это соотносится с унаследованным кодом? В унаследованном коде у нас может и не быть подходящих тестов для изменений, которые нам требуется внести, а следовательно, мы никак не можем проверить, сохраняем ли мы поведение при внесении изменений в такой код. По этой причине лучшее, что мы можем сделать, когда нам требуется внести изменения в унаследованный код, — это подкрепить изменяемый участок кода тестами подобно страховочной сетке. Если по ходу дела мы обнаружим программные ошибки, то нам придется их устранять, но если обнаружение и устранение всех программных ошибок превращается в нашу главную цель, то нам, как правило, так и не удается довести дело до конца.

Характеристические тесты

Итак, нам нужны тесты, но как их написать? Для этого можно, в частности, выяснить основные функции программы и на их основании написать тесты. С этой целью мы можем попытаться достать старую документацию, чтобы ознакомиться с техническими требованиями и пояснительными записками к проекту, а затем приступить к написанию тестов. Это один подход, но он далеко не самый лучший. Ведь практически в любой унаследованной системе намного важнее то, что она на самом делает, а не, что она должна делать. Если мы будем писать тесты, исходя из того, что должна делать система, то опять вернемся к обнаружению программных ошибок. Разумеется, обнаружение ошибок — очень важная задача, но ведь наша цель состоит в размещении тестов по местам, чтобы они помогали нам вносить изменения более определенно.

Тесты, которые нам требуются для сохранения поведения, называются *характеристическими*. Такие тесты характеризуют конкретное поведение фрагмента кода. Они не имеют никакого отношения к тому, что должна или может делать система, а просто документируют ее конкретное поведение.

Ниже приведен небольшой алгоритм написания характеристических тестов.

1. Ввести небольшой фрагмент кода в средства тестирования.
2. Написать утверждение, которое заведомо приведет к сбою или непрохождению теста.
3. Предоставить сбою возможность выявить конкретное поведение.
4. Изменить тест так, чтобы в нем предполагалось поведение, которое дает код.
5. Повторить.

Ниже приведен пример теста, в отношении которого можно быть более или менее уверенным в том, что класс `PageGenerator` (формирователь страницы) не сформирует строку `"fred"`.

```
void testGenerator() {
    PageGenerator generator = new PageGenerator();
    assertEquals("fred", generator.generate());
}
```

Выполните свой тест и позвольте ему не пройти. Если он не пройдет, то вы выясните, что именно код делает при таком условии. Так, в приведенном выше коде вновь созданный класс `PageGenerator` сформирует пустую строку при вызове его метода `generate`. Ниже приведен результат выполнения данного теста.

```
.F
Time: 0.01
(Время выполнения: 0.01)
There was 1 failure:
(обнаружен 1 сбой)
1) testGenerator(PageGeneratorTest)
junit.framework.ComparisonFailure: expected:<fred> but was:<>
(ожидалось: <fred>, а получилось: <>)
    at PageGeneratorTest.testGenerator
        (PageGeneratorTest.java:9)
    at sun.reflect.NativeMethodAccessorImpl.invoke0
```

```
        (Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke
        (NativeMethodAccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke
        (DelegatingMethodAccessorImpl.java:25)

FAILURES!!!
(СБОИ!!!)
Tests run: 1, Failures: 1, Errors: 0
(Выполненных тестов: 1, Сбоев: 1, Ошибок: 0)
```

Далее мы можем изменить тест таким образом, чтобы он прошел.

```
void testGenerator() {
    PageGenerator generator = new PageGenerator();
    assertEquals("", generator.generate());
}
```

Теперь тест проходит. Более того, он документирует один из самых основных фактов, касающихся класса `PageGenerator`: если при создании данного класса сразу же предписать ему сформировать что-нибудь, то он сформирует пустую строку.

Аналогичным образом мы можем выяснить, каким будет поведение, если ввести в тест другие данные.

```
void testGenerator() {
    PageGenerator generator = new PageGenerator();
    generator.assoc(RowMappings.getRow(Page.BASE_ROW));
    assertEquals("fred", generator.generate());
}
```

В данном случае в сообщении об ошибке, выдаваемом средствами тестирования, указывается получаемая в результате строка `"<node><carry>1.1 vectrai</carry></node>"`, и поэтому мы можем сделать эту строку предполагаемым в тесте значением.

```
void testGenerator() {
    PageGenerator generator = new PageGenerator();
    assertEquals("<node><carry>1.1 vectrai</carry></node>",
        generator.generate());
}
```

Если такие тесты действительно считать тестами, то в них есть нечто принципиально странное. Ведь если мы только вводим в них значения, которые выдает программа, то что они вообще тестируют? Что, если в программе имеется ошибка? В таком случае предполагаемые значения, вводимые в тесты, могут просто оказаться неверными.

Все становится на свои места, если посмотреть на эти тесты иначе. Они на самом деле не пишутся как некая норма для функционирования программного обеспечения. С их помощью мы не пытаемся немедленно обнаружить программные ошибки, а лишь пробуем ввести механизм для обнаружения в дальнейшем ошибок, которые позволили бы нам выявить отличия от текущего поведения системы. С учетом этой цели изменится и наше представление о таких тестах: они не имеют никакого особого веса, а лишь документируют то, что система действительно делает. Если мы точно знаем, что делают отдельные части системы, то можем использовать это знание вместе со знанием того, что система должна делать, для внесения изменений в код. На самом деле знать, что именно система делает в действительности, очень важно. Для того чтобы выяснить, какое именно поведение нам

требуется ввести, мы обычно общаемся с другими или же проводим некоторые расчеты, но за исключением тестов, у нас нет другого способа узнать, что же система действительно делает, если, конечно, "не напрягать мозги", читая код и пытаясь вычислить значения, получаемые в отдельные моменты времени. Одним удается делать это быстрее, чем другим, но, независимо от интеллектуальных способностей, такой умственный труд весьма утомителен и непроизводителен, если заниматься им постоянно.

Характеристические тесты регистрируют конкретное поведение фрагмента кода. Если же мы обнаружим что-нибудь неожиданное при их написании, то мы должны постараться прояснить ситуацию: возможно, это была программная ошибка. Но это совсем не означает, что мы не должны включать такой тест в наш набор тестов. Напротив, мы должны отметить его как подозрительный и выяснить, к какому результату приведет его исправление.

Написание характеристических тестов отличается многими другими особенностями по сравнению с описанными выше. В приведенном выше примере формирователя страницы тестовые значения получались едва ли не вслепую, когда значения вводились в код и выводились обратно в утверждениях. Это вполне допустимо, если хорошо представлять себе, что именно код должен делать. В некоторых случаях, например, когда, ничего не делая с объектом, можно посмотреть, что же возвращают отдельные методы, код нетрудно осмыслить и охарактеризовать, но что делать дальше? Каково общее число тестов, применяемых к такому компоненту системы, как формирователь кода? Это число бесконечно. Мы можем очень долго составлять один контрольный пример за другим для такого класса. На чем же остановиться? Можно ли узнать, насколько одни контрольные примеры важнее других?

Очень важно понимать, что мы не пишем в данном случае тесты по принципу черного ящика. У нас все же есть возможность анализировать код, чтобы охарактеризовать его. Ведь сам код может подсказать нам, что именно он делает, и если у нас возникают вопросы, то тесты — идеальное средство разрешить эти вопросы. Для того чтобы охарактеризовать код, следует, прежде всего, проявить любопытство к его поведению. На данном этапе мы просто пишем тесты до тех пор, пока не удовлетворим свое любопытство. Достаточно ли этого, чтобы покрыть код тестами полностью? Вероятно, нет. Тогда мы переходим к следующему этапу, на котором мы осмысливаем изменения, вносимые в код, и пытаемся выяснить, можно ли распознать с помощью имеющихся у нас тестов любые осложнения, которые могут возникнуть в связи с изменением кода. Если это невозможно, то мы вводим дополнительные тесты до тех пор, пока не будем уверены в том, что тесты сделают это возможным. Если же мы не можем быть в этом уверены, то надежнее внести изменения в программу другим способом. Возможно, стоит частично вернуться к первоначальному варианту.

Правило применения метода

Прежде чем применять метод в унаследованной системе, проверьте, имеются ли для него тесты. В противном случае напишите их. Если вы делаете это постоянно, то пользуйтесь тестами как средством общения. Анализируя их, другие смогут лучше понять, чего следует и чего не следует ожидать от данного метода. Само действие, направленное на то, чтобы сделать класс тестируемым, способствует повышению качества кода. В таком коде легче разобраться, изменить его, исправить программные ошибки и двигаться дальше.

Выяснение характерных особенностей классов

Допустим, что у нас имеется класс и нам нужно выяснить, что и как в нем тестировать. Как нам это сделать? Прежде всего, мы можем попытаться выяснить, что именно класс делает на высоком уровне. Мы можем сначала написать тесты для самой простой, на наш взгляд, функции этого класса, а далее руководствоваться своим любопытством. Ниже приведена процедура эвристического анализа, которая может оказаться полезной в данном случае.

1. Найдите фрагменты запутанной логики. Если вы не понимаете выбранный фрагмент кода, рассмотрите возможность для *внедрения переменной распознавания*, чтобы охарактеризовать его. Используйте переменные распознавания, чтобы убедиться в том, что вы выполняете конкретные участки кода.
2. Обнаруживая разные виды ответственности класса или метода, останавливайтесь на время для составления списка действий, которые могут оказаться неверными. Попробуйте сформулировать тесты, которые запускают эти действия.
3. Обдумайте входные данные, предоставляемые в тесте, а также возможные последствия ввода крайних значений.
4. Выясните, должны ли любые условия соблюдаться постоянно в течение срока действия класса. Зачастую такие условия называются инвариантами. Попробуйте написать тесты для их проверки. Для обнаружения подобных условий нередко требуется реорганизация кода, которая обычно дает новое представление о том, каким должен быть код.

Тесты, которые пишутся для того, чтобы охарактеризовать код, очень важны. Ведь они документируют настоящее поведение системы. Как и при написании любой другой документации, обдумайте то, что должно быть наиболее важным для пользователя. Поставив себя на его место, задайте себе вопрос: что бы вы сами хотели узнать о классе, с которым вы работаете, если он никогда не встречался вам прежде? В каком порядке вы хотели бы получить информацию о нем? Если вы пользуетесь средами тестирования xUnit, то тесты представляют собой обычные методы, сохраняемые в файле. Разместите их в таком порядке, чтобы пользователям было удобнее изучать испытуемый ими код. Начните с самых простых контрольных примеров, демонстрирующих основное назначение класса, а затем перейдите к примерам, выявляющим его характерные особенности. Непременно задокументируйте самые важные особенности, обнаруживаемые тестами. Перейдя к внесению изменений, вы, скорее всего, обнаружите, что написанные вами тесты вполне подходят для работы, которую вы собираетесь выполнять. Изменения, которые вы собираетесь внести в код, сознательно или бессознательно направляют ваше любопытство в нужное русло.

Что делать при обнаружении программных ошибок

В процессе выявления характерных особенностей унаследованного кода иногда обнаруживаются программные ошибки. Такие ошибки характерны для любого унаследованного кода и находятся в прямой зависимости от степени его запутанности. Что же делать при обнаружении программных ошибок?

Ответ на этот вопрос зависит от конкретной ситуации. Если система никогда не разворачивалась, то ответ прост: программные ошибки следует устранить. Если же система разворачивалась, то следует выяснить вероятность чьей-то зависимости от такого поведения — даже если в нем усматривается программная ошибка. Для того чтобы найти

способ устранения программной ошибки, не вызывая цепную реакцию, зачастую требуется некоторый анализ ситуации.

Я лично склоняюсь к устранению программных ошибок, как только они обнаруживаются. Если поведение оказывается явно ошибочным, то его следует исправить. Если же возникает подозрение в неправильности какого-то поведения, то его следует сначала отметить в тестовом коде как подозрительное, а затем усугубить. Необходимо как можно быстрее выяснить, является ли такое поведение программной ошибкой, и найти наилучший способ ее устранения.

Нацеленное тестирование

После написания тестов для лучшего понимания фрагмента кода мы переходим к анализу тех элементов, которые требуется в нем изменить, и выясняем, насколько тесты их покрывают. Ниже приведен пример метода из класса Java, вычисляющего стоимость топлива в арендуемых резервуарах.

```
public class FuelShare
{
    private long cost = 0;
    private double corpBase = 12.0;
    private ZonedHawthorneLease lease;
    ...
    public void addReading(int gallons, Date readingDate){
        if (lease.isMonthly()) {
            if (gallons < Lease.CORP_MIN)
                cost += corpBase;
            else
                cost += 1.2 * priceForGallons(gallons);
        }
        ...
        lease.postReading(readingDate, gallons);
    }
    ...
}
```

Нам требуется внести самое непосредственное изменение в класс `FuelShare` (квота на топливо). Мы уже написали для этого ряд тестов и поэтому готовы к внесению изменений, которые заключаются в следующем: нам требуется извлечь из данного класса оператор `if` верхнего уровня в новый метод, а затем перенести этот метод в класс `ZonedHawthorneLease` (аренда в районе Готорна). Переменная аренды (`lease`) в изменяемом коде представляет собой экземпляр данного класса.

После реорганизации код будет выглядеть следующим образом.

```
public class FuelShare
{

    public void addReading(int gallons, Date readingDate){
        cost += lease.computeValue(gallons,
                                   priceForGallons(gallons));
```

```
        ...
        lease.postReading(readingDate, gallons);
    }
    ...
}

public class ZonedHawthorneLease extends Lease
{
    public long computeValue(int gallons, long totalPrice) {
        long cost = 0;
        if (lease.isMonthly()) {
            if (gallons < Lease.CORP_MIN)
                cost += corpBase;
            else
                cost += 1.2 * totalPrice;
        }
        return cost;
    }
    ...
}
```

Какого рода тесты нам понадобятся, чтобы убедиться в правильности такой реорганизации кода? В данный момент очевидно лишь одно: мы вообще не собираемся видоизменять следующий фрагмент кода:

```
if (gallons < Lease.CORP_MIN)
    cost += corpBase;
```

Конечно, было бы неплохо разместить на месте тест, чтобы посмотреть, как количество топлива в галлонах вычисляется ниже предела `Lease.CORP_MIN` (корпоративный минимум для аренды), но в этом нет особой необходимости. С другой стороны, в исходном коде подлежит изменению следующий оператор `else`:

```
else
    valueInCents += 1.2 * priceForGallons(gallons);
```

Когда этот код переносится в новый метод, он принимает следующий вид:

```
else
    valueInCents += 1.2 * totalPrice;
```

Это незначительное, но все же изменение. Поэтому нам следует проверить правильность выполнения данного оператора `else` в одном из наших тестов. Рассмотрим исходный метод еще раз.

```
public class FuelShare
{
    public void addReading(int gallons, Date readingDate){
        if (lease.isMonthly()) {
            if (gallons < CORP_MIN)
                cost += corpBase;
            else
                cost += 1.2 * priceForGallons(gallons);
        }
```

```
        ...
        lease.postReading(readingDate, gallons);
    }
    ...
}
```

Если мы в состоянии создать объект класса `FuelShare` с месячной арендой и попытаемся выполнить метод `addReading` (ввести показания) для количества топлива в галлонах, превышающего предел `Lease.CORP_MIN`, то сможем испытать данную ветвь оператора `else`.

```
public void testValueForGallonsMoreThanCorpMin() {
    StandardLease lease = new StandardLease(Lease.MONTHLY);

    FuelShare share = new FuelShare(lease);
    share.addReading(FuelShare.CORP_MIN +1, new Date());
    assertEquals(12, share.getCost());
}
```

Когда вы пишете тест для испытания ветви программы, выясните, имеется ли другой способ для прохождения теста, помимо выполнения данной ветви. Если вы не уверены в этом, то воспользуйтесь *переменной распознавания* или же отладчиком, чтобы выяснить для теста такую возможность.

Охарактеризовывая подобные ветви программы, очень важно выяснить, имеется ли у предоставляемых входных данных особое поведение, способное привести к успешному прохождению теста, когда он не должен пройти. Обратимся к примеру. Допустим, что в приведенном ниже коде для представления стоимости топлива используется тип данных `double` вместо `int`.

```
public class FuelShare
{
    private double cost = 0.0;
    ...
    public void addReading(int gallons, Date readingDate){
        if (lease.isMonthly()) {
            if (gallons < CORP_MIN)
                cost += corpBase;
            else
                cost += 1.2 * priceForGallons(gallons);
        }
        ...
        lease.postReading(readingDate, gallons);
    }
    ...
}
```

Речь в данном случае идет не о потере в приложении дробных долей центов из-за ошибки округления стоимости до плавающей десятичной точки, а о более серьезных упущениях. Если не отнестись внимательно к выбору входных данных, то при извлечении метода можно совершить ошибку, даже не осознавая этого. Одна из характерных ошибок может произойти в том случае, если мы попытаемся извлечь метод и присвоить одному из его аргументов тип данных `int` вместо `double`. В Java и многих других языках программирования поддерживается автоматическое преобразование типа данных `double` в `int`, которые

во время выполнения просто усекаются. Поэтому если не отнестись к выбору входных данных внимательно, то подобную ошибку легко допустить, но намного труднее исправить.

Обратимся к конкретному примеру. Как повлияет на упомянутый выше код выбор значения `10` переменной `Lease.CORP_MIN` и значения `12,0` переменной `corpBase` при выполнении приведенного ниже теста?

```
public void testValue () {
    StandardLease lease = new StandardLease(Lease.MONTHLY);
    FuelShare share = new FuelShare(lease);

    share.addReading(1, new Date());
    assertEquals(12, share.getCost());
}
```

Значение `1` меньше значения `10`, и поэтому значение `12,0` просто добавляется к исходному нулевому значению переменной `cost`. Пока что все идет хорошо, но что, если мы извлечем подобный метод и объявим переменную `cost` типа `long`, а не `double`?

```
public class ZonedHawthorneLease
{
    public long computeValue(int gallons, long totalPrice) {
        long cost = 0;
        if (lease.isMonthly()) {
            if (gallons < CORP_MIN)
                cost += corpBase;
            else
                cost += 1.2 * totalPrice;
        }
        return cost;
    }
}
```

Написанный нами тест по-прежнему проходит, несмотря на то, что мы незаметно усекли значение переменной `cost` при ее возврате, когда было выполнено преобразование типа данных `double` в `long`, но оно не было испытано полностью. Произошло то же самое, что и в отсутствие подобного преобразования, как если бы целое значение типа `int` было присвоено переменной типа `int`.

> Выполняя реорганизацию кода, мы должны проверять две вещи — существует ли поведение после реорганизации кода и связано ли оно правильно?
>
> Многие характеристические тесты подобны проверкам на "безоблачность". Они проверяют не множество специальных условий, а только присутствие в коде конкретного поведения. По их наличию мы можем заключить, что в результате реорганизации кода, выполненной нами для переноса или извлечения кода, поведение сохранилось.

Как же нам выйти из подобного положения? Из него имеются три выхода. Во-первых, мы можем рассчитать вручную значения, предполагаемые в тестируемом фрагменте кода. При каждом преобразовании мы видим, возникает ли проблема усечения данных. Во-вторых, мы можем воспользоваться отладчиком и проверить в пошаговом режиме присвоения значений, чтобы посмотреть, какие именно преобразования типов вызывают входные данные. И в-третьих, мы можем прибегнуть к *внедрению переменных распознавания*, чтобы убедиться в покрытии конкретной ветви программы и испытании соответствующих преобразований.

Самые ценные характеристические тесты испытывают конкретную ветвь программы и каждое преобразование в этой ветви.

Имеется и четвертый выход: просто охарактеризовать более мелкий фрагмент кода. Если в нашем распоряжении имеется инструментальное средство реорганизации кода, которое помогает нам безопасно извлекать методы, то мы можем извлечь более мелкие методы, чем метод `computeValue` (вычислить стоимость) в приведенном выше примере, и написать тесты для этих фрагментов кода. К сожалению, инструментальные средства реорганизации кода поддерживаются не во всех языках программирования, а иногда и те средства, которые доступны, не позволяют извлекать методы так, как нам этого бы хотелось.

Особенности инструментальных средств реорганизации кода

Хорошее инструментальное средство реорганизации кода просто неоценимо, но зачастую тем, кто пользуется такими средствами, приходится реорганизовывать код вручную. Рассмотрим типичный пример. Допустим, что у нас имеется класс `A` с некоторым кодом, который мы хотели бы извлечь из его метода `b()`.

```
public class A
{
    int x = 1;
    public void b() {
        int y = 0;
        int c = x + y;
    }
};
```

Если нам требуется извлечь выражение `x + y` из метода `b()` и создать метод под названием `add` (сложить), то с такой задачей может справиться, по крайней мере, одно из имеющихся на рынке инструментальных средств реорганизации кода, хотя оно извлечет код в виде `add(y)`, а не в виде `add(x, y)`. Почему? А потому что `x` является переменной экземпляра, доступной для любых извлекаемых нами методов.

Эвристический анализ для написания характеристических тестов

Ниже приведена процедура эвристического анализа, которая оказывается полезной для написания характеристических тестов.

1. Напишите тесты для участка кода, где вы предполагаете внести изменения. Затем напишите столько контрольных примеров, сколько, на ваш взгляд, потребуется для того, чтобы понять поведение кода.
2. После этого проанализируйте конкретные элементы кода, которые вы собираетесь изменить, и попытайтесь написать для них тесты.
3. Если вы пытаетесь извлечь или перенести ряд функций, то напишите тесты, проверяющие существование и связанность поведения этих функций на последовательном ряде конкретных контрольных примеров. Убедитесь в том, что вы испытываете именно тот код, который вы собираетесь перенести, и что он связан надлежащим образом. Испытайте преобразования.

Глава 14

Убийственная зависимость от библиотек

Разработке программного обеспечения в значительной степени способствует, в частности, многократное использование кода. Если мы приобретем библиотеку, решающую многие задачи разработки за нас, и научимся ею пользоваться, то, скорее всего, сумеем существенно сократить сроки выполнения проекта. Единственный недостаток такого подхода заключается в том, что мы можем очень легко стать слишком зависимыми от библиотеки, особенно если мы пользуемся ею в коде без разбора. Некоторые группы разработчиков, с которыми мне приходилось работать, сильно обожглись, слишком полагаясь в своей работе на библиотеки. А в одном случае поставщик библиотеки настолько поднял лицензионную плату за свою библиотеку, что выпущенное на рынок приложение не окупило затраты на разработку. Его разработчики просто не смогли перейти на библиотеку другого поставщика, поскольку для удаления вызовов библиотеки первоначального поставщика пришлось бы переписывать весь код.

Избегайте беспорядочного расположения прямых вызовов библиотек в своем коде. Если вы посчитаете, что вам никогда не придется изменять их, то накликаете на себя беду.

На момент написания этой книги бо́льшая часть разработчиков программного обеспечения была разделена на два лагеря — Java и .NET. Обе компании Microsoft и Sun сделали все мыслимое для как можно более широкого распространения своих платформ, создавая многие библиотеки, с тем чтобы разработчики продолжали и впредь пользоваться их программными продуктами. В какой-то степени это благо для многих проектов, но сильная зависимость от библиотек может по-прежнему оставаться. Всякое применение в коде жестко кодированного библиотечного класса служит тем местом, где мог бы быть шов. В одних библиотеках неплохо определены интерфейсы для всех их конкретных классов, а в других — конкретные классы объявлены как *конечные* или *герметичные* или же содержат главные функции, не являющиеся виртуальными, не оставляя никакой возможности для их подделки при тестировании. В подобных случаях самым лучшим выходом из положения может оказаться написание тонкой оболочки поверх тех классов, которые требуется отделить. Не забудьте при этом написать поставщику используемой вами библиотеки жалобу на те трудности, которые она представляет для разработки.

Разработчики библиотек, использующие языковые средства для соблюдения проектных ограничений, нередко совершают ошибку. Они забывают о том, что качественный код выполняется в условиях производства и тестирования. Ограничения для первых могут сделать код неработоспособным в последних.

Между языковыми средствами, способствующими внедрению в жизнь удачных проектных решений, и задачами, которые приходится решать при тестировании кода, существуют глубокие противоречия. Одним из самых главных противоречий подобного рода

является *дилемма однократности*. Если в библиотеке предполагается, что в системе должен существовать только один экземпляр класса, то использование фиктивных объектов данного класса оказывается затруднительным. В частности, отсутствует всякая возможность воспользоваться *вводом статического установщика* или многими другими способами разрыва зависимостей, подходящими для тех случаев, когда приходится иметь дело с одиночками. Иногда единственным выходом из этого положения оказывается заключение одиночки в оболочку.

С этим связана также *дилемма ограниченности переопределения*. В одних языках ООП все методы являются виртуальными, в других — они виртуальны по умолчанию, но могут быть сделаны невиртуальными, а в третьих — их приходится делать виртуальными явно. С точки зрения проектирования есть определенный смысл в том, что некоторые методы могут быть сделаны невиртуальными. В разное время разработчики предлагали делать методы невиртуальными как можно чаще. Иногда их аргументы выглядели убедительными, но ведь нельзя отрицать тот факт, что подобная практика затрудняет введение распознавания и разделения в базы кода. Нелегко отрицать и тот факт, что разработчики часто пишут очень качественный код на языке Smalltalk, где такая практика вообще невозможна, на Java, где это, как правило, не делается, и даже на C++, где немалая доля кода вообще обходится без этого. Если же просто сделать вид, что общедоступный метод является невиртуальным в выходном коде, то его можно выборочно переопределить в тестовом коде и тем самым убить сразу двух зайцев.

Иногда придерживаться соглашения по программированию оказывается не менее полезно, чем использовать ограничивающее языковое средство. В связи с этим рекомендуется тщательно обдумать, что именно требуется для тестирования.

Глава 15

Приложение состоит из сплошных вызовов интерфейса API

Создавать, приобретать и заимствовать. Такой выбор остается у нас при разработке программного обеспечения. Всякий раз, когда мы разрабатываем приложение, у нас возникает сильное ощущение, что мы могли бы сэкономить немного времени и труда, приобретя библиотеку у определенного поставщика, используя открытые программные средства и даже значительные фрагменты кода из библиотек, связанных с используемыми нами платформами (J2EE, .NET и прочими). Выбирая для внедрения в приложение код, который нелегко изменить, приходится принимать во внимание самые разные обстоятельства. Ведь нам нужно знать, насколько этот код устойчив, самодостаточен и прост в использовании. А когда мы все же решаем воспользоваться чужим кодом, то нередко оставляем нерешенной еще одну проблему: наше приложение получается в итоге таким, как будто оно состоит из сплошных вызовов чужой библиотеки. Как же внести изменения в такой код?

В связи с этим сразу же возникает искушение вообще отказаться от тестирования кода. Ведь мы не делаем ничего особенного, а лишь вызываем в своем коде готовые методы, а сам код достаточно прост. И он действительно прост. Так, что же в этом плохого?

Многие унаследованные проекты начинались так же скромно и просто, но затем они постепенно разрастались до таких масштабов, что уже переставали быть простыми. Со временем в таком коде можно обнаружить участки без обращения к интерфейсу API, но они вплетены в пеструю мозаику неустойчивого кода. Приложение приходится выполнять всякий раз, когда в него вносятся изменения, чтобы убедиться в его работоспособности, что в конечном счете возвращает нас к главной дилемме работы с унаследованным кодом, которая состоит в неопределенности изменений: мы еще не написали весь код, но нам уже нужно его сопровождать.

Иметь дело с системами, испещренными библиотечными вызовами, во многих отношениях намного труднее, чем с доморощенными системами. Во-первых, очень трудно зачастую решить, как же улучшить структуру кода, поскольку в нем встречаются сплошные вызовы интерфейса API. В ней отсутствует какой-либо намек на подобную возможность. Во-вторых, работать с системами, в которых интенсивно используется интерфейс API, нелегко, поскольку это чужой прикладной интерфейс. Если бы он нам принадлежал, то мы могли бы переименовать интерфейсы, классы и методы, чтобы упростить свою задачу, или же добавить дополнительные методы в классы, чтобы сделать их доступными в разных частях кода.

Обратимся к конкретному примеру. Ниже приведен листинг очень плохо написанного кода для сервера рассылки почты. Нельзя даже с уверенностью сказать, что этот код работает.

```
import java.io.IOException;
import java.util.Properties;
```

```
import javax.mail.*;
import javax.mail.internet.*;

public class MailingListServer
{
    public static final String SUBJECT_MARKER = "[list]";
    public static final String LOOP_HEADER = "X-Loop";

    public static void main (String [] args) {
        if (args.length != 8) {
            System.err.println ("Usage: java MailingList <popHost> " +
                "<smtpHost> <pop3user> <pop3password> " +
                "<smtpuser> <smtppassword> <listname> " +
                "<relayinterval>");
           return;
    }

    HostInformation host = new HostInformation (
            args [0], args [1], args [2], args [3],
            args [4], args [5]);
    String listAddress = args[6];
    int interval = new Integer (args [7]).intValue ();
    Roster roster = null;
    try {
        roster = new FileRoster("roster.txt");
    } catch (Exception e) {
        System.err.println ("unable to open roster.txt");
        return;
    }
    try {
        do {
            try {
                Properties properties = System.getProperties ();
                Session session = Session.getDefaultInstance (
                      properties, null);
                Store store = session.getStore ("pop3");
                store.connect (host.pop3Host, -1,
                      host.pop3User, host.pop3Password);
                Folder defaultFolder = store.getDefaultFolder();
                if (defaultFolder == null) {
                    System.err.println("Unable to open default folder");
                    return;
                }
                Folder folder = defaultFolder.getFolder ("INBOX");
                if (folder == null) {
                    System.err.println("Unable to get: "
                            + defaultFolder);
                    return;
                }
                folder.open (Folder.READ_WRITE);
```

```
            process(host, listAddress, roster, session,
                store, folder);
        } catch (Exception e) {
              System.err.println(e);
              System.err.println ("(retrying mail check)");
        }
        System.err.print (".");
        try { Thread.sleep (interval * 1000); }
        catch (InterruptedException e) {}
        } while (true);
    }
    catch (Exception e) {
        e.printStackTrace ();
    }
}

private static void process(
    HostInformation host, String listAddress, Roster roster,
    Session session,Store store, Folder folder)
        throws MessagingException {
  try {
      if (folder.getMessageCount() != 0) {
          Message[] messages = folder.getMessages ();
         doMessage(host, listAddress, roster, session,
             folder, messages);
      }
  } catch (Exception e) {
    System.err.println ("message handling error");
    e.printStackTrace (System.err);
  }
  finally {
      folder.close (true);
      store.close ();
  }
}

private static void doMessage(
    HostInformation host,
    String listAddress,
    Roster roster,
    Session session,
    Folder folder,
    Message[] messages) throws
        MessagingException, AddressException, IOException,
        NoSuchProviderException {
    FetchProfile fp = new FetchProfile ();
    fp.add (FetchProfile.Item.ENVELOPE);
    fp.add (FetchProfile.Item.FLAGS);
    fp.add ("X-Mailer");
    folder.fetch (messages, fp);
    for (int i = 0; i < messages.length; i++) {
```

```
            Message message = messages [i];
            if (message.getFlags ().contains (Flags.Flag.DELETED))
                continue;
            System.out.println("message received: "
                + message.getSubject ());
            if (!roster.containsOneOf (message.getFrom ()))
                continue;
            MimeMessage forward = new MimeMessage (session);
            InternetAddress result = null;
            Address [] fromAddress = message.getFrom ();
            if (fromAddress != null && fromAddress.length > 0)
                result =
                    new InternetAddress (fromAddress [0].toString ());
            InternetAddress from = result;
            forward.setFrom (from);
            forward.setReplyTo (new Address [] {
                new InternetAddress (listAddress) });
            forward.addRecipients (Message.RecipientType.TO,
                 listAddress);
            forward.addRecipients (Message.RecipientType.BCC,
                roster.getAddresses ());
            String subject = message.getSubject();
            if (-1 == message.getSubject().indexOf (SUBJECT_MARKER))
                subject = SUBJECT_MARKER + " " + message.getSubject();
            forward.setSubject (subject);
            forward.setSentDate (message.getSentDate ());
            forward.addHeader (LOOP_HEADER, listAddress);
            Object content = message.getContent ();
            if (content instanceof Multipart)
                forward.setContent ((Multipart)content);
            else
                forward.setText ((String)content);

            Properties props = new Properties ();
            props.put ("mail.smtp.host", host.smtpHost);

            Session smtpSession =
                Session.getDefaultInstance (props, null);
            Transport transport = smtpSession.getTransport ("smtp");
            transport.connect (host.smtpHost,
                host.smtpUser, host.smtpPassword);
            transport.sendMessage (forward, roster.getAddresses ());
            message.setFlag (Flags.Flag.DELETED, true);
        }
    }
}
```

Это относительно небольшой фрагмент кода, но он не очень ясен. В нем трудно найти строки без обращения к интерфейсу API. Можно ли улучшить структуру такого кода, причем так, чтобы упростить внесение в него изменений? Да, можно.

Прежде всего необходимо выявить вычислительное ядро в этом коде, т.е. то, что он действительно делает. Для этого полезно составить следующее краткое описание функций данного кода.

Этот код считывает данные конфигурации из командной строки и список адресов электронной почты из файла. Он периодически проверяет почту. Обнаружив почту, он отправляет ее по каждому из адресов, хранящихся в файле.

На первый взгляд, основной функцией данной программы является ввод и вывод, но у нее имеются и другие функции. В этом коде выполняется отдельный поток. Исходно этот поток неактивен, но периодически он активизируется для проверки почты. Кроме того, входящие почтовые сообщения не просто отправляются по другому адресу, а из них формируются новые сообщения. Для этого требуется установить все поля сообщений, проверить и изменить поле темы, чтобы отразить в нем тот факт, что сообщение поступает из списка рассылки. Таким образом, рассматриваемый здесь код делает нечто вполне реальное.

Если попытаться разделить данный код по видам ответственности, то в конечном итоге получится следующий перечень требований к ним.

1. Прием каждого входящего сообщения и его подача в систему.
2. Отправка почтового сообщения.
3. Формирование новых сообщений из каждого входящего сообщения исходя из списка получателей.
4. Периодическая активизация средства проверки поступившей почты.

Можно ли, глядя на этот перечень, определить, что одни виды ответственности в большей степени связаны с интерфейсом Java Mail API, чем другие? Виды ответственности 1 и 2 определенно связаны с этим прикладным интерфейсом. Сложнее дело обстоит с видом ответственности 3. Классы сообщений, которые нам требуются, являются частью почтового интерфейса API, но мы все же можем протестировать данный вид ответственности независимым образом, сформировав фиктивные входящие сообщения. Вид ответственности 4 на самом деле не имеет ничего общего с почтой. Для него просто требуется поток, активизируемый через заданные промежутки времени.

На рис. 15.1 приведена небольшая структура кода, разделяющая перечисленные выше виды ответственности.

Класс `ListDriver` (формирователь списка) управляет всей системой. Он содержит поток, который неактивен большую часть времени, но периодически активизируется для проверки почты. Для проверки почты класс `ListDriver` предписывает сделать эту проверку классу `MailReceiver` (получатель почты). Класс `MailReceiver` читает почту и по очереди направляет почтовые сообщения классу `MessageForwarder` (отправитель сообщений). Класс `MessageForwarder` формирует сообщения для каждого из адресатов из списка получателей и отправляет их, обращаясь к классу `MailSender` (отправитель почты).

Такая структура кода выглядит достаточно удачной. Интерфейсы `MessageProcessor` (обработчик сообщений) и `MailService` (почтовая служба) удобны тем, что они позволяют нам тестировать классы независимо друг от друга. В частности, довольно привлекательной представляется возможность работать с классом `MessageForwarder` в средствах тестирования, фактически не отправляя почту. Этого нетрудно добиться, если создать класс `FakeMailSender` (фиктивный отправитель почты), реализующий интерфейс `MailService`.

Практически у каждой системы имеется своя базовая логика, которую можно отделить от вызовов интерфейса API. Несмотря на то, что данный пример относительно невелик, он демонстрирует едва ли не самый худший случай из всех возможных. Класс

`MessageForwarder` относится к самой системе, а его ответственность почти полностью зависит от механизма отправки и приема почты. Тем не менее в нем используются классы сообщений из интерфейса Java Mail API. На первый взгляд, в коде данной системы не так уж много места для старых добрых классов Java. Но разделение системы на четыре класса и два интерфейса по блок-схеме, приведенной на рис. 5.1, позволяет представить ее отдельными уровнями. Основная логика списка рассылки находится в классе `MessageForwarder`, и поэтому мы можем протестировать ее. А в исходном коде она скрыта и недоступна. Данную систему практически нельзя разделить на мелкие части, не получив в итоге одни части более высокого уровня, чем другие.

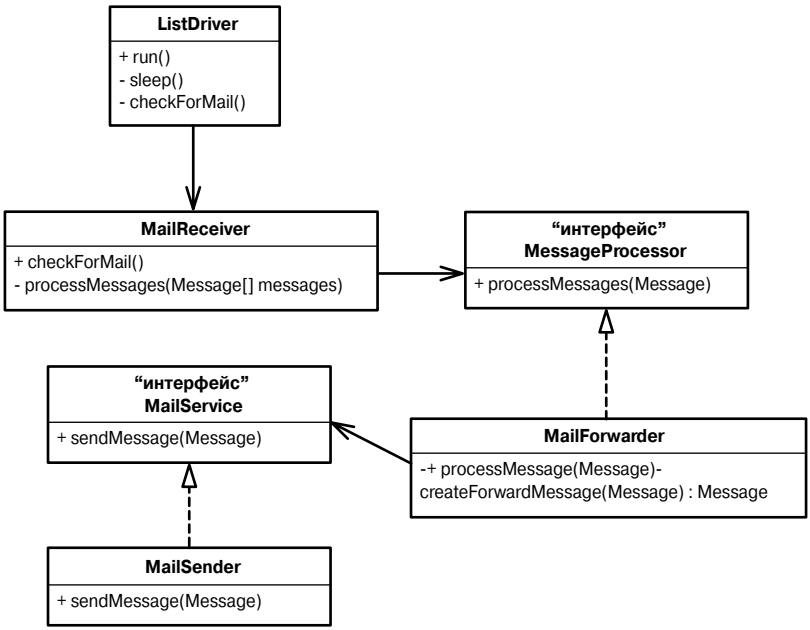


Рис. 15.1. Улучшенная структура кода сервера рассылки почты

Когда мы имеем дело с системой, которая выглядит так, как будто она состоит из сплошных вызовов интерфейса API, ее удобно представить в виде одного крупного объекта, а затем применить к ней эвристический анализ разделения ответственности, представленный в главе 20. Возможно, нам и не удастся сразу же перейти к лучшей структуре кода, но одна только возможность выявить разные виды ответственности облегчает принятие более взвешенных решений, чтобы продвинуться хотя бы на шаг к желаемой структуре.

Итак, мы выяснили, как должна выглядеть улучшенная структура кода. Конечно, приятно осознавать, что она возможна, но вернемся к суровой реальности: как нам двигаться дальше? Для этого фактически имеются два пути.

1. Заключение интерфейса API в оболочку.
2. Извлечение по видам ответственности.

Когда мы *заключаем интерфейс API в оболочку*, создаем интерфейсы, зеркально отображающие интерфейс API как можно более точно и создающие оболочки вокруг этого прикладного интерфейса. Для того чтобы свести ошибки к минимуму, мы можем *сохранить сигнатуры* по ходу работы. Преимущество заключения интерфейса API в оболочку

состоит, в частности, в том, что оно позволяет избавиться от зависимостей от базового кода API. Построенные таким образом оболочки способны делегировать свои полномочия реальному интерфейсу API в выходном коде, тогда как в тестовом коде могут быть использованы подделки.

Можем ли мы применить такой прием в приведенном ранее примере сервера рассылки почты? В этом сервере отправкой почтовых сообщений фактически занимается следующий код:

```
...
Session smtpSession = Session.getDefaultInstance (props, null);
Transport transport = smtpSession.getTransport ("smtp");
transport.connect (host.smtpHost, host.smtpUser,
    host.smtpPassword);
transport.sendMessage (forward, roster.getAddresses ());
...
```

Если бы нам потребовалось разорвать зависимость от класса `Transport`, то мы могли бы создать для него оболочку, но в данном коде мы не создаем объект класса `Transport`, а получаем его из класса `Session` (сеанс связи). А можем ли мы создать оболочку для класса `Session`? Нет, не можем, поскольку это конечный класс. В Java конечные классы не могут иметь подклассы.

Данный код почтовой рассылки не совсем подходит для заключения в оболочку. Ведь интерфейс API достаточно сложен. Но если у нас нет никаких инструментальных средств реорганизации кода, то такой путь все равно окажется самым безопасным.

К счастью, для Java имеются инструментальные средства реорганизации кода, и поэтому мы можем выбрать другой путь: *извлечение по видам ответственности*. В этом случае мы выявляем в коде разные виды ответственности и начинаем извлекать для них соответствующие методы.

Какие же виды ответственности можно выявить в приведенном выше фрагменте кода? Его основное назначение состоит в отправке почтовых сообщений. А что ему для этого требуется? Сеанс связи по протоколу SMTP и транспортировка сообщений по установленному соединению. В приведенном ниже коде представлен вид ответственности за отправку сообщений, извлеченный в отдельный метод, который затем добавлен в новый класс `MailSender`.

```
import javax.mail.*;
import javax.mail.internet.InternetAddress;
import java.util.Properties;

public class MailSender
{
    private HostInformation host;
    private Roster roster;

    public MailSender (HostInformation host, Roster roster) {
        this.host = host;
        this.roster = roster;
    }

    public void sendMessage (Message message) throws Exception {
        Transport transport
```

```
            = getSMTPSession ().getTransport ("smtp");
        transport.connect (host.smtpHost,
                  host.smtpUser, host.smtpPassword);
        transport.sendMessage (message, roster.getAddresses ());
    }

    private Session getSMTPSession () {
        Properties props = new Properties ();
        props.put ("mail.smtp.host", host.smtpHost);
        return Session.getDefaultInstance (props, null);
    }
}
```

Как же нам сделать выбор между заключением интерфейса API в оболочку и извлечением по видам ответственности? Ниже представлены некоторые компромиссные соображения по поводу такого выбора.

Заключение интерфейса API в оболочку подходит при следующих обстоятельствах.

- Интерфейс API относительно мал.
- Требуется полностью отделить зависимости от сторонней библиотеки.
- Тесты отсутствуют, а написать их нельзя, поскольку протестировать интерфейс API невозможно.

Заключая интерфейс API в оболочку, мы получаем возможность подвергнуть тестированию весь наш код, за исключением тонкого слоя делегирования полномочий от оболочки к классам реального интерфейса API.

Извлечение по видам ответственности подходит при следующих обстоятельствах.

- Интерфейс API относительно сложен.
- Имеется инструментальное средство, обеспечивающее безопасное извлечение методов, или же достаточная уверенность в безопасности извлечения вручную.

Найти золотую середину между преимуществами и недостатками обоих рассматриваемых здесь способов не так-то просто. Заключение интерфейса API в оболочку требует больших усилий, но оно может оказаться особенно полезным, если нам требуется обособиться от сторонних библиотек, а такая потребность возникает довольно часто. Подробнее об этом см. в главе 14. Когда же мы применяем извлечение по видам ответственности, в конечном счете можно извлечь логику собственного кода вместе с кодом API, а следовательно, и метод с именем более высокого уровня. В таком случае наш код будет зависеть, скорее, от интерфейсов более высокого уровня, чем от вызовов низкоуровневого интерфейса API, но тогда мы вряд ли сможем получить код, извлеченный при тестировании.

Во многих группах разработчиков применяются оба способа: тонкая оболочка — для тестирования, а оболочка более высокого уровня — для представления интерфейса, более подходящего для разрабатываемого приложения.

Глава 16

Код недостаточно понятен для его изменения

Поначалу программисты вмешиваются в неизвестный код, особенно унаследованный, с некоторой опаской, но со временем они делают это смелее. У них постепенно вырабатывается уверенность в своих силах, когда им приходится постоянно встречаться в таком коде с чудовищными созданиями и уничтожать их, хотя избавиться от подобной боязни до конца им так и не удается. Время от времени каждому программисту встречаются в унаследованном коде демонические создания, с которыми ему трудно справиться. Еще хуже, если заранее предчувствуешь свои опасения, не глядя даже на код. Ведь никогда не знаешь, сколько времени потребуется на внесение изменений в код: час или целая неделя адского труда и проклятий в адрес изменяемой системы, ситуации и всего, что находится вокруг. Если же мы хорошо понимаем, что нам нужно сделать для внесения изменений в код, то дело продвигается гораздо более гладко. Как же нам добиться такого понимания?

Рассмотрим типичную ситуацию. Допустим, что мы выяснили свойство, которое нам требуется добавить в систему. Мы садимся за рабочий стол и начинаем просматривать код. Иногда все, что нам требуется знать, удается найти очень быстро, но в унаследованном коде для этого может понадобиться время. Мы все время составляем и анализируем в уме перечень неотложных действий, подбирая один за другим разные подходы к решению поставленной задачи. В какой-то момент мы ощущаем наметившийся прогресс и достаточную уверенность в своих силах, чтобы приступить к делу. Но иногда от просмотра и анализа кода, который мы пытаемся усвоить, нам просто становится дурно. Чтение кода никак не помогает, и тогда мы просто начинаем работать с кодом, делая то, что хорошо знаем, надеясь на удачу.

Существуют и другие способы понять код, но многие программисты не пользуются ими, поскольку они слишком увлечены стремлением поскорее разобраться в коде. В конце концов долгие попытки понять код выглядят подозрительно, как отлынивание от работы. И если нам удастся довольно быстро понять код, то мы начнем оправдывать свою зарплату. Каким бы нелепым такой подход к пониманию кода ни показался, но именно так чаще всего и поступают разработчики, и это, на мой взгляд, очень прискорбно, поскольку существуют простые, технически несложные приемы, с которых можно начать работу с кодом, опираясь на них, как на прочное основание.

Составление примечаний и эскизов

Если чтение кода вызывает трудности, то имеет смысл начать составлять эскизы и делать пометки. Запишите сначала имя последнего важного объекта, который вам встретился в коде, а затем имя следующего объекта. Если вы обнаружите между ними взаимосвязь, то соедините их линией. Такие эскизы не обязательно должны быть выполнены в стиле полноценных блок-схем, составленных на языке UML, или графов вызовов функций, построенных с использованием специальных условных обозначений, хотя в ситуации с запутанным кодом формальный и аккуратный способ организации собственных мыслей на

бумаге оказывается более уместным. Составление эскизов зачастую помогает взглянуть на вещи иначе. Кроме того, это отличный способ сохранить ясность ума при попытке понять нечто особенно сложное.

На рис. 16.1 воссоздан эскиз, однажды набросанный мной и другим программистом, когда мы просматривали код. Мы набросали его на обратной стороне памятной записки (названия в этом эскизе были изменены из цензурных соображений).

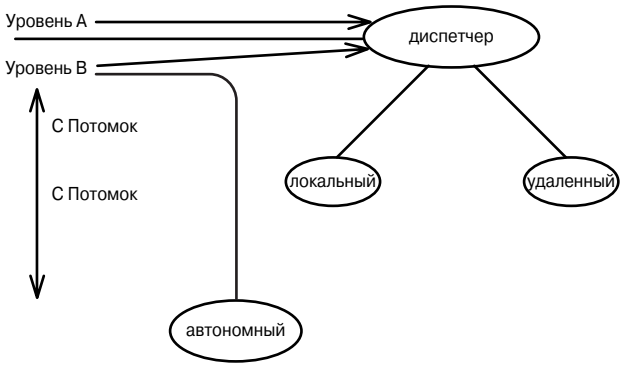


Рис. 16.1. Эскиз

Этот эскиз теперь не совсем понятен даже мне, но тогда его было достаточно для обсуждения кода. Он помог нам лучше понять код и выбрать подход к работе с ним.

Делают ли это все? И да, и нет. Немногие делают это часто. Я подозреваю, что эскизы составляются потому, что для этого не существует каких-то определенных правил, в отличие от неотвязной мысли о том, что, беря ручку и лист бумаги, мы должны непременно написать фрагмент кода или воспользоваться синтаксисом UML. Составлять блок-схемы на языке UML совсем неплохо, но не менее полезно просто рисовать кружки и проводить между ними линии, понятные только присутствующим при анализе кода. Особая точность в составлении эскизов на бумаге не требуется. Бумага и ручка служат лишь средствами, облегчающими обсуждение кода и помогающим нам запомнить понятия, которые мы обсуждаем и усваиваем.

Замечательная особенность составления эскизов отдельных частей структуры кода при попытке понять их заключается в неформальности и заразительности такого приема. Если вы найдете его полезным для себя, то не настаивайте на том, чтобы им стали пользоваться и остальные члены вашей группы в процессе работы над проектом. Вместо этого дождитесь момента, когда вам придется разбирать код вместе с одним из своих коллег, и в ходе обсуждения набросайте небольшой эскиз, поясняющий ход ваших мыслей. Если ваш коллега действительно заинтересован в том, чтобы разобраться в данной части кода, он будет невольно обращаться к эскизу вместе с вами по ходу анализа и обсуждения кода.

Как только вы начнете составлять частичные эскизы системы, у вас, скорее всего, возникнет желание понять всю картину в целом. Способы, способствующие лучшему пониманию и осмыслению крупной базы кода, рассматриваются в главе 17.

Разметка листингов

Составление эскизов — это не единственный способ, помогающий лучше понять код. Нередко применяется и другой способ, называемый *разметкой листингов*. Он особенно

полезен для усвоения длинных методов. В его основу положен простой принцип, и каждый разработчик так или иначе пользовался этим способом, хотя его, откровенно говоря, явно недооценивают.

Способ разметки листингов зависит от того, что требуется понять в коде. Сначала следует распечатать код, с которым требуется работать, а затем приступить к разметке листинга при попытке выполнить одно из приведенных ниже действий.

Разделение ответственности

Если требуется разделить виды ответственности, отметьте маркером группу соответствующих объектов. Если несколько объектов связывает нечто общее, то пометьте их специальным знаком, проставляемым рядом с ними, чтобы их легче было распознать. Старайтесь выполнять разметку разным цветом.

Уяснение структуры метода

Если требуется разобраться в крупном методе, то выровняйте блоки его кода. Зачастую отступы в длинных методах сильно затрудняют чтение их кода. Для того чтобы выровнять блоки кода, проведите линии от начала до конца каждого блока или же снабдите конец блока комментариями в виде исходного текста цикла или же условия, с которого этот блок начинается.

Блоки проще всего выравнивать изнутри наружу. Например, работая с кодом, написанным на одном из языков семейства C, начните его чтение с верхней части листинга, пропуская каждую открывающую фигурную скобку и продолжая до тех пор, пока вы не достигните первой закрывающей фигурной скобки. Пометьте эту закрывающую скобку, а затем вернитесь назад и пометьте соответствующую ей открывающую скобку. Продолжайте чтение кода до тех пор, пока не достигните следующей закрывающей фигурной скобки, и повторите описанные выше действия, вернувшись назад к соответствующей открывающей скобке.

Извлечение методов

Если требуется разделить на части крупный метод, обведите кружком код, который нужно извлечь. Пометьте код его связующим числом (подробнее о связующем числе речь пойдет в главе 22).

Уяснение воздействий в связи с изменением

Если требуется понять влияние определенного изменения, которое вы собираетесь внести в код, то вместо составления *эскиза воздействий* пометьте строки кода, которые вам предстоит изменить. Затем пометьте каждую переменную, значение которой может измениться в результате такого изменения, а также каждый вызов метода, на который оно может оказать влияние. Далее пометьте переменные и методы, на которые могут воздействовать помеченные объекты. Повторите эти действия столько раз, сколько требуется, чтобы выяснить, каким образом воздействия распространяются от места внесения изменения. Делая это, вы сможете лучше понять, что именно вам следует тестировать.

Черновая реорганизация кода

Одним из самых лучших способов изучения кода является его реорганизация. Для этого просто выберите требуемый код и начните перемещать его элементы, чтобы сделать код яснее. Единственный недостаток такой реорганизации кода заключается в том, что если у вас нет подходящих тестов, то выполнять ее очень опасно. Ведь вы не знаете, нарушите ли вы что-нибудь, пытаясь понять код с помощью такой реорганизации? На самом деле вы можете работать, даже не думая ни о чем подобном, и сделать это очень просто. Отметьте код в системе контроля версий. Забудьте о написании тестов. Извлекайте методы, переносите переменные, реорганизуйте код как угодно, чтобы лучше его понять, — только не проверяйте код вновь, а отбросьте его. В этом и заключается вся суть *черновой реорганизации кода*.

Когда я впервые упомянул о таком способе своему коллеге по работе, он посчитал поначалу, что это напрасная трата времени, но благодаря подобной реорганизации кода, с которым мы работали, нам удалось буквально за полчаса узнать о нем очень много. После этого мой коллега увлекся этой идеей.

Черновая реорганизация кода — отличный способ добраться до сути и узнать по-настоящему, как код работает, хотя это и рискованное занятие по двум причинам. Во-первых, мы совершаем большую ошибку, когда в результате реорганизации кода приходим к выводу, что система делает не то, что должна делать. В этом случае у нас возникает неверное представление о системе, что может привести к осложнениям, когда мы приступим к настоящей реорганизации кода. И во-вторых, мы можем настолько пристраститься к такой реорганизации кода, что начнем постоянно рассматривать его именно с этих позиций. Разумеется, это не обязательно плохо, но добром может и не кончиться. По целому ряду причин структура кода может и не оказаться такой же самой, когда мы, наконец, дойдем до настоящей реорганизации кода. В частности, можно найти более совершенный способ структуризации кода. Кроме того, наш код может со временем измениться, а вместе с ним и наше представление о нем. Если же мы слишком привязаны к конечной цели черновой реорганизации кода, то можем просто упустить из виду все эти представления о коде.

Черновая реорганизация кода — неплохой способ убедить себя в том, что наиболее важные части кода вам понятны, а одно только это способно упростить работу с кодом. Самое главное, что вы обретаете уверенность в своих силах и не испытываете боязни от того, что вас ожидает что-то ужасное буквально на каждом шагу, а если и ожидает, то вы, по крайней мере, сможете вовремя это заметить.

Удаление ненужного кода

Если код, который вы анализируете, оказывается запутанным и вы решите, что некоторая его часть не нужна, то удалите ее. Ненужный код ничего полезного не делает, а только мешает.

Иногда удаление ненужного кода считается напрасной тратой времени. Ведь кто-то потратил время, чтобы написать этот код, а значит, он может для чего-нибудь пригодиться. Для этой цели и служит система контроля версий. Такой код будет находиться в предыдущих версиях, и если он понадобится, то его можно всегда найти.

Глава 17

У приложения отсутствует структура

Долговечные приложения постепенно разрастаются. Они могут начаться с хорошо продуманной архитектуры, но со временем в силу производственной необходимости они достигают такой стадии своего развития, что никто не в состоянии полностью понять их структуру. Разработчики могут годами работать над проектом и не иметь никакого представления о том, где предполагается ввести новые свойства. Они знают лишь о недавних усовершенствованиях системы. Когда они вводят новые свойства, то переходят к "точкам усовершенствований", поскольку именно эти места они знают лучше всего.

Простого решения этой проблемы не существует, а крайняя необходимость диктуется конкретной ситуацией. В некоторых случаях программисты просто оказываются в безвыходном положении. Им трудно ввести новые свойств, что отрицательно сказывается на работе всей организации. Программистам приходится делать нелегкий выбор между перепроектированием системы и переписыванием ее кода. В других организациях системы кое-как, но все же работают годами. Конечно, для ввода новых свойств в такие системы потребуется больше времени, но это уже, так сказать, производственные издержки. Ведь никто не знает, насколько лучше могла бы быть структура системы или же насколько велики убытки из-за ее неудачной структуры.

Если разработчики не знают архитектуру системы, то она постепенно деградирует. А что мешает им знать ее?

- Система может быть настолько сложной, что для выяснения полной картины ее работы потребуется немало времени.
- Система может быть настолько сложной, что получить полную картину ее работы невозможно.
- Разработчики работают в "пожарном режиме", устраняя одну аварийную ситуацию за другой, постепенно теряя всякое представление о работе всей системы в целом.

По традиции для решения подобных проблем во многих организациях прибегали к услугам разработчиков архитектуры системы. Как правило, задача разработчика архитектуры — выработать общую картину работы системы и принять ряд решений, чтобы сохранить эту картину для группы разработчиков системы. Такой подход вполне работоспособен, но с одной оговоркой. Разработчик архитектуры должен постоянно работать с группой разработчиков системы, иначе код не будет отражать общую картину. Это может произойти двумя путями — кто-то мог сделать в коде что-то неуместное или же общая картина могла измениться сама. В самом худшем случае, с которым мне приходилось сталкиваться в своей практике, у разработчика архитектуры складывалось совершенно иное представление о системе, чем у программистов. Такое нередко случается из-за того, что у разработчика архитектуры иные обязанности и он не может вникать в код или общаться с программистами достаточно часто, чтобы быть в курсе дела. В результате теряется связь со всей организацией в целом.

Сермяжная правда заключается в том, что архитектура системы слишком важна, чтобы быть уделом немногих. Конечно, к проектированию системы неплохо привлечь разработ-

чика архитектуры, но для сохранения в неприкосновенности архитектуры системы очень важно, чтобы каждый ее разработчик знал, что собой представляет архитектура системы, и был кровно заинтересован в ее сохранении. Всякий, кто соприкасается с кодом, должен знать архитектуру системы и всякий другой, кто также соприкасается с кодом, должен извлекать пользу из этого знания. Если все члены группы разработчиков выработают одинаковое представление о системе и ее архитектуре, то совокупное знание группы о системе расширится. Так, если из всех 20 членов группы разработчиков только трое подробно знают архитектуру проектируемой системы, то возможны два варианта — либо этим троим придется приложить немало труда, чтобы остальные 17 разработчиков были постоянно в курсе дела, либо остальные 17 разработчиков будут постоянно совершать ошибки из-за отсутствия правильного представления о работе всей системы в целом.

Как же составить общую картину работы крупной системы? Это можно сделать разными способами. В частности, способы решения данного вопроса подробно изложены в книге *Object-Oriented Reengineering Patterns* (Шаблоны объектно-ориентированного перепроектирования) Сержа Демейе (Serge Demeyer), Стефана Дюкасса (Stephane Ducasse) и Оскара М. Нирштраса (Oscar M. Nierstrasz), Morgan Kaufmann Publishers, 2002. А в этой главе описывается ряд других не менее эффективных способов. Если часто применять их в практике группового проектирования систем, то вопросы архитектуры будут постоянно в центре внимания всей группы разработчиков системы, а это, вероятно, самое главное для сохранения ее архитектуры. Ведь очень трудно уделять внимание тому, о чем думаешь нечасто.

Описание системы

Когда я работаю с группами разработчиков, часто прибегаю к приему, который я называю “описанием системы”. Для этого требуются два человека. Один из них начинает с вопроса: “Что собой представляет архитектура системы?”, задаваемого другому. А другой пытается объяснить архитектуру системы, пользуясь лишь несколькими понятиями, возможно, двумя или тремя. Если вы оказываетесь в роли объясняющего, то должны учесть, что задавшему этот вопрос якобы ничего неизвестно о системе. В нескольких словах вы должны попытаться описать отдельные части системы и их взаимодействие. После этого вы должны подчеркнуть наиболее важные, на ваш взгляд, свойства системы. Далее опишите вторые по степени важности свойства системы, и так далее до тех пор, пока вы не сообщите все важные факты о базовой структуре системы.

Как только вы начнете описывать систему, у вас возникнет странное ощущение. Для того чтобы кратко описать архитектуру системы, вам приходится упрощать. Так, вы можете сказать: “Шлюз получает наборы правил из активной базы данных”, но, услышав это, ваш собеседник может воскликнуть: “Не так. Шлюз получает наборы правил не только из активной базы данных, но и также из текущего рабочего множества”. Когда вы пытаетесь описывать систему кратко, то чувствуете, то что-то не договариваете. Но ведь вы стараетесь дать простое описание системы, чтобы легче было понять ее архитектуру. Например, почему шлюз должен получать наборы правил из нескольких мест? Не проще ли объединить их?

Из прагматических соображений люди привыкли все усложнять, но определенный смысл есть и в подчеркивании простого взгляда на вещи. По крайней мере, это помогает каждому понять, каким должен быть идеал и что следует отнести к практической целесообразности. Еще одна важная особенность данного способа состоит в том, что он заставляет собеседников по-настоящему задуматься о том, что важнее всего в системе и что важно сообщить о ней?

Группы разработчиков могут дойти только до этой черты, когда система, над которой они работают, представляет для них сплошную тайну. Как ни странно, простое описание

работы системы служит своеобразным ориентиром при поиске подходящих мест для ввода новых свойств. Кроме того, оно делает саму систему намного менее устрашающей.

Почаще описывайте систему своим коллегам по работе, чтобы составить общее представление о ней. Описывайте ее по-разному, подбирая подходящие понятия по степени их важности. Когда вы станете обдумывать изменения в системе, заметите, что одни изменения в большей степени соответствуют описанию системы, чем другие, т.е. они делают более краткое описание системы больше похожим на правду. Если вам приходится выбирать между двумя способами сделать что-то с системой, то ее описание поможет вам лучше понять, каким из этих способов система станет более понятной.

Рассмотрим применение упомянутого выше способа описания системы на конкретном примере. Ниже приведено обсуждение средства тестирования JUnit. При этом подразумевается, что вы хотя бы немного знакомы с архитектурой JUnit. В противном случае уделите время ознакомлению с исходным кодом JUnit, который может быть загружен по адресу `www.junit.org`.

- **Что собой представляет архитектура JUnit?**
- В JUnit имеются два основных класса. Первый класс называется `Test`, а второй — `TestResult`. Пользователи создают и выполняют тесты, передавая их классу `TestResult`. Если тест не проходит, то он сообщает об этом классу `TestResult`. Затем пользователи могут запросить у класса `TestResult` сведения обо всех сбоях.

Приведем следующий перечень упрощений.

1. В JUnit имеется и много других классов. Классы `Test` и `TestResult` названы мной основными только потому, что я так думаю. На мой взгляд, их взаимодействие составляет основу работы всей системы. У других по этому поводу может сложиться иное, в равной степени верное мнение об архитектуре JUnit.
2. Пользователи не создают тестовые объекты. Последние создаются в классах контрольных примеров с помощью рефлексии.
3. `Test` является не классом, а интерфейсом. Тесты, выполняемые в JUnit, обычно пишутся в подклассах класса, называемого `TestCase` и реализующего интерфейс `Test`.
4. Пользователи обычно не запрашивают у класса `TestResult` сведения о сбоях. Класс `TestResult` регистрирует прослушивающие процессы, которые уведомляются всякий раз, когда класс `TestResult` получает сведения от теста.
5. Тесты сообщают не только о сбоях, но и о количестве выполненных тестов и ошибок. (Ошибки — это осложнения, возникающие в тесте и не проверяемые явно. А сбои — это неудавшиеся проверки.)

Дают ли эти упрощения хоть какой-то намек на возможность упростить архитектуру JUnit? Очень слабый. В ряде более простых версий среды тестирования xUnit создается класс `Test`, а класс `TestCase` опускается. В других средах тестирования ошибки и сбои объединяются, и поэтому о них сообщается одинаково.

Вернемся, однако, к нашему описанию.

- **Это все?**
- Нет, не все. Тесты могут быть сгруппированы в объекты, называемые тестовыми наборами. Мы можем выполнить тестовый набор как одиночный тест. Все тесты, выполняемые в таком наборе, сообщают нам результат, когда они не проходят.

Какие упрощения имеются в данном случае?

1. Объекты тестовых наборов (класса `TestSuite`) не только содержат и выполняют ряд тестов, но и создают с помощью рефлексии экземпляры объектов, производных от классов `TestCase`.
2. Из предыдущего упрощения вытекает еще одно упрощение: тесты сами не выполняются. Они передаются классу `TestResult`, который, в свою очередь, вызывает выполняющий тесты метод в самом тесте. Это происходит то и дело на довольно низком уровне. Такое упрощение представляется удобным. И хотя оно не совсем соответствует действительности, именно так это обычно и происходило, когда среда тестирования JUnit была проще.

- **Это все?**
- Нет, не все. На самом деле `Test` является интерфейсом. Существует класс, называемый `TestCase` и реализующий интерфейс `Test`. Пользователи выполняют сначала подклассификацию класса `TestCase`, а затем пишут собственные тесты в виде общедоступных пустых методов, начинающихся со слова `test` в своем подклассе. В классе `TestSuite` используется рефлексия для построения группы тестов, которые могут быть выполнены единственным вызовом метода `run` из класса `TestSuite`.

Мы могли бы продолжить описание, но приведенного выше достаточно, чтобы дать ясное представление о рассматриваемом здесь способе. Сначала мы даем краткое описание системы. Когда мы упрощаем описание системы, скрывая подробности, то на самом деле прибегаем к абстракции. Вынуждая себя давать очень простое представление системы, мы зачастую находим новые абстракции.

Если система оказывается не такой простой, как ее описание, то так ли это плохо? Нет, конечно. По мере своего разрастания системы неизбежно становятся сложнее. А их описание служит для нас определенным ориентиром.

Допустим, что мы собираемся ввести в JUnit новое свойство, чтобы эта среда тестирования формировала отчет обо всех тестах, которые не вызывают никаких утверждений, когда мы их выполняем. Какие варианты у нас есть с учетом того, что описано в JUnit?

Один вариант состоит в том, чтобы добавить в класс `TestCase` метод под названием `buildUsageReport` (сформировать отчет об использовании), выполняющий каждый метод и формирующий отчет обо всех методах, которые не вызывают метод `assert`. Удачный ли это способ ввода нового свойства? Как это скажется на нашем описании системы? Пожалуй, это приведет к очередной недосказанности из-за пропусков в нашем предельно кратком описании системы, которое приводится ниже еще раз.

- В JUnit имеются два основных класса. Первый класс называется `Test`, а второй — `TestResult`. Пользователи создают и выполняют тесты, передавая их классу `TestResult`. Если тест не проходит, то он сообщает об этом классу `TestResult`. Затем пользователи могут запросить у класса `TestResult` сведения обо всех сбоях.

По-видимому, у объектов класса `Test` появилась теперь совершенно другая ответственность: они формируют отчеты, о которых мы даже не упоминали в своем описании.

А что, если ввести новое свойство другим способом? В частности, мы могли бы изменить сначала взаимодействие классов `TestCase` и `TestResult`, чтобы в классе `TestResult` учитывалось количество утверждений, выполненных при каждом прогоне тестов, а затем создать класс для формирования отчетов и зарегистрировать его в классе `TestResult` в качестве прослушивающего процесса. Как это повлияет на описание системы? Это могло бы послужить веским основанием для незначительного обобщения: объекты класса `Test` сообщают объектам класса `TestResult` не только о количестве сбоев, но и о количестве

ошибок, числе выполненных тестов и утверждений. Мы могли бы изменить наше краткое описание следующим образом.

- В JUnit имеются два основных класса. Первый класс называется `Test`, а второй — `TestResult`. Пользователи создают и выполняют тесты, передавая их классу `TestResult`. Если тест выполняется, то сообщает об этом классу `TestResult`. Затем пользователи могут запросить у класса `TestResult` сведения обо всех выполнявшихся тестах.

Насколько такое описание лучше? Откровенно говоря, мне больше нравится первоначальное описание с регистрацией сбоев. На мой взгляд, это один из основных видов поведения JUnit. Если мы изменяем код таким образом, чтобы объекты класса `TestResult` регистрировали количество выполненных утверждений, то по-прежнему что-то скрываем, хотя и представляем в лучшем свете другие сведения о результатах выполнения тестов. С другой стороны, возложение на класс `TestCase` ответственности за выполнение ряда контрольных примеров и формирование отчета о них было бы еще более явным сокрытием истины, поскольку мы вообще ничего не сказали в нашем описании об этой дополнительной ответственности класса `TestCase`. Было бы лучше, если бы тесты сами сообщали о количестве выполненных утверждений при их прогоне. Наш первый вариант описания оказался чуть более обобщенным, но, по крайней мере, он, в основном, правдив. Это означает, что наши изменения в большей степени согласуются с архитектурой системы.

Открытое событийное взаимодействие классов

На заре развития ООП многим программистам приходилось с трудом перестраиваться на новые принципы проектирования. Ведь очень трудно было воспринять объектно-ориентированный подход к проектированию, имея за плечами опыт программирования на процедурных языках. Иными словами, нужно было совсем иначе представлять себе код. Я помню, как мне впервые попытались наглядно показать на бумаге принцип объектно-ориентированного проектирования. Мне показывали блок-схемы и давали устные пояснения, но у меня постоянно вертелся в голове вопрос: “А где же функция `main()`?” Где же точка входа для всех этих новых объектов? Поначалу я был просто сбит с толку, по постепенно до меня стало доходить. С подобными трудностями столкнулся не только я, но и многие программисты, переходившие на ООП. И это происходит до сих пор со всяким, кому приходится впервые иметь дело с объектно-ориентированным кодом.

В 1980-е годы решением этой проблемы занялись Уард Каннингхэм и Кент Бек. Они попробовали помочь программистам научиться мыслить о проектировании категориями объектов. В то время Уард Каннингхэм пользовался инструментальным средством под названием Hypercard, позволявшим создавать карточки на компьютерном мониторе и формировать связи между ними. И вдруг его осенило: почему бы не воспользоваться настоящими учетными карточками для представления классов? Это сделало бы более понятным и простым обсуждение классов. Так, если нам требуется обсудить класс `Transaction`, мы выбираем карточку с видами его ответственности и взаимодействующими с ним классами.

Событийное взаимодействие классов (Class, Responsibility, Collaboration — CRC) действует по следующему принципу: мы заносим на отдельную карточку имя каждого класса, виды его ответственности и список взаимодействующих с ним классов (т.е. других классов, с которым сообщается данный класс). Если мы считаем, что конкретный вид ответственности не принадлежит отдельному классу, то вычеркиваем его из карточки данного класса и заносим на карточку другого класса или же создаем для него новую карточку.

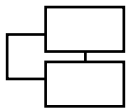
Несмотря на то, что событийное взаимодействие классов нашло на какое-то время широкое распространение, в конечном итоге наметилась тенденция пользоваться, главным образом, блок-схемами. Практически у каждого обучавшего ООП имелась своя система условных обозначений классов и взаимосвязей между ними. Но в конце концов различные условные обозначения были обобщены совместными многолетними усилиями в виде языка UML, и тогда многие посчитали, что все вопросы проектирования системы уже разрешены. Он стали рассматривать условное обозначение как метод, а UML — как средство для разработки систем: составить сначала множество блок-схем, а затем написать код. Но со временем они осознали, что, несмотря на все преимущества UML как средства условного обозначения для документирования систем, это далеко не единственный способ проработки замыслов при построении систем. На момент написания этой книги мне был известен намного более совершенный способ обсуждения проекта в группе разработчиков. Некоторые мои коллеги называют этот способ открытым событийным взаимодействием классов, поскольку он, в отличие от традиционного способа событийного взаимодействия классов, не требует записи на карточки. К сожалению, описать его словами не так-то просто, но я все же попытаюсь это сделать.

Несколько лет назад я познакомился на одной из конференций с Роном Джеффрисом. Он обещал мне показать, как он поясняет архитектуру системы, используя карточки таким образом, чтобы взаимодействия в ней выглядели наглядными и легко запоминающимися. И он, конечно, сдержал свое обещание. Вот как это делается. Тот, кто описывает систему, пользуется стопкой пустых учетных карточек, выкладывая их одну за другой на стол. Он может перемещать карточки, указывать на них и делать с ними все, что требуется для описания типичных объектов и их взаимодействия в системе.

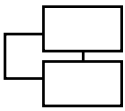
Рассмотрим для примера описание системы голосования в оперативном режиме. Ниже приведены пояснения того, кто описывает данную систему, и его действия, указываемые в скобках.

"Вот как работает система голосования в реальном масштабе времени. Это сеанс связи с клиентом" (указывает на карточку).

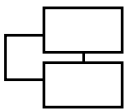
"В каждом сеансе связи устанавливаются два соединения — входящее и исходящее" (накладывает каждую карточку на первоначальную и указывает на них по очереди).

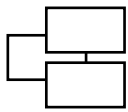


"При запуске системы организуется сеанс связи на сервере вот здесь" (выкладывает карточку справа).



"На каждый сеанс связи с сервером также приходятся два соединения" (выкладывает две карточки, обозначающие соединения справа).





“Когда начинается сеанс связи с сервером, регистрируется администратор голосования” (выкладывает карточку выше карточек, обозначающих сеанс связи с сервером).

“У нас может быть множество сеансов связи на стороне сервера” (выкладывает еще ряд карточек для обозначения нового сеанса связи с сервером и его соединений).

“Когда клиент голосует, его голос передается сеансу связи на стороне сервера” (показывает руками взаимосвязь одного из соединений в сеансе связи на стороне клиента с соединением в сеансе связи на стороне сервера).

“Сеанс связи на стороне сервера отвечает подтверждением и регистрирует голос с помощью администратора голосования” (указывает сначала в обратном направлении от сеанса связи на стороне сервера к сеансу связи на стороне клиента, а затем от сеанса связи на стороне сервера к администратору голосования).

“После этого администратор голосования предписывает сеансу связи на стороне сервера сообщить сеансу связи на стороне клиента число голосов” (указывает в направлении от администратора голосования к каждому сеансу связи на стороне клиента по очереди).

Безусловно, этому описанию явно недостает наглядности. Ведь в книге невозможно показать, каким образом карточки перемещаются и указываются, как если бы мы с вами сидели рядом за столом. Тем не менее такой способ описания системы довольно эффективен. Он позволяет представить отдельные части системы в виде вполне осязаемых предметов. Вместо карточек вы можете воспользоваться любыми другими подходящими для этой цели предметами. Самое главное, что, располагая и перемещая их, вы можете наглядно показать, каким образом взаимодействуют отдельные части системы. Такой прием нередко упрощает понимание сложных вещей, помогает разобраться в запутанных ситуациях, а также способствует лучшему запоминанию отдельных структур.

Применяя способ открытого событийного взаимодействия классов, следует руководствоваться лишь двумя принципами.

1. Карточки обозначают экземпляры, а не классы.
2. Накладывающиеся карточки обозначают совокупность экземпляров.

Особое внимание к обсуждению проекта

Во время работы с унаследованным кодом нередко возникает искушение избегать создания абстракций. Когда я анализирую четыре или пять классов, каждый из которых состоит из тысяч строк кода, я пытаюсь, скорее, выяснить, что именно мне нужно изменить, чем думаю о новых классах.

Отвлекаясь подобным образом на выяснение одних вещей, мы зачастую упускаем из виду другие вещи, способные дать нам пищу для дополнительных идей. Обратимся к конкретному примеру. Однажды мне пришлось работать с несколькими членами группы разработчиков, получивших задание создать довольно крупный фрагмент кода, исполняемого в нескольких потоках. Код был довольно сложный, и поэтому существовало несколько возможностей для его зависания. Мы осознавали, что если бы нам удалось гарантировать блокировку и разблокировку ресурсов в определенном порядке, то это позволило бы нам избежать зависания кода. Для того чтобы сделать это возможным, мы стали искать способ видоизменить код. По ходу дела мы обсуждали новое правило блокировки и пытались выяснить, как сохранить результаты подсчета в массивах. Когда же другой член группы начал вставлять код правил блокировки в исходный текст, я сказал: “Погоди, мы ведь обсуждаем правила блокировки, правда? Почему бы нам не создать класс под названием `LockingPolicy` и сохранить результаты подсчета именно в нем? Мы можем использовать имена методов, наглядно описывающие то, что мы пытаемся сделать, и это было бы понятнее, чем код, заполняющий массивы результатами подсчета”.

Самое ужасное, что эта группа была достаточно опытной. В базе кода были и другие удачные места, но крупные фрагменты процедурного кода буквально приковывали к себе внимание: они так и напрашивались на улучшение.

Внимательно слушайте обсуждение проекта. Насколько понятия, употребляемые при обсуждении, соответствуют понятиям, используемым в коде? Конечно, не все они должны соответствовать один другому. Программное обеспечение должно удовлетворять более строгим ограничениям, чем те, что проще обсуждать, но если отсутствует строгое соответствие между предметом обсуждения и кодом, то следует задаться вопросом: почему это происходит? Ответ на этот вопрос может состоять из определенного сочетания следующих двух вариантов: либо код не удалось адаптировать для полного понимания группой разработчиков, либо группе нужно понять его иначе. В любом случае очень полезно чутко воспринимать понятия, которыми участники обсуждения обычно оперируют, описывая структуру кода. Кода разработчики обсуждают проект, они пытаются высказываться так, чтобы он стал понятным другим. Именно такое понимание и следует воплотить в коде.

В этой главе рассмотрены два способа раскрытия и описания архитектуры крупных систем, которые уже существуют. Для выработки проектных решений при разработке новых систем имеется немало других вполне подходящих способов. Но проектирование остается таковым независимо от стадии, на которой оно выполняется. Одной из самых непростительных ошибок, которые может совершить группа разработчиков, является возникающее у нее на определенной стадии ощущение, что проектирование уже завершено. Если оно “завершено”, а изменения по-прежнему вносятся, то вполне возможно, что качественный новый код окажется не в тех местах, где нужно, а классы распухнут от кода, поскольку никто из разработчиков не считает удобным вводить новые абстракции. И это самый верный способ сделать унаследованный код еще хуже.

Глава 18

Когда тестовый код мешает

Когда блочные тесты приходится писать впервые, такое занятие может показаться не совсем естественным. У программистов нередко возникает ощущение, будто тесты им только мешают. Просматривая свой проект, они иногда просто забывают, что именно они анализируют — тестовый или же выходной код. От такого ощущения не помогает избавиться даже большое количество тестового кода. Поэтому, чтобы не увязнуть в нем, следует установить определенные условные обозначения.

Условные обозначения имен классов

Прежде всего следует установить условные обозначения для имен классов. Как правило, на каждый проверяемый класс приходится, по крайней мере, один класс из блочного теста, поэтому целесообразно сделать имя последнего разновидностью имени первого. Для этой цели обычно используются два вида условных обозначений. Самым распространенным из них является применение слова `Test` в качестве префикса или суффикса в имени класса. Так, если имеется класс под названием `DBEngine` (процессор базы данных), то тестовый класс можно назвать как `TestDBEngine` или же как `DBEngineTest`. Какой из этих вариантов предпочтительнее? Это не имеет особого значения. Я лично предпочитаю пользоваться суффиксом `Test`. Если вы пользуетесь интегрированной средой разработки, в которой классы перечисляются в алфавитном порядке, то каждый класс выравнивается в ряд с соответствующим тестовым классом, что упрощает переход между ними.

Какие еще классы появляются при тестировании? Нередко оказывается полезным создавать в пакете или каталоге фиктивные классы, подделывающие взаимодействие проверяемого класса с другими классами. В качестве условного обозначения фиктивных классов рекомендуется префикс `Fake` (подделка). Благодаря этому фиктивные классы располагаются в алфавитном порядке при их просмотре в браузере и на некотором отдалении от основных классов в пакете, что очень удобно, поскольку фиктивные классы зачастую являются подклассами соответствующих классов в других каталогах.

Еще одной разновидностью класса является *тестирующий подкласс*, нередко применяемый при тестировании. Тестирующим подклассом называется класс, специально создаваемый для тестирования отдельного класса, но у него имеются определенные зависимости, которые требуется отделить. Именно такой подкласс создается при использовании *подклассификации и переопределения метода*. В качестве условного обозначения для тестирующего подкласса рекомендуется использовать префикс `Testing` (тестирующий). Если классы перечисляются в пакете или каталоге в алфавитном порядке, то все тестирующие подклассы группируются вместе.

Ниже приведен пример содержимого каталога для небольшого пакета бухгалтерского учета.

- `CheckingAccount`
- `CheckingAccountTest`
- `FakeAccountOwner`
- `FakeTransaction`
- `SavingsAccount`
- `SavingsAccountTest`
- `TestingCheckingAccount`
- `TestingSavingsAccount`

Обратите внимание на то, что рядом с каждым выходным классом находится тестовый класс. А фиктивные классы и тестирующие подклассы располагаются отдельными группами.

Такую организацию классов не следует принимать как догму. Она оказывается удобной во многих случаях, хотя имеется немало причин для применения многочисленных ее разновидностей. Главным критерием при этом служит простота перехода от проверяемых классов к тестам.

Расположение тестов

До сих пор в этой главе предполагалось, что тестовый и выходной коды располагаются в одном и том же каталоге. Как правило, это самый простой способ структурирования проекта, хотя, выбирая именно такой способ расположения, следует принимать во внимание и другие обстоятельства.

Прежде всего следует принять во внимание ограничения на масштабы развертывания приложения. Так, для приложения, выполняемого на сервере, такие ограничения могут отсутствовать. Если увеличение масштаба развертывания приложения практически в два раза, включая двоичные файлы выходного и тестового кода, оказывается вполне допустимым, то и тот, и другой код проще всего хранить в том же каталоге, где и разворачиваемые двоичные файлы.

С другой стороны, если программное обеспечение является коммерческим продуктом и выполняется на чужом компьютере, то масштаб его развертывания, возможно, придется ограничить. В частности, можно попробовать хранить весь тестовый код отдельно от выходного кода, но в то же время учесть, насколько это повлияет на удобство перемещения в коде.

Иногда это не имеет особого значения, как в приведенном ниже примере. В Java пакет может располагаться в двух разных каталогах.

```
source
     com
          orderprocessing
               dailyorders

test
      com
           orderprocessing
                dailyorders
```

Выходные классы могут быть размещены в каталоге `dailyorders` (ежедневные заказы), входящем в каталог `source` (исходный код), а тестовые классы — в каталоге `dailyorders`, входящем в каталог `test` (тестовый код), причем они будут доступны как содержимое одного и того же пакета. В некоторых интегрированных средах разработки классы из этих двух каталогов показываются в одном и том же представлении, и благодаря этому не нужно особенно беспокоиться, где они расположены физически.

Во многих языках программирования и интегрированных средах разработки расположение кода имеет значение. Если для перехода от тестового кода к выходному и обратно приходится постоянно перемещаться между каталогами, то такое перемещение обычно связано с дополнительными затратами времени и труда. Обычно это приводит к тому, что разработчики просто перестают писать тесты, а их работа над проектом в итоге замедляется.

В качестве альтернативы рекомендуется хранить тестовый и выходной код в одном и том же каталоге, но пользоваться сценариями или настройками компоновки для исключения тестового кода из развертывания. Этого может оказаться достаточно, если используются удобные условные обозначения.

Но самое главное, что для выбора варианта разделения тестового и выходного кода непременно должны быть веские основания. Ведь очень часто разработчики разделяют код исключительно из эстетических соображений, поскольку им просто претит мысль о совместном расположении тестового и выходного кода. Но это затрудняет перемещение по проекту. К наличию тестов рядом с проверяемым исходным кодом нетрудно привыкнуть, а со временем это уже становится нормой.

Глава 19

Как благополучно изменить процедурный код

Название этой главы может показаться несколько провокационным. Изменения можно благополучно вносить в код, написанный на любом языке, но в одних языках это делается проще, чем в других. Несмотря на широкое распространение ООП, имеются и другие языки и способы программирования, в том числе языки продукционного, функционального, ограничительного программирования. Но среди них наибольшее распространение нашли старые добрые процедурные языки программирования, включая C, COBOL, Fortran, Pascal и BASIC.

Унаследованный код, написанный на процедурных языках, вызывает особые трудности. Такой код очень важно протестировать, прежде чем видоизменять его, но для внедрения блочных тестов в процедурные языки имеется не так уж и много возможностей. Нередко бывает проще всего тщательно обдумать изменения в коде и сделать вставку в программу, надеясь на то, что изменения окажутся правильными.

Эта дилемма тестирования носит всеобщий характер в унаследованном процедурном коде. В отличие от языков ООП и многих языков функционального программирования, процедурные языки зачастую просто не допускают появление швов в программах. Смышленые разработчики могут обойти данное препятствие, аккуратно манипулируя зависимостями в своем коде (на C, например, написано немало очень качественного кода), но и в этом случае код может очень легко получиться настолько запутанным, что его изменение постепенно или путем проверки представляется весьма затруднительным.

Если разорвать зависимости в процедурном коде настолько трудно, то самым лучшим зачастую оказывается следующий прием: сначала попытаться протестировать крупный фрагмент кода перед тем, как вносить в него какие-либо изменения, а затем использовать те же самые тесты для получения ответной реакции в ходе дальнейшей разработки. В этом могут оказать помощь способы описанные в главе 12. Они применимы как к процедурному, так и к объектно-ориентированному коду. Проще говоря, целесообразно найти сначала *точку сужения*, а затем использовать *компоновочный шов* для такого разрывания зависимостей, которого будет достаточно, чтобы ввести код в средства тестирования. Если же язык программирования, на котором написан код, поддерживает макропрепроцессор, то можно воспользоваться также *швом предварительной обработки*.

Это стандартный прием, но он не единственный. В остальной части этой главы мы рассмотрим другие приемы локального разрыва зависимостей в процедурных программах, способы упрощения проверки внесенных изменений и пути продвижения вперед при использовании языка, имеющего средства для перехода к ООП.

Простой случай

Процедурный код не всегда вызывает трудности. Ниже приведен пример функции C из операционной системы Linux. Насколько сложно было бы написать тесты для этой функции, если бы потребовались изменения в ней?

```
void set_writetime(struct buffer_head * buf, int flag)
{
    int newtime;

    if (buffer_dirty(buf)) {
        /* переместить буфер в черновой список, если переменная jiffies
           очищена */
        newtime = jiffies + (flag ? bdf_prm.b_un.age_super :
            bdf_prm.b_un.age_buffer);
        if(!buf->b_flushtime || buf->b_flushtime > newtime)
            buf->b_flushtime = newtime;
    } else {
        buf->b_flushtime = 0;
    }
}
```

Для тестирования этой функции мы можем задать значение переменной `jiffies` (миги), создать список `buffer_head` (заголовок буфера) и передать его данной функции, а затем проверить значения этой функции после ее вызова. Во многих функциях у нас такой счастливой возможности нет. Иногда одна функция вызывает другую, а та — еще одну функцию. В конце концов дело доходит до самой трудной для тестирования части кода: функции, которая выполняет ввод-вывод или же находится в сторонней библиотеке. Обычно нам нужно проверить, что именно делает код. Но зачастую оказывается, что он делает нечто такое, что выходит за рамки компетенции как самой программы, так и того, кто ее проверяет.

Тяжелый случай

Ниже приведена функция C, которую требуется изменить. Было бы неплохо протестировать эту функцию, прежде чем вносить в нее изменения.

```
#include "ksrlib.h"

int scan_packets(struct rnode_packet *packet, int flag)
{
    struct rnode_packet *current = packet;
    int scan_result, err = 0;

    while(current) {
        scan_result = loc_scan(current->body, flag);
        if(scan_result & INVALID_PORT) {
            ksr_notify(scan_result, current);
        }
        ...
        current = current->next;
    }
    return err;
}
```

В приведенном выше коде вызывается функция `ksr_notify` (уведомить о приеме и передаче данных с помощью клавиатуры), дающая неприятный побочный эффект. Она за-

писывает уведомление в стороннюю систему, причем вполне может сделать это и во время тестирования.

Данное препятствие можно, в частности, обойти, с помощью *компоновочного шва*. Если нам требуется протестировать процедурный код без вызовов функций из сторонней библиотеки, то для этой цели мы можем создать библиотеку с фиктивными функциями, называющимися так же, как исходные функции, но на самом деле ничего не делающими. В таком случае тело функции `ksr_notify` будет выглядеть следующим образом:

```
void ksr_notify(int scan_code, struct rnode_packet *packet)
{
}
```

Далее мы можем ввести эту функцию в библиотеку и прикомпоновать ее. Аналогичным образом будет вести себя и функция `scan_packets` (сканировать пакеты), за исключением следующего: она не будет отправлять уведомление. Но этого оказывается достаточно, если нам требуется выяснить другое поведение данной функции перед ее изменением.

Насколько такая методика оказывается эффективной? Это зависит от обстоятельств. Если в библиотеке `ksrlib.h` содержится много функций, а их вызовы приходится рассматривать в качестве своего рода периферии по отношению к основной логике системы, то действительно имеет смысл создать библиотеку фиктивных функций и прикомпоновать ее во время теста. С другой стороны, если нам требуется распознать эти функции или изменить некоторые возвращаемые ими значения, то пользоваться *компоновочным швом* нецелесообразно, поскольку это слишком трудоемкая процедура. Замена библиотечных функций происходит во время компоновки, и поэтому для каждого исполняемого кода, который мы компонуем, мы можем предоставить определение только одной функции. Так, если нам требуется, чтобы поведение функции `ksr_notify` было одним в одном тесте, а другим — в другом, то нам придется ввести соответствующий код в тело этой функции и задать в тесте условия, вынуждающие функцию действовать определенным образом. Но это все равно не очень приятное занятие. К сожалению, многие процедурные языки не оставляют нам иной возможности.

Но в C имеется другая возможность. В этом процедурном языке поддерживается макропрепроцессор, с помощью которого мы можем упростить написание тестов для функции `scan_packets`. Ниже приведено содержимое файла, включая функцию `scan_packets` и дополнительный код ее тестирования.

```
#include "ksrlib.h"

#ifdef TESTING
#define ksr_notify(code,packet)
#endif

int scan_packets(struct rnode_packet *packet, int flag)
{
    struct rnode_packet *current = packet;
    int scan_result, err = 0;

    while(current) {
        scan_result = loc_scan(current->body, flag);
        if(scan_result & INVALID_PORT) {
            ksr_notify(scan_result, current);
        }
        ...
```

```
        current = current->next;
    }

return err;
}

#ifdef TESTING
#include <assert.h>
int main () {
    struct rnode_packet packet;
    packet.body = ...
    ...
    int err = scan_packets(&packet, DUP_SCAN);
    assert(err & INVALID_PORT);
    ...
    return 0;
}
#endif
```

В этом коде имеется директива определения предварительной обработки (`TESTING`), по которой определяется вызов функции `ksr_notify` вне существующего кода при выполнении тестирования. Кроме того, в нем имеется небольшая заглушка, содержащая тесты.

Такое объединение в одном файле тестов и исходного кода на самом деле будет не самым ясным из возможных путей, поскольку затрудняет перемещение по коду. В качестве альтернативы можно воспользоваться директивой включения файла, чтобы расположить тестовый и выходной коды в разных файлах.

```
#include "ksrlib.h"

#include "scannertestdefs.h"

int scan_packets(struct rnode_packet *packet, int flag)
{
    struct rnode_packet *current = packet;
    int scan_result, err = 0;

    while(current) {
        scan_result = loc_scan(current->body, flag);
        if(scan_result & INVALID_PORT) {
            ksr_notify(scan_result, current);
        }
        ...
        current = current->next;
    }
    return err;
}

#include "testscanner.tst"
```

При таком изменении код получается более похожим на код без инфраструктуры тестирования. Единственное отличие состоит в том, что в данный код включен оператор `#include` в конце его файла. Если же требуется сначала определить тестируемые функции, то мы можем переместить все, что находится в нижнем включаемом файле, в верхний файл.

Для выполнения тестов достаточно определить тестирование с помощью директивы `TESTING` и отдельно скомпоновать тестовый файл (`testscanner.tst`). Если тестирование определено, то функция `main()` из тестового файла будет скомпилирована и прикомпонована к исполняемому коду, который и выполнит тесты. Функция `main()` выполняет только тестирование функций сканирования. Мы можем организовать одновременное выполнение групп тестов, определив отдельные тестирующие функции для каждого из тестов.

```
#ifdef TESTING
#include <assert.h>
void test_port_invalid() {
    struct rnode_packet packet;
    packet.body = ...
    ...
    int err = scan_packets(&packet, DUP_SCAN);
    assert(err & INVALID_PORT);
}

void test_body_not_corrupt() {
    ...
}

void test_header() {
...
}

#endif
```

В другом файле мы можем вызвать тестирующие функции из функции `main()`.

```
int main() {
    test_port_invalid();
    test_body_not_corrupt();
    test_header();

    return 0;
}
```

Более того, мы можем добавить регистрирующие функции, упрощающие группирование тестов. Подробнее об этом можно узнать, ознакомившись с различными средами тестирования на языке C, доступными по адресу `www.xprogramming.com`.

Несмотря на то, что макропрепроцессорами легко злоупотребить, в данном контексте они оказываются очень полезными. Благодаря включению файла и макрозамене мы можем обойти все препятствия, связанные с зависимостями, даже в самом запутанном коде. А поскольку мы воздерживаемся от безудержного использования макрокоманд, ограничиваясь только тестируемым кодом, то нам не нужно особенно беспокоиться о злоупотреблении макрокомандами, которое могло бы отрицательно сказаться на выходном коде.

C относится к одним из немногих основных языков программирования, поддерживающих макропрепроцессор. Как правило, для разрывания зависимостей в коде, написанном на других процедурных языках, нам приходится использовать *компоновочный шов* и пытаться подвергнуть тестированию как можно более крупные участки кода.

Ввод нового поведения

В унаследованный процедурный код целесообразно вводить новые функции, а не новый код в дополнение к старому. По крайней мере, мы можем написать тесты для новых функций.

Как же избежать препятствий, скрывающихся за введением зависимостей в процедурный код? Один из способов, представленных в главе 8, состоит в применении *разработки посредством тестирования*, пригодной как для объектно-ориентированного, так и для процедурного кода. Зачастую попытка сформулировать тест для каждого фрагмента кода, который предполагается написать, приводит к изменению его структуры в лучшую сторону. Мы сосредоточиваем сначала основное внимание на написании функций, выполняющих определенные вычисления, а затем интегрируем их в остальную часть приложения.

Для этого нам нередко приходится по-другому осмысливать то, что мы собираемся написать. Обратимся к примеру. Допустим, что нам требуется написать функцию под названием `send_command`, которая должна посылать идентификационный номер, имя и командную строку в другую систему с помощью функции, называемой `mart_key_send`. Подходящий код для этой функции может быть представлен следующим образом:

```
void send_command(int id, char *name, char *command_string) {
    char *message, *header;
    if (id == KEY_TRUM) {
        message = ralloc(sizeof(int) + HEADER_LEN + ...
        ...
    } else {
        ...
    }
    sprintf(message, "%s%s%s", header, command_string, footer);
    mart_key_send(message);

    free(message);
}
```

Но как написать тест для такой функции, особенно если выяснить происходящее можно только в точке вызова функции `mart_key_send`? Что, если мы выберем другой подход?

Мы могли бы проверить всю эту логику до вызова функции `mart_key_send`, если бы она оказалась в другой функции. Первый написанный нами тест может выглядеть следующим образом:

```
char *command = form_command(1,
                             "Mike Ratledge",
                             "56:78:cusp-:78");
assert(!strcmp("<-rsp-Mike Ratledge><56:78:cusp-:78><-rspr>",
               command));
```

Затем мы можем написать функцию `form_command`, которая возвращает сформированную команду.

```
char *form_command(int id, char *name, char *command_string)
{
    char *message, *header;
    if (id == KEY_TRUM) {
        message = ralloc(sizeof(int) + HEADER_LEN + ...
```

```
        ...
    } else {
        ...
    }
    sprintf(message, "%s%s%s", header, command_string, footer);

    return message;
}
```

Имея все это, мы можем теперь написать самую простую функцию `send_command`, которая только нам потребуется.

```
void send_command(int id, char *name, char *command_string) {
    char *command = form_command(id, name, command_string);
    mart_key_send(command);

    free(message);
}
```

Как правило, такая переформулировка оказывается именно тем, что нам требуется для продвижения вперед. Мы помещаем всю чистую логику в одном ряде функций, чтобы избавить ее от проблематичных зависимостей. Добившись этого, мы получаем в конечном итоге интерфейсные функции, подобные `send_command`, связывающие логику и зависимости. Это, конечно, не самый идеальный способ, тем не менее, он оказывается вполне работоспособным, если зависимости не слишком сильно распространены.

В других случаях приходится писать функции, испещренные внешними вызовами. В таких функциях не очень много вычислений, но в них имеет большое значение упорядочение вызовов. Так, если нам требуется написать функцию, рассчитывающую проценты по займу, то самый простой способ сделать это может выглядеть следующим образом:

```
void calculate_loan_interest(struct temper_loan *loan, int calc_type)
{
    ...
    db_retrieve(loan->id);
    ...
    db_retrieve(loan->lender_id);
    ...
    db_update(loan->id, loan->record);
    ...
    loan->interest = ...
}
```

Что делать в подобном случае? Во многих процедурных языках самый лучший выход из подобного положения — отложить на время написание теста и попробовать написать саму функцию как можно лучше. Возможно, нам удастся протестировать правильность выполнения этой функции на более высоком уровне. Но в C у нас имеется другая возможность. В C поддерживаются указатели, и поэтому мы можем воспользоваться ими для получения очередного шва в определенном месте. Вот как это делается.

Сначала мы можем создать структуру, содержащую указатели на функции.

```
struct database
{
    void (*retrieve)(struct record_id id);
```

```
    void (*update)(struct record_id id, struct record_set *record);
    ...
};
```

Далее мы можем инициализировать эти указатели на адреса функций доступа к базе данных и передать данную структуру любым новым функциям, которым требуется доступ к базе данных. В выходном коде эти функции могут указывать на реальные функции доступа к базе данных. А для тестирования мы можем сделать так, чтобы они указывали на фиктивные функции.

Для старых компиляторов нам бы, возможно, пришлось воспользоваться старым синтаксисом указателей на функции.

```
extern struct database db;
(*db.update)(load->id, loan->record);
```

А в остальных случаях мы можем вызывать функции во вполне естественном для нас объектно-ориентированно стиле.

```
extern struct database db;
db.update(load->id, loan->record);
```

Такой прием не является характерным только для C. Его можно использовать в процедурных языках, поддерживающих указатели на функции, делегаты или иные аналогичные механизмы.

Использование преимуществ ООП

В языках ООП нам доступны *объектные швы*. Такие швы обладают рядом следующих замечательных свойств.

- Они легко обнаруживаются в коде.
- Их можно использовать для разделения кода на более мелкие и понятные фрагменты.
- Они обеспечивают большую гибкость. Швы, вводимые для тестирования, могут оказаться полезными и при расширении функций программного обеспечения.

К сожалению, не все процедурные программы легко преобразуются в объектно-ориентированные, но в одних случаях сделать это оказывается проще, чем в других. Ведь многие процедурные языки постепенно эволюционировали в объектно-ориентированные. Так, язык Visual Basic компании Microsoft лишь совсем недавно стал полностью объектно-ориентированным. Существуют объектно-ориентированные расширения языков COBOL и Fortran, а большинство компиляторов C допускает также компиляцию кода C++.

Если используемый вами процедурный язык допускает переход к объектно-ориентированному программированию, то у вас появляется больше возможностей. Как правило, первый этап состоит в *инкапсуляции глобальных ссылок* для получения фрагментов кода, изменяемых при тестировании. Подобным способом можно избежать неприятной ситуации, связанной с зависимостями, аналогично рассмотренному ранее примеру функции `scan_packets`. Напомним о затруднении, связанном с функцией `ksr_notify`: для выполнения тестов уведомления не требовались.

```
int scan_packets(struct rnode_packet *packet, int flag)
{
    struct rnode_packet *current = packet;
```

```
    int scan_result, err = 0;

    while(current) {
        scan_result = loc_scan(current->body, flag);
        if(scan_result & INVALID_PORT) {
            ksr_notify(scan_result, current);
        }
        ...
        current = current->next;
    }
    return err;
}
```

Такой код следует, прежде всего, скомпилировать как код C++, а не C. Подобное изменение может быть мелким или крупным в зависимости от того, как его сделать. Так, скрепя сердце, мы можем попытаться перекомпилировать весь проект в коде C++ или же сделать это по частям, хотя и медленнее.

Если результат скомпилирован в коде C++, то мы можем приступить к поиску объявления функции `ksr_notify` и ее заключения в оболочку класса.

```
class ResultNotifier
{
public:
    virtual void ksr_notify(int scan_result,
                            struct rnode_packet *packet);
};
```

Кроме того, мы можем ввести новый исходный файл для класса и поместить в нем стандартную реализацию данной функции.

```
extern "C" void ksr_notify(int scan_result,
                           struct rnode_packet *packet);

void ResultNotifier::ksr_notify(int scan_result,
                           struct rnode_packet *packet)
{
    ::ksr_notify(scan_result, packet);
}
```

Обратите внимание на то, что мы не изменяем имя функции или ее сигнатуру. Мы используем *сохранение сигнатур*, чтобы свести к минимуму вероятность появления любых ошибок.

Далее мы объявляем глобальным экземпляр класса `ResultNotifier` (уведомитель результата) и помещаем его в исходном файле:

```
ResultNotifier globalResultNotifier;
```

Теперь мы можем перекомпилировать код и по полученным ошибкам выяснить места, где нам придется внести изменения. Мы поместили объявление функции `ksr_notify` в классе, и поэтому компилятор уже не обнаруживает ее объявление в глобальной области действия.

Ниже приведена исходная функция.

```
#include "ksrlib.h"

int scan_packets(struct rnode_packet *packet, int flag)
```

```
{
    struct rnode_packet *current = packet;
    int scan_result, err = 0;
    while(current) {
        scan_result = loc_scan(current->body, flag);
        if(scan_result & INVALID_PORT) {
            ksr_notify(scan_result, current);
        }
        ...
        current = current->next;
    }
    return err;
}
```

Для того чтобы теперь скомпилировать эту функцию, мы можем воспользоваться внешним объявлением, сделав объект `globalResultNotifier` видимым и предварив объявление функции `ksr_notify` именем данного объекта.

```
#include "ksrlib.h"

extern ResultNotifier globalResultNotifier;

int scan_packets(struct rnode_packet *packet, int flag)
{
    struct rnode_packet *current = packet;
    int scan_result, err = 0;

    while(current) {
        scan_result = loc_scan(current->body, flag);
        if(scan_result & INVALID_PORT) {
            global ResultNotifier.ksr_notify(scan_result, current);
        }
        ...
        current = current->next;
    }
    return err;
}
```

В настоящий момент код работает так, как и должен работать. Метод `ksr_notify` класса `ResultNotifier` делегирует свои полномочия функции `ksr_notify`. Но что это нам дает? Пока еще ничего. Поэтому далее нам предстоит найти какой-то способ сделать так, чтобы использовать объект `ResultNotifier` в выходном коде, а другой объект — при тестировании. Для этого существуют разные способы, но продвинуться дальше в нужном направлении нам, пожалуй, позволит повторная *инкапсуляции глобальных ссылок* и размещение функции `scan_packets` в еще одном классе, который можно назвать `Scanner`.

```
class Scanner
{
public:
    int scan_packets(struct rnode_packet *packet, int flag);
};
```

Теперь мы можем применить *параметризацию конструктора* и изменить класс `Scanner` таким образом, чтобы в нем использовался объект `ResultNotifier`, который мы предоставляем.

```
class Scanner
{
private:
    ResultNotifier& notifier;
public:
        Scanner();
        Scanner(ResultNotifier& notifier);

    int scan_packets(struct rnode_packet *packet, int flag);
};

// в исходном файле

Scanner::Scanner()
: notifier(globalResultNotifier)
{}

Scanner::Scanner(ResultNotifier& notifier)
: notifier(notifier)
{}
```

Сделав подобное изменение, мы можем найти места в коде, где используется функция `scan_packets`, создать экземпляр класса `Scanner` и вызвать из него данную функцию.

Такие изменения в коде делаются достаточно безопасно и механически. Это далеко не самые лучшие примеры объектно-ориентированного проектирования, но они вполне подходят в качестве клина, с помощью которого можно разорвать зависимости, протестировать код и двигаться дальше.

Объектно-ориентированный характер процедурного кода

Некоторые сторонники процедурного программирования любят покритиковать ООП, считая его бесполезным или слишком сложным и поэтому неоправданным. Но если вдуматься, то становится очевидно, что все процедурные программы на самом деле оказываются объектно-ориентированными, и просто непростительно, если они содержат только один объект. Для того чтобы убедиться в этом, представьте себе процедурную программу, состоящую более чем из 100 функций. Ниже приведены их объявления.

```
...

int db_find(char *id, unsigned int mnemonic_id,
            struct db_rec **rec);
...
...
void process_run(struct gfh_task **tasks, int task_count);
...
```

А теперь представьте, что мы поместили все эти объявления в один файл и окружили их объявлением класса.

```
class program
{
public:
    ...
    int db_find(char *id, unsigned int mnemonic_id,
                struct db_rec **rec);
    ...
    ...
    void process_run(struct gfh_task **tasks, int task_count);
    ...
};
```

Теперь мы можем найти определение каждой функции. Ниже приведено одно из таких определений.

```
int db_find(char *id,
            unsigned int mnemonic_id,
            struct db_rec **rec);
{
    ...
}
```

Далее мы предваряем имя функции именем класса в качестве префикса.

```
int program::db_find(char *id,
                     unsigned int mnemonic_id,
                     struct db_rec **rec)
{
    ...
}
```

И наконец, нам остается лишь написать новую функцию `main()` для программы.

```
int main(int ac, char **av)
{
    program the_program;

    return the_program.main(ac, av);
}
```

Оказывают ли подобные изменения влияние на поведение системы? Совсем нет. Эти изменения носят лишь механический характер, оставляя назначение и поведение программы точно таким же. Прежняя процедурная программа C на самом деле оказалась одним крупным объектом. Начиная применять *инкапсуляцию глобальных ссылок*, мы создаем новые объекты и подразделяем систему таким образом, чтобы упростить работу с ней.

Если процедурные языки имеют объектно-ориентированные расширения, то они позволяют нам двигаться в направлении ООП. Это еще не настоящее ООП, а только использование объектов, чтобы разделить программу на части для последующего тестирования.

Что еще мы можем сделать, помимо извлечения зависимостей, если процедурный язык поддерживает ООП? Прежде всего мы можем постепенно двигаться в направлении более совершенной объектно-ориентированной структуры кода. Как правило, это означает, что связанные вместе функции нужно объединить в классы и извлечь множество методов, что-

бы разделить сложные виды ответственности. Дополнительные рекомендации по этому поводу приведены в главе 20.

Процедурный код не предоставляет нам столько же возможностей, сколько объектно-ориентированный код, однако мы можем заметно продвинуться в работе с унаследованным процедурным кодом. Отдельные швы, доступные в процедурных языках, существенно упрощают эту работу. Если у используемого процедурного языка есть объектно-ориентированный преемник, то рекомендуется двигаться именно в этом направлении. *Объектные швы* пригодны не только для размещения тестов по местам. Компоновочные швы, а также швы предварительной обработки вполне подходят для подготовки кода к тестированию, но, помимо того, они мало способствуют улучшению структуры кода.

Глава 20

Класс слишком крупный и его дальнейшее укрупнение нежелательно

Многие новые свойства, которые обычно вводятся в систему, оборачиваются не более чем незначительными ее улучшениями. Они требуют добавления небольшого количества кода и, возможно, еще нескольких дополнительных методов. В этой связи возникает искушение внести подобные изменения в уже существующий класс. Скорее всего, в коде, который нужно добавить, используются данные из некоторого существующего класса, и поэтому создается впечатление, будто код проще всего добавить непосредственно в этот класс. К сожалению, такой, на первый взгляд, простой способ внесения изменений может привести к серьезным осложнениям. Вводя код в существующие классы, мы в конечном итоге удлиняем методы и укрупняем классы. Наше программное обеспечение превращается в настоящее болото, в котором нетрудно увязнуть, выясняя способы ввода новых свойств или же пытаясь хотя бы понять, как действуют старые свойства.

Однажды меня пригласили консультировать группу, разработавшую архитектуру системы, которая на бумаге выглядела идеально. Разработчики ознакомили меня с основными классами данной архитектуры и порядком их взаимодействия в обычных случаях. Затем они показали мне пару аккуратно составленных блок-схем UML, наглядно демонстрировавших структуру системы. Но я был неприятно поражен, как только начал просматривать код. Каждый из классов, написанных разработчиками, можно было бы разделить, как минимум, на 10 более мелких классов и тем самым преодолеть трудности проектирования, которые испытывала эта группа.

Какие же трудности представляют крупные классы? Прежде всего, они запутанны. Когда в классе насчитывается 50–60 методов, зачастую очень трудно представить себе, что же нужно изменить и насколько это повлияет на весь остальной код. В худшем случае в крупных классах насчитывается невероятно большое количество переменных экземпляров, и поэтому трудно сказать, на что именно повлияет изменение конкретной переменной. Другая трудность состоит в планировании заданий. Если на класс возложено около 20 видов ответственности, то причин для его изменения, скорее всего, окажется намного больше. На одном и том же шаге итерации над разными свойствами такого класса могут одновременно работать несколько программистов. Если же они работают параллельно, то разработка может серьезно затормозиться, особенно из-за третьего затруднения: крупные классы очень трудно тестировать. Может быть воспользоваться инкапсуляцией? Только не предлагайте этого тем, кто тестирует код, иначе они будут готовы оторвать вам голову. Ведь в крупных классах и так скрывается слишком многое. Инкапсуляция отлично подходит для осмысления кода. Она помогает нам выяснить, что некоторые элементы кода можно изменить только при определенных обстоятельствах. Но если код инкапсулируется слишком сильно, то он склонен к загниванию и порче. Когда же распознать воздействия в связи с изменением кода оказывается очень непросто, разработчики возвращаются к про-

граммированию методом *правки наудачу*, тогда возможны два исхода — либо изменения затянутся надолго, либо возрастет количество программных ошибок. Ведь за отсутствие ясности в изменяемом коде все равно приходится расплачиваться.

Первый вопрос, который возникает у нас, когда мы имеем дело с крупным классом, заключается в следующем: как нам работать, не ухудшая положение еще больше? Основными тактическими приемами в данном случае могут стать *почкование класса* и *почкование метода*. Если нам приходится вносить изменения в код, то мы должны рассмотреть возможность поместить код в новый класс или же новый метод. Почкование класса действительно позволяет избежать ухудшения положения. Поместив добавляемый код в новый класс, мы, конечно, должны передать ему полномочия исходного класса, но, по крайней мере, мы не сделаем исходный класс еще крупнее. Не менее полезным оказывается и почкование метода, хотя это и не столь очевидно. Если мы введем код в новый метод, то, конечно, получим еще один метод, но, по крайней мере, мы явно обозначим еще одну функцию, выполняемую изменяемым классом. Зачастую имена методов дают представление о том, как разделить класс на более мелкие части.

Лучшим средством борьбы с крупными классами является реорганизация кода. Она помогает разделить крупные классы на более мелкие классы. Но такая мера требует разрешения следующего насущного вопроса: как должны выглядеть более мелкие классы? К счастью, для этого у нас имеется руководящий принцип.

Принцип единственной ответственности

На каждый класс может быть возложена лишь одна ответственность. У него должно быть единственное назначение в системе, а следовательно, и единственная причина для его изменения.

Принцип единственной ответственности (SPR) трудно описать словесно, поскольку само понятие ответственности весьма расплывчато. В самой простой интерпретации этот принцип можно представить как наличие у каждого класса только одного метода. Конечно, методы можно рассматривать в качестве видов ответственности. Так, класс `Task` может отвечать за выполнение определенной задачи (запускаемой с помощью метода `run`), за сообщение о числе имеющихся подзадач (с помощью метода `taskCount`) и т.д. Но истинный смысл, который мы вкладываем в понятие ответственности, проявляется лишь тогда, когда мы говорим об *основном назначении*. Обратимся к примеру, приведенному на рис. 20.1.

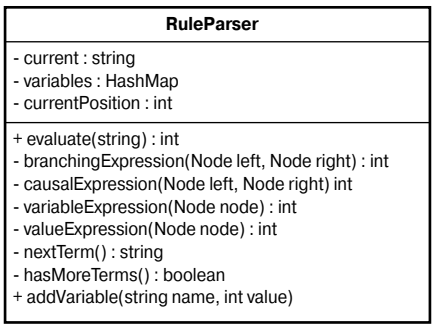


Рис. 20.1. Класс синтаксического анализа правил

Допустим, что у нас имеется небольшой класс, выполняющий синтаксический анализ строк, содержащих выражения с правилами на каком-то неопределенном языке. Какие

виды ответственности возложены на этот класс? Один из видов ответственности можно определить, глядя на имя класса: он выполняет синтаксический анализ. Но является ли это основным назначением данного класса? По-видимому, нет. Этот класс оценивает еще что-то.

Что еще делает этот класс? Он сохраняет текущую строку, т.е. ту строку, которую он анализирует, а также содержит поле, в котором хранится текущее состояние синтаксического анализа строки. Оба эти вида мини-ответственности, по-видимому, подпадают под категорию синтаксического анализа.

Рассмотрим еще одну переменную — поле `variables`. В этом поле хранятся переменные, используемые синтаксическим анализатором для вычисления таких арифметических выражений, как `a + 3`. При вызове метода `addVariable` (сложить со значением переменной) с аргументами `a` и `1` результат вычисления выражения `a + 3` будет равен `4`. Таким образом, на данный класс возлагается еще один вид ответственности: манипулирование переменными.

Имеются ли другие виды ответственности? Их можно также обнаружить по именам методов. Но имеется ли естественный способ группирования имен методов? По-видимому, методы могут быть разделены на следующие группы.

```
evaluate   branchingExpression   nextTerm          addVariable
           causalExpression      hasMoreTerms
           variableExpression
           valueExpression
```

Метод `evaluate` служит в качестве точки входа в данный класс. Это один из двух общедоступных методов. Он обозначает основной вид ответственности, возлагаемой на данный класс: вычисление выражений. Аналогичный вид ответственности обозначают все классы, имена которых оканчиваются на суффикс `Expression`. Помимо общего сходства имен, эти методы воспринимают узлы (`Node`) в качестве аргумента и возвращают целое значение типа `int`, обозначающее результат вычисления подвыражения. Аналогичным образом действуют методы `nextTerm` (следующий член) и `hasMoreTerms` (содержит дополнительные члены). По-видимому, они преобразуют отдельные члены анализируемого выражения в лексемы. Как упоминалось выше, метод `addVariable` занимается манипулированием переменными.

Подводя итог, можно сказать, что на класс `RuleParser`, по-видимому, возложены следующие виды ответственности.

- Синтаксический анализ
- Вычисление выражений
- Преобразование членов выражения в лексемы
- Манипулирование переменными

Если бы нам пришлось придумывать структуру данного класса с самого начала, разделив все эти виды ответственности, то результат получился бы подобным приведенному на рис. 20.2.

Не является ли такое разделение ответственности чрезмерным? Возможно, и является. Обычно те, кто разрабатывает небольшие языковые интерпретаторы, объединяют синтаксический анализ с вычислением выражений, вычисляя последние по ходу их анализа. Но несмотря на все удобства такого совмещения видов ответственности, оно не допускает расширений по мере развития самого языка. Ограниченная ответственность присуща также классу `SymbolTable` (таблица символов). Если единственной ответственностью, возлага-

емой на класс SymbolTable, является преобразование имен переменных в целые числа, то использование такого класса не дает никаких преимуществ по сравнению с обычным списком или хеш-таблицей. Структура, приведенная на рис. 20.2, выглядит довольно изящно, но она весьма гипотетична. Если нам не придется переписывать заново данную часть системы, то подобная многоклассовая структура окажется не более чем воздушным замком.

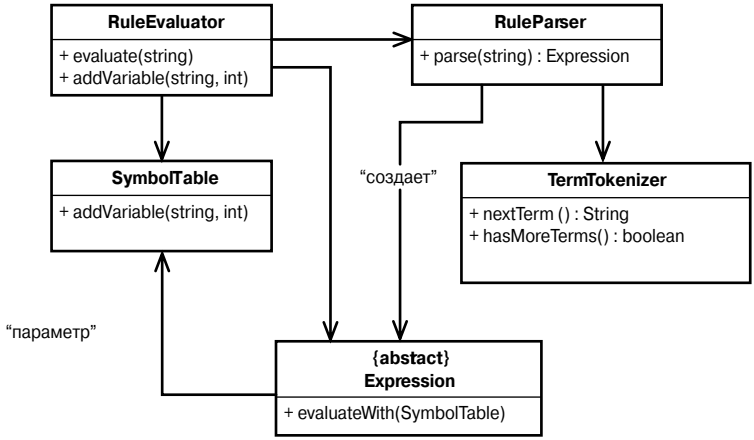


Рис. 20.2. Классы отдельных правил с разделенными видами ответственности

Когда приходится иметь дело с крупными классами в реальных условиях разработки, сначала следует выявить разные виды ответственности, а затем выбрать способ постепенной конкретизации видов ответственности.

Выявление видов ответственности

В примере класса RuleParser из предыдущего раздела было продемонстрировано конкретное разбиение крупного класса на более мелкие классы. Но оно было сделано довольно механически. Сначала мы перечислили все имеющиеся в данном классе методы, а затем приступили к осмыслению их назначения, задавая себе следующие наводящие вопросы: почему этот метод находится здесь и что он делает в данном классе? После этого мы сгруппировали методы по их предполагаемому сходному назначению.

Такое выявление видов ответственности можно назвать группированием. Но это лишь один из многих способов распознавания видов ответственности в существующем коде.

Умение выявлять виды ответственности является одним из главных навыков проектирования программного обеспечения, приобретаемых с опытом. По-видимому, говорить о таком навыке в контексте работы с унаследованным кодом было бы не совсем уместно, но на самом деле выявление видов ответственности в существующем коде мало чем отличается от их формулирования для еще ненаписанного кода. Самое главное — научиться сначала распознавать виды ответственности, а затем разделять их. Во всяком случае унаследованный код предоставляет больше возможностей для применения подобного навыка проектирования, чем новый. Глядя на код, который требуется изменить, намного проще оценивать разные варианты проектирования, а также обнаруживать соответствие структуры кода определенному контексту, поскольку этот контекст реальный, а не предполагаемый.

В этом разделе описывается поэтапный эвристический анализ, который можно применить для выявления видов ответственности в существующем коде. Следует иметь в виду, что виды ответственности в данном случае не придумываются, а просто обнаруживаются в коде. Независимо от структуры унаследованного кода, функции его фрагментов поддаются распознаванию. Иногда они различаются с трудом, но рассматриваемые здесь приемы существенно помогут в этом. Вы можете даже опробовать их на коде, который не требует немедленного изменения. Чем больше видов ответственности вы начнете замечать в коде, тем лучше вы его узнаете.

Этап эвристического анализа №1. Группирование методов

Найдите сходные методы по их именам. Выпишите все методы отдельного класса вместе с типами доступа к ним (общедоступного, частного и прочего) и попытайтесь найти методы, которые можно как-то сгруппировать.

Такой способ группирования методов послужит в качестве неплохой отправной точки, особенно при анализе крупных классов. Самое главное, что он не предполагает разделения всех методов по их именам на новые классы. Нужно лишь попытаться найти у методов часть общей ответственности. Если вам удастся выявить такие части ответственности, которые составляют главную ответственность, возложенную на класс, то по этому признаку можно будет в дальнейшем разделить и сам код. Дождитесь момента, когда придется видоизменить один из разделенных на группы методов, а затем решите, требуется ли извлечение класса в данной точке.

Группирование методов может также служить отличным упражнением для группы разработчиков. Установите в рабочем помещении щиты с плакатами, где перечислены имена методов каждого из основных классов. Члены группы будут периодически делать пометки на плакатах, обозначая разные виды группирования методов. В конце концов вся группа примет участие в таком группировании и придет к общему решению, на какие из методов и в какое время следует направить основные усилия.

Этап эвристического анализа №2. Анализ скрытых методов

Уделите внимание частным и защищенным методам. Если их много в отдельном классе, то зачастую это служит верным признаком того, что в нем имеется другой класс, который так и просится, чтобы его извлекли.

Крупные классы могут скрывать слишком много. Перед теми, кто только начинает осваивать блочное тестирование, постоянно встает следующий вопрос: как тестировать частные методы? Многие разработчики тратят немало времени, пытаясь разрешить данный вопрос, но, как упоминалось в этой главе, ответ на него на самом деле состоит в следующем: если требуется протестировать частный метод, он не должен быть частным. Если же частный метод трудно сделать общедоступным, то, скорее всего, он частично относится к ответственности, разделяемой с другим классом.

Характерным примером тому служит рассмотренный ранее класс `RuleParser`. В нем имеются два общедоступных метода: `evaluate` и `addVariable`, а все остальные методы — частные. Каким бы стал класс `RuleParser`, если бы мы сделали общедоступными методы `nextTerm` и `hasMoreTerms`? Пожалуй, он выглядел бы довольно странно. У пользователей синтаксического анализатора могло бы сложиться представление, будто для синтаксического анализа и вычисления выражений им придется воспользоваться этими двумя методами наряду с методом `evaluate`. Было бы странно, если бы эти методы ока-

зались общедоступными в классе RuleParser, но менее странно, а фактически идеально, если бы они оказались таковыми в классе TermTokenizer (преобразователь членов выражения в лексемы). Благодаря этому класс RuleParser нисколько не становится менее инкапсулированным. Несмотря на то, что методы nextTerm и hasMoreTerms окажутся общедоступными в классе TermTokenizer, они будут по-прежнему доступны частным образом в классе RuleParser, как показано на рис. 20.3.

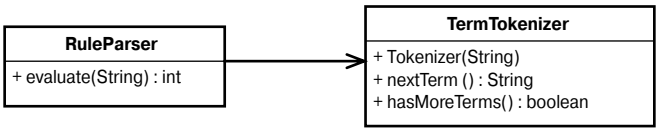


Рис. 20.3. Классы RuleParser и TermTokenizer

Этап эвристического анализа №3. Поиск решений, которые можно изменить

Найдите решения, которые вы уже воплотили в коде, а не собираетесь только воплотить. Имеется ли в коде какая-нибудь функция (например, обращение к базе данных, к другой совокупности объектов и т.д.), которая кажется слишком жестко закодированной? Можете ли вы представить себе, как ее изменить?

При попытке разделить крупный класс возникает искушение уделить немало внимания именам методов. Ведь в конечном итоге они представляют собой самые примечательные указания на свойства класса. Но имена методов не могут дать полную картину того, что делает класс. Нередко в крупных классах находятся методы, выполняющие самые разные функции на различных уровнях абстракции. Например, метод под названием updateScreen() (обновить экран) может сформировать текст для отображения, отформатировать и отправить его самым разным объектам графического пользовательского интерфейса. По одному только имени данного класса нельзя сказать точно, какие функции и виды ответственности возложены на этот метод.

Именно по этой причине целесообразно немного реорганизовать код за счет извлечения методов, прежде чем приступать к извлечению самих классов. Какие же методы следует извлечь? Для этого следует найти соответствующие решения. Сколько функций допускается в коде? Вызывает ли код методы из конкретного интерфейса API? Предполагается ли, что код будет всегда обращаться к одной и той же базе данных? Если код выполняет все эти функции, то имеет смысл извлечь методы, отражающие конкретные намерения на более высоком уровне. Так, если из базы данных выбирается определенная информация, то извлечь следует метод, названный по выбираемой информации. В результате подобных извлечений получается намного больше методов, и тогда, возможно, проще будет прибегнуть к группированию методов. Более того, вполне может оказаться, что некоторые ресурсы полностью инкапсулированы внутри ряда извлеченных методов. А извлечение класса для них может привести к нарушению некоторых зависимостей от деталей более низкого уровня.

Этап эвристического анализа №4. Поиск внутренних взаимосвязей

Найдите взаимосвязи между переменными экземпляров и методами. Имеются ли такие переменные экземпляров, которые используются в одних методах и не используются в других?

На самом деле очень трудно найти классы, где во всех методах используются все переменные экземпляров. В каждом классе всегда можно найти что-нибудь "несуразное". В частности, три переменные могут использоваться лишь в двух или трех методах. И зачастую об этом можно судить по именам. Например, в классе `RulerParser` имеется поле `variables` и метод под названием `addVariable`. Это явно указывает на взаимосвязь между данным методом и полем `variables`. И хотя это ничего не говорит нам о наличии других методов, получающих доступ к данному полю, но, по крайней мере, может служить отправной точкой для дальнейшего поиска.

Для поиска подобных "несуразностей" можно также набросать небольшой эскиз взаимосвязей внутри класса. Это так называемый *эскиз свойств*. Он очень просто создается и наглядно показывает, какие методы и переменные экземпляров использует каждый метод отдельного класса. Рассмотрим следующий пример кода.

```
class Reservation
{
    private int duration;
    private int dailyRate;
    private Date date;
    private Customer customer;
    private List fees = new ArrayList();

    public Reservation(Customer customer, int duration,
            int dailyRate, Date date) {
        this.customer = customer;
        this.duration = duration;
        this.dailyRate = dailyRate;
        this.date = date;
    }

    public void extend(int additionalDays) {
        duration += additionalDays;
    }

    public void extendForWeek() {
        int weekRemainder = RentalCalendar.weekRemainderFor(date);
        final int DAYS_PER_WEEK = 7;
        extend(weekRemainder);
        dailyRate = RateCalculator.computeWeekly(
                        customer.getRateCode())
                    / DAYS_PER_WEEK;
    }

    public void addFee(FeeRider rider) {
        fees.add(rider);
    }

    int getAdditionalFees() {
        int total = 0;
        for(Iterator it = fees.iterator(); it.hasNext(); ) {
            total += ((FeeRider)(it.next())).getAmount();
```

```
        }
        return total;
    }

    int getPrincipalFee() {
        return dailyRate
               * RateCalculator.rateBase(customer)
               * duration;
    }

    public int getTotalFee() {
        return getPrincipalFee() + getAdditionalFees();
    }
}
```

Прежде всего, обведем кружками каждую переменную, как показано на рис. 20.4.

Далее проанализируем каждый метод и обозначим его отдельным кружком. Затем проведем линию от кружка каждого метода к кружкам любых переменных и методов, к которым он обращается или видоизменяет. При этом конструкторы обычно опускаются, поскольку они видоизменяют каждую переменную экземпляра.

На рис. 20.5 показан вид блок-схемы после добавления кружка для метода `extend`.

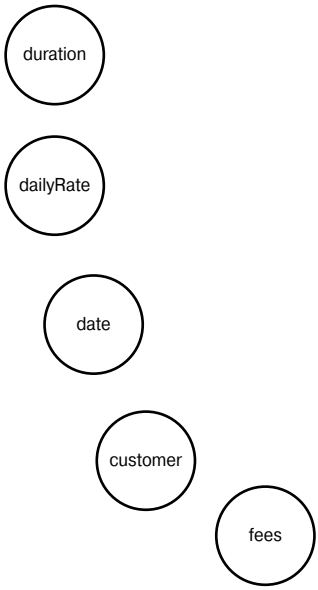


Рис. 20.4. Переменные из класса `Reservation`

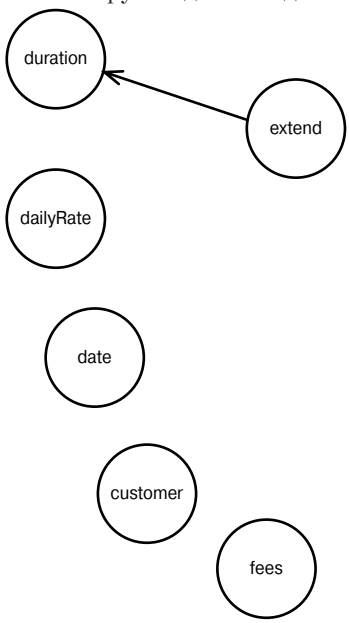


Рис. 20.5. В методе `extend` *используется переменная* `duration`

Если вы читали главу 11, посвященную составлению эскизов воздействий, то, вероятно, обратили внимание на сходство рассматриваемых здесь *эскизов свойств* с *эскизами воздействий*. И действительно они очень похожи. Главное отличие заключается в том, что стрелки в них обращены в разные стороны. В эскизах свойств стрелки указывают в направлении метода или переменной, используемой другим методом или переменной.

А в эскизах воздействий стрелки направлены в сторону методов или переменных, на которые воздействуют другие методы и переменные.

Эти виды эскизов представляют собой два вполне допустимых способа анализа взаимодействий в системе. Так, эскизы свойств удобны для отображения внутренней структуры классов, тогда как эскизы воздействий — для осмысления в прямом направлении от точки внесения изменений в код.

Доставляет ли какие-то неудобства сходство этих эскизов? Совсем не доставляет. Такие эскизы служат лишь один раз. Они быстро набрасываются разработчиками перед внесением изменений в код и затем выбрасываются. А поскольку хранить эти эскизы не имеет никакого смысла, то вряд ли их можно перепутать.

На рис. 20.6 показан вид эскиза после добавления кружков для каждого свойства и линий для всех свойств, которые ими используются.

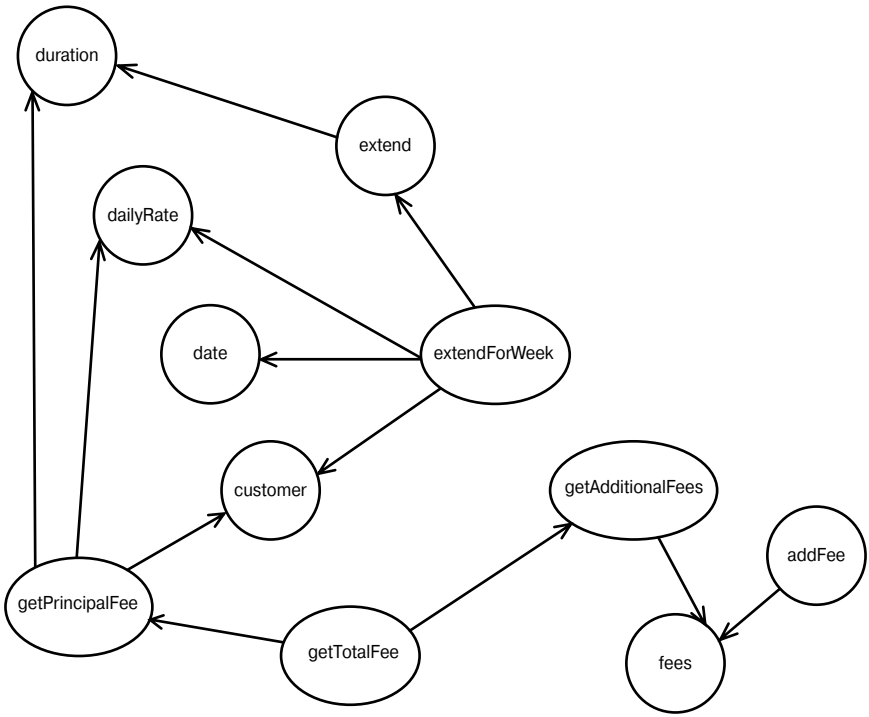


Рис. 20.6. Эскиз свойств для класса `Reservation`

Что же можно узнать из такого эскиза? Самое очевидное, что в данном классе наблюдается некоторая кластеризация. В частности, переменные `duration` (продолжительность), `dailyRate` (дневная ставка), `date` (дата) и `customer` (потребитель) используются, главным образом, методами `getPrincipalFee` (получить основную оплату), `extendForWeek` (продлить на неделю) и `extend`. Является ли какой-нибудь из этих методов общедоступным? Да, методы `extend` и `extendForWeek` общедоступны, тогда как метод `getPrincipalFee` таковым не является. Как бы выглядела анализируемая нами система, если бы мы выделили этот кластер, или совокупность объектов, в отдельный класс? Результат такого выделения приведен на рис. 20.7.

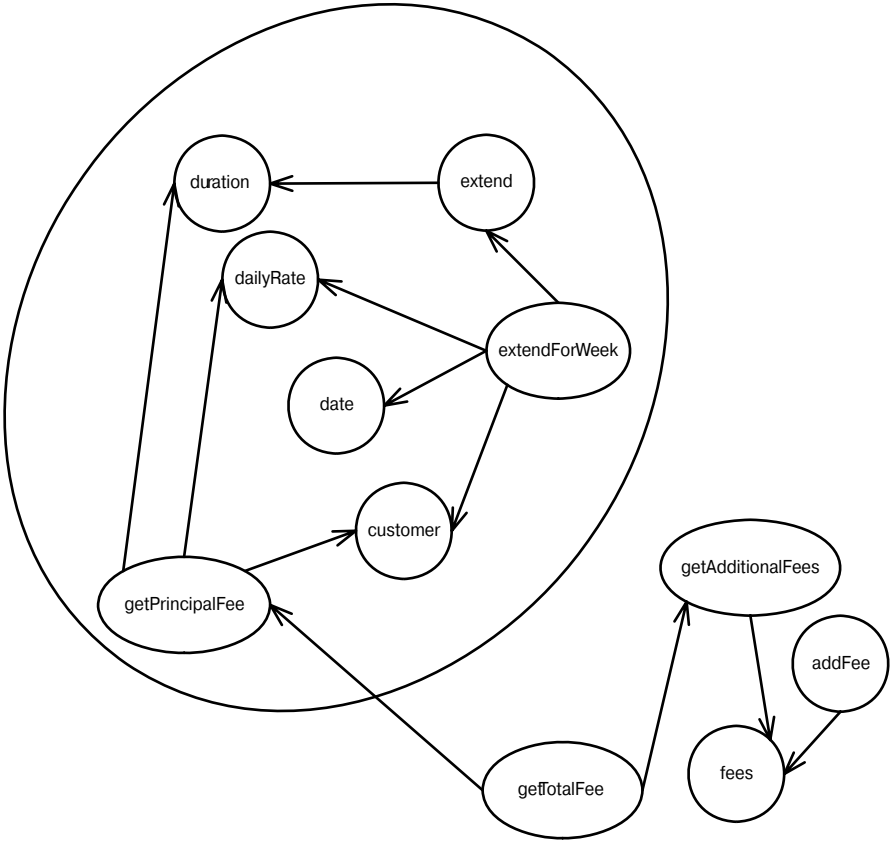


Рис. 20.7. Кластер в классе `Reservation`

Все, что обведено большим кружком в эскизе, приведенном на рис. 20.7, может быть выделено в новый класс. В таком классе пришлось бы сделать общедоступными методы `extend`, `extendForWeek` и `getPrincipalFee`, а остальные методы можно было бы оставить частными. В то же время методы `fees` (виды оплаты), `addFee` (добавить оплату), `getAdditionalFees` (получить дополнительные виды оплаты) и `getTotalFee` (получить итоговую оплату) можно было бы оставить в классе `Reservation` (резервирование) и передать соответствующие полномочия новому классу (рис. 20.8).

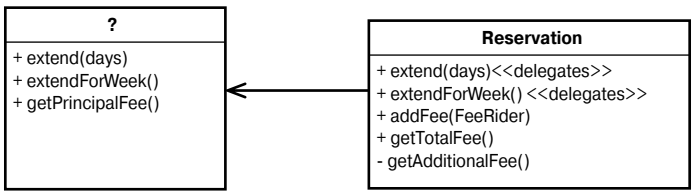


Рис. 20.8. Использование классом `Reservation` *нового класса*

Прежде чем пытаться создавать новый класс, очень важно выяснить, насколько отчетливой и определенной представляется ответственность, возлагаемая на этот класс. Можно

ли подобрать для него подходящее имя? По-видимому, этот класс выполняет две функции — продлевает резервирование и рассчитывает его основную оплату. Для него вполне подошло бы имя `Reservation`, но оно уже присвоено исходному классу.

Рассмотрим совсем иную возможность. Вместо того чтобы извлекать весь код, обведенный большим кружком на рис. 20.7, мы можем извлечь другой код, как показано на рис. 20.9.

Извлеченный класс можно назвать `FeeCalculator` (калькулятор оплаты). Такой класс вполне работоспособен, но ведь методу `getTotalFee` нужно вызывать метод `getPrincipalFee` из класса `Reservation`, не так ли?

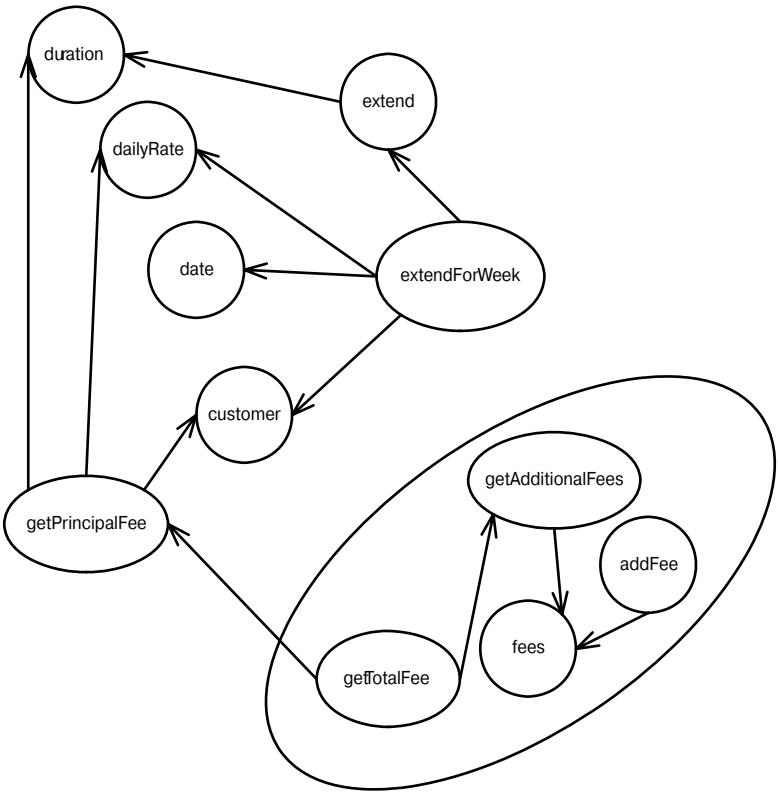


Рис. 20.9. Иное представление о классе `Reservation`

Что, если вызвать метод `getPrincipalFee` из класса `Reservation`, а затем передать значение классу `FeeCalculator`? Ниже приведен фрагмент кода, соответствующий набросанному эскизу.

```
public class Reservation
{
    ...
    private FeeCalculator calculator = new FeeCalculator();

    private int getPrincipalFee() {
    ...
```

```
    }

    public Reservation(Customer customer, int duration,
            int dailyRate, Date date) {
        this.customer = customer;
        this.duration = duration;
        this.dailyRate = dailyRate;
        this.date = date;
    }

    ...

    public void addFee(FeeRider fee) {
        calculator.addFee(fee);
    }

    public getTotalFee() {
        int baseFee = getPrincipalFee();
        return calculator.getTotalFee(baseFee);
    }

}
```

В окончательном виде структура рассматриваемого здесь кода выглядит так, как показано на рис. 20.10.

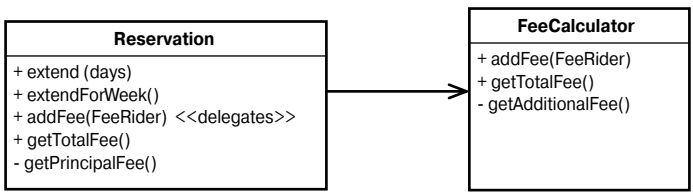


Рис. 20.10. Использование классом `Reservation` *класса* `FeeCalculator`

Мы можем даже рассмотреть возможность переноса метода `getPrincipalFee` в класс `FeeCalculator`, чтобы имена классов точнее соответствовали видам возлагаемой на них ответственности. Но если учесть, что метод `getPrincipalFee` зависит от целого ряда переменных из класса `Reservation`, то его, вероятно, лучше оставить там, где он теперь находится.

Эскизы свойств отлично подходят для поиска в классах отдельных видов ответственности. Мы можем попытаться сгруппировать свойства и выяснить по именам, какие классы можно извлечь. Но помимо того что эскизы свойств помогают нам обнаружить виды ответственности, они позволяют также выявить структуру зависимостей внутри классов, а это зачастую не менее важно, чем выявление ответственности, когда нам приходится решать, что именно следует извлечь. В приведенном выше примере отчетливо просматриваются две совокупности переменных и методов. Единственной связью между ними является вызов метода `getPrincipalFee` внутри метода `getTotalFee`. Подобные связи мы нередко прослеживаем в эскизах свойств по линиям, соединяющим небольшие совокупности объектов с более крупными. Это так называемые *точки сужения*, о которых речь уже шла в главе 12.

Иногда, набросав эскиз свойств, просто невозможно обнаружить точки сужения. Они не всегда присутствуют в таком эскизе. Но, по крайней мере, в эскизе свойств можно проследить имена и зависимости.

Набросав эскиз, вы можете опробовать разные способы разделения класса на части. Для этого достаточно обвести кружком выбранные группы свойств. Линии, которые пересекает кружок, могут определять границу раздела для нового класса. Обводя свойства кружком, попробуйте подобрать подходящее имя класса для каждой группы свойств. Ведь помимо выбора вариантов для извлечения класса, это отличная практика в приобретении навыков присвоения имен и неплохая возможность исследовать альтернативные варианты проектирования.

Этап эвристического анализа №5. Выявление основной ответственности

Попробуйте описать одним предложением возлагаемую на класс ответственность.

Принцип единственной ответственности гласит, что на класс должна быть возложена единственная ответственность. В таком случае ее нетрудно описать одним предложением. Попробуйте сделать это для крупного класса в вашей системе. Обдумывая то, что клиенты требуют и ожидают от класса, добавьте в предложение фразы, обозначающие отдельные функции класса. Если одна функция окажется важнее, чем все остальные, то она, вероятно, и есть основной ответственностью, возлагаемой на класс. А остальные виды ответственности, возможно, следует перенести в другие классы.

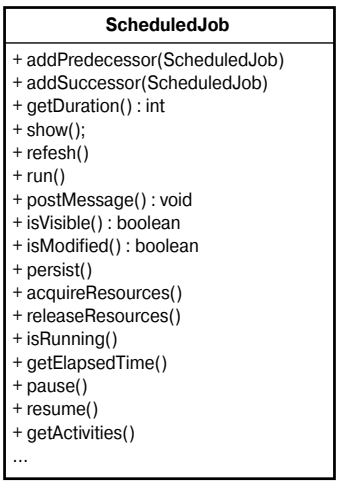


Рис. 20.11. Класс `ScheduledJob`

Принцип единственной ответственности может быть нарушен как на уровне интерфейса, так и на уровне реализации. Данный принцип нарушается на уровне интерфейса, когда класс представляет интерфейс, явно показывающий, что этот класс отвечает за очень большое количество функций. Например, интерфейс класса `ScheduledJob` (планируемое задание), приведенного на рис. 20.11, выглядит так, как будто его можно разделить на три или четыре класса.

Но больше всего нас должно беспокоить нарушение принципа единственной ответственности на уровне реализации. Проще говоря, нас должно заботить, действительно ли класс выполняет все возложенные на него обязанности или же он просто делегирует свои полномочия ряду других классов. В последнем случае мы имеем дело не с крупным и монолитным классом, а лишь с неким фасадом, за которым скрывается целый ряд мелких классов, и поэтому справиться с таким классом несколько проще.

На рис. 20.12 приведен класс `ScheduledJob` с видами ответственности, делегируемыми ряду других классов.

В данном примере принцип единственной ответственности по-прежнему нарушается на уровне интерфейса, но на уровне реализации дело обстоит чуть лучше.

Как же разрешить данное затруднение на уровне интерфейса? Сделать это не так-то просто. Обычно для этого нужно выяснить, могут ли клиенты непосредственно использовать некоторые классы, которым делегируются полномочия. Так, если лишь некоторые клиенты заинтересованы в выполнении заданий с помощью класса `ScheduledJob`, можно реорганизовать его структуру так, как показано на рис. 20.13.

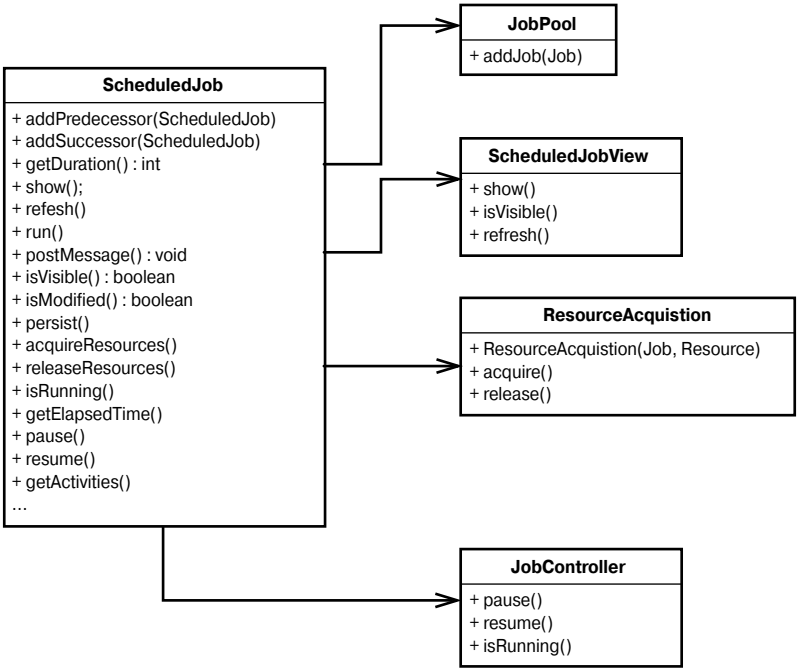


Рис. 20.12. Класс `ScheduledJob` *вместе с извлеченными классами*

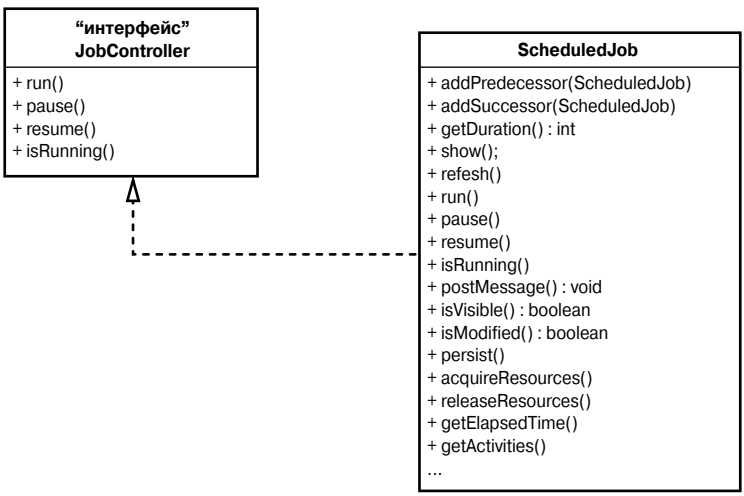


Рис. 20.13. Интерфейс класса `ScheduledJob`*, ориентированный на клиентов*

Теперь клиенты, которых заботит только контроль заданий, могут воспринимать объекты класса `ScheduledJob` как интерфейсы `JobController` (контролер заданий). Такой способ организации интерфейса для конкретного ряда клиентов вполне согласуется с *принципом отделения интерфейса* (ISP).

Принцип отделения интерфейса

Когда класс оказывается крупным, то клиенты такого класса редко пользуются всеми его методами. В нем нередко можно выделить разные группы методов, используемых отдельными клиентами. Если мы создадим интерфейс для каждой из этих групп и реализуем такие интерфейсы в данном крупном классе, то он будет доступен каждому клиенту через отдельный интерфейс. Это поможет нам скрывать нужную информацию и уменьшить зависимость в системе. А клиентам уже не придется перекомпилировать код всякий раз, когда это делается для данного крупного класса.

Когда у нас имеются интерфейсы для конкретного ряда клиентов, мы можем, как правило, приступить к переносу кода из крупного класса в новый класс, использующий исходный класс, как показано на рис. 20.14.

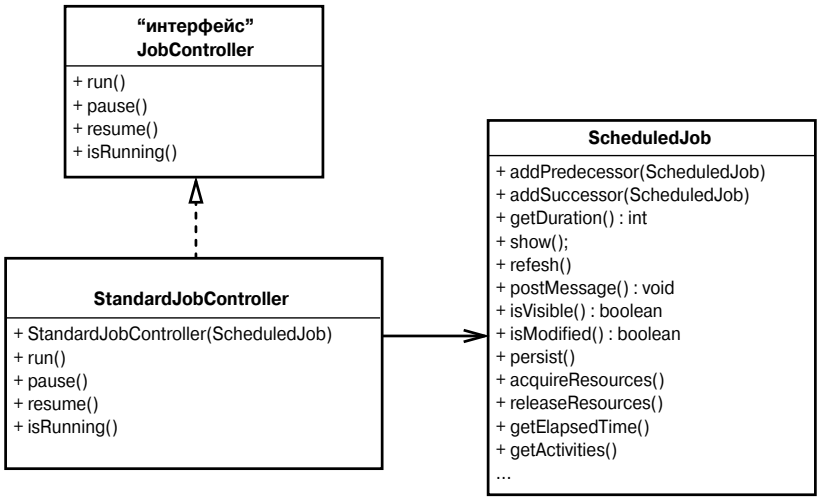


Рис. 20.14. Отделения интерфейса от класса `ScheduledJob`

Вместо того чтобы делегировать полномочия класса `ScheduledJob` интерфейсу `JobController`, мы поступили наоборот, делегировав полномочия интерфейса `JobController` классу `ScheduledJob`. Если теперь клиенту потребуется обратиться к классу `ScheduledJob` для выполнения задания, то он создаст интерфейс `JobController`, передав ему объект класса `ScheduledJob`, и затем воспользуется интерфейсом `JobController`, чтобы выполнить задание.

Такую реорганизацию кода обычно труднее сделать, чем это кажется на первый взгляд. Для этого зачастую приходится раскрывать больше методов в общедоступном интерфейсе исходного класса (`ScheduledJob`), чтобы новый лицевой класс (`StandardJobController`) имел доступ ко всему, что ему требуется для выполнения своих функций. Для внесения в код подобных изменения обычно требуется приложить немало труда. Ведь теперь нужно внести изменения в клиентский код, чтобы воспользоваться новым классом вместо старого. И для того чтобы сделать это благополучно, необходимо окружить клиентский код соответствующими тестами. Но подобная реорганизация кода все же примечательна тем, что она позволяет выделить интерфейс из крупного класса. Обратите внимание на то, что

в классе `ScheduledJob` отсутствуют методы, которые теперь находятся в интерфейсе `JobController`.

Этап эвристического анализа №6. Черновая реорганизация, когда ничто другое не помогает

Если вы испытываете серьезные затруднения при выявлении видов ответственности в отдельном классе, то вам может помочь незначительная черновая реорганизация кода.

Черновая реорганизация кода — весьма эффективное средство. Не следует только забывать, что это искусственный прием. Функции, которые вы обнаруживаете в ходе черновой реорганизации кода, совсем не обязательно останутся после такой реорганизации.

Этап эвристического анализа №7. Сосредоточенность на текущей работе

Уделите особое внимание тому, что вам предстоит сделать в настоящий момент. Если вы пытаетесь реализовать какую-нибудь функцию иначе, то, скорее всего, вы выявили ответственность, которую следует извлечь, а затем заменить.

Количество отдельных видов ответственности, выявленных в классе, может оказаться просто подавляющим. Не забывайте, однако, что изменения, которые вы делаете в настоящий момент, указывают совершенно конкретно, каким образом может измениться программа. Нередко осознания такого изменения оказывается достаточно, чтобы выявить во вновь создаваемом коде отдельную ответственность.

Другие способы

Эвристический анализ для выявления видов ответственности может оказать существенную помощь в обнаружении и раскрытии новых абстракций в старых классах, но это всего лишь отдельный прием. Для того чтобы по-настоящему овладеть искусством выявлять виды ответственности, нужно проштудировать литературу по шаблонам проектирования, а еще лучше — почаще просматривать чужой код. Найдите открытые проекты и посмотрите, как другие разработчики реализуют свои замыслы в коде. Уделите особое внимание присвоению имен классам и их соответствию именам методов. Со временем вы научитесь лучше выявлять скрытые виды ответственности и сразу обращать на них внимание, просматривая незнакомый код.

Дальнейшие действия

Как только вы выявите самые разные виды ответственности в крупном классе, вам останется решить два вопроса — стратегический и тактический. Рассмотрим сначала стратегический вопрос.

Стратегия

Что делать после выявления отдельных видов ответственности? Следует ли уделить целую неделю попыткам усовершенствовать крупные классы в системе? И следует ли

разделить их на мелкие фрагменты? Если у вас есть на все это время, то прекрасно, но такое бывает редко. Кроме того, это бывает рискованно. Как показывает мой опыт, когда разработчики начинают безудержно реорганизовывать код, система становится на какое-то время неустойчивой — даже если они действуют осторожно и пишут тесты. Если вы находитесь лишь на начальной стадии работы над проектом, готовы пойти на риск и располагаете временем, то можете предаться реорганизации кода. Только не допускайте много программных ошибок, чтобы они не отбили у вас охоту к другой реорганизации кода.

Самый лучший подход к разделению классов на части заключается в следующем: выявить виды ответственности, убедиться в том, что они понятны всем остальным членам группы, а затем разделить каждый класс на части по мере надобности. Поступая подобным образом, вы равномерно распределяете риск от изменений и можете по ходу дела выполнять другую работу.

Тактика

Приступая к работе с крупным классом в унаследованной системе, самое большее, на что вы можете рассчитывать, — это попробовать соблюсти принцип единственной ответственности на уровне реализации, а именно: извлечь классы из крупного класса и передать им соответствующие полномочия. Соблюсти принцип единственной ответственности на уровне интерфейса немного труднее, поскольку придется внести изменения в клиенты изменяемого класса и протестировать их. Примечательно, что соблюдение данного принципа на уровне реализации упрощает его соблюдение на уровне интерфейса. Рассмотрим сначала, как это делается на уровне реализации.

Выбор способа извлечения классов зависит от целого ряда факторов и, в частности, от простоты размещения тестов вокруг методов, на которые могут оказать влияние подобные изменения. В связи с этим целесообразно проанализировать изменяемый класс и составить перечень всех переменных экземпляров и методов, которые требуется перенести. Из такого перечня можно составить ясное представление о тех методах, для которых следует написать тесты. Так, если бы в рассмотренном ранее примере класса `RuleParser` потребовалось разделить на части класс `TermTokenizer`, то нам бы пришлось перенести строковое поле `current` и поле `currentPosition` (текущее положение) вместе с методами `hasMoreTerms` и `nextTerm`. А поскольку оба эти метода являются частными, то это означает, что мы не можем написать для них тесты непосредственно. Мы могли бы сделать эти методы частными (ведь их все равно придется переносить), но, возможно, проще было бы создать класс `RuleParser` в средствах тестирования и предоставить ему ряд строк для синтаксического анализа. Сделав это, мы получим в свое распоряжение тесты, косвенно покрывающие оба метода `hasMoreTerms` и `nextTerm`, чтобы затем благополучно перенести их в новый класс.

К сожалению, экземпляры многих крупных классов очень трудно получить в средствах тестирования. Соответствующие рекомендации по этому поводу приведены в главе 9. Если же вам удастся получить экземпляр крупного класса, то для последующего размещения тестов по местам вы можете воспользоваться рекомендациями, приведенными в главе 10.

Как только вам удастся разместить тесты по местам, вы можете приступить к извлечению класса самым непосредственным образом, прибегнув к реорганизации кода путем *извлечения класса*, описанной в упоминавшейся ранее книге Мартина Фаулера, *Рефакторинг. Улучшение существующего кода*. Но даже если вам не удастся разместить тесты по местам, то вы все равно сможете продвинуться вперед, хотя и несколько рискованным путем. Это довольно консервативный подход, который оказывается действенным независимо от при-

меняемого инструментального средства реорганизации кода. Ниже приведена многоэтапная процедура его применения.

1. Выявите ответственность, которую требуется выделить в отдельный класс.
2. Выясните, имеются ли какие-нибудь переменные экземпляров для переноса в новый класс. Если таковые имеются, то отделите соответствующую часть объявления класса от остальных переменных экземпляра.
3. Если имеются целые методы для переноса в новый класс, извлеките тело каждого из них в новые методы. Имя каждого нового метода должно быть таким же, как и у старого метода, но с уникальным префиксом, предшествующим имени, как, например, набранный прописными буквами префикс `MOVING` (перенос). Если вы не пользуетесь инструментальным средством реорганизации кода, то не забудьте *сохранить сигнатуры* при извлечении методов. Извлекая каждый метод, поместите его в упомянутой выше отдельной части объявления класса рядом с переносимыми переменными.
4. Если некоторые части методов должны быть перенесены в другой класс, извлеките их из исходных методов. Вновь добавьте префикс `MOVING` к именам этих методов и поместите в отдельной части.
5. На данном этапе у вас должна быть часть класса, содержащая переменные экземпляров и ряд методов, которые требуется перенести. Выполните текстовый поиск в текущем классе всех его подклассов, чтобы убедиться в том, что ни одна из переносимых переменных не используется за пределами методов, которые вы собираетесь перенести. На данном этап очень важно не делать *упор на компилятор*. Во многих языках ООП допускается объявление переменных в производном классе с тем же именем, что и в базовом классе. Такой прием обычно называется *сокрытием переменных*. Если класс скрывает любые переменные и другие предполагаемые места их применения, то при переносе переменных вы можете изменить поведение кода. С другой стороны, если вы делаете упор на компилятор, чтобы найти все места применения переменной, скрывающей другую переменную, то вы не сумеете обнаружить все места, где эта переменная используется. Если же закомментировать объявление скрываемой переменной, то станет видимой переменная, которая ее скрывает.
6. Теперь вы можете переместить все отделенные переменные экземпляров и методы непосредственно в новый класс. Создайте экземпляр нового класса в старом классе и сделайте упор на компилятор, чтобы найти именно в этом экземпляре, а не в старом классе, все места, где должны вызываться перенесенные методы.
7. По завершении переноса и компиляции кода вы можете приступить к удалению префикса `MOVING` в именах всех перенесенных методов. Вновь сделайте упор на компилятор, чтобы перейти к тем местам, где требуется изменить имена.

Приведенная выше процедура реорганизации кода довольно сложна, но если вам приходится иметь дело с очень запутанным фрагментом кода, то без нее вам не обойтись, если вы хотите благополучно извлечь классы без тестирования.

Извлечение классов без тестирования может пройти не совсем гладко по ряду причин. Самые неуловимые программные ошибки, которые мы можем внести в код, связаны с наследованием. Перенос метода из одного класса в другой сам по себе достаточно безопасен. В этом нам может оказать помощь *упор на компилятор*, но во многих языках программирования положение радикально меняется, если мы попытаемся перенести метод, перео-

пределяющий другой метод. В этом случае метод, вызываемый из исходного класса, будет вызываться с тем же именем из базового класса. То же самое происходит и с переменными. Переменная из подкласса может скрывать переменную с тем же самым именем из суперкласса. Но при ее переносе становится видимой скрытая переменная.

Во избежание подобных осложнений мы вообще не должны переносить исходные методы. Вместо этого мы создаем новые методы, извлекая в них тела из старых методов. А префикс, добавляемый к их именам, просто служит для того, чтобы не перепутать имена методов при их переносе. С переменными дело обстоит несколько сложнее. Нам придется найти вручную все места, где эти переменные применяются, прежде чем перенести их. Но при этом легко совершить ошибку. Поэтому мы должны быть очень внимательны, и лучше всего делать это вместе с коллегой.

Действия после извлечения класса

Извлечение классов из крупного класса нередко оказывается лишь удачно проведенным первым этапом. На практике самая большая опасность, которая подстерегает при этом разработчиков, заключается в излишней самоуверенности. Выполнив *черновую реорганизацию кода* или же выработав несколько иное представление о системе, вы не должны все же забывать, что имеете дело с вполне работоспособной структурой приложения, которая поддерживает определенные функции. Возможно, ее просто нельзя настроить так, чтобы двигаться дальше. Иногда самое лучшее, что можно сделать, — сначала сформулировать представление о том, как крупный класс должен выглядеть после реорганизации кода, а затем просто забыть от этом. Ведь это сделано лишь для того, чтобы обнаружить возможные варианты. А для того чтобы двигаться дальше, нужно очень чутко относится к тому, что уже имеется в коде, и не обязательно стремиться к его идеальной структуре, а хотя бы двигаться в правильном направлении.

Глава 21

Изменение одного и того же кода повсеместно

Это проблема может оказаться самой досадной в унаследованных системах. Если вам нужно внести изменения в код, то сначала вы относитесь к этому немного самоуверенно, как к простой задаче, но затем обнаруживаете, что одно и то же изменение вам нужно вносить снова и снова, поскольку в системе оказываются десятки мест с аналогичным кодом. Тогда вы ловите себя на мысли, что вам не пришлось бы постоянно сталкиваться с данной проблемой, если бы можно было перепроектировать систему или реорганизовать ее структуру, но, к сожалению, времени на это не хватает. В итоге еще один больной вопрос в системе так и остается неразрешенным, только добавляя хлопот.

Если вы умеете реорганизовывать код, то ваше положение не столь безнадежно. Вы знаете, что исключить дублирование кода намного легче, чем перепроектировать систему или же реорганизовать ее архитектуру, поскольку вам нужно делать это в небольших фрагментах кода. Со временем система станет лучше, если кто-нибудь без вашего ведома не станет плодить дублированный код. В последнем случае вам, скорее всего, придется принять против виновных в дублировании кода крутые меры, но это уже другой вопрос. Другое дело, стоит ли вообще этим заниматься, усердно выкашивая всякое дублирование из фрагмента кода? Что мы получаем в итоге? Весьма неожиданные результаты.

Обратимся к конкретному примеру. Допустим, что у нас имеется сетевая система на основе Java и нам требуется посылать команды на сервер. Классы команды, имеющиеся в нашем распоряжении, называются `AddEmployeeCmd` (команда ввода сотрудника) и `LogonCommand` (команда входа в систему). Когда нам нужно выдать команду, мы получаем экземпляр класса этой команды и передаем выходной поток методу `write` данного класса.

Ниже приведен листинг для обоих классов команд. Сможете ли вы обнаружить в нем дублирование кода?

```
import java.io.OutputStream;

public class AddEmployeeCmd {
    String name;
    String address;
    String city;
    String state;
    String yearlySalary;

    private static final byte[] header = {(byte)0xde, (byte)0xad};
    private static final byte[] commandChar = {0x02};
    private static final byte[] footer = {(byte)0xbe, (byte)0xef};
    private static final int SIZE_LENGTH = 1;
    private static final int CMD_BYTE_LENGTH = 1;
```

```
    private int getSize() {
        return header.length +
            SIZE_LENGTH +
            CMD_BYTE_LENGTH +
            footer.length +
            name.getBytes().length + 1 +
            address.getBytes().length + 1 +
            city.getBytes().length + 1 +
            state.getBytes().length + 1 +
            yearlySalary.getBytes().length + 1;
    }

    public AddEmployeeCmd(String name, String address,
                          String city, String state,
                          int yearlySalary) {
        this.name = name;
        this.address = address;
        this.city = city;
        this.state = state;
        this.yearlySalary = Integer.toString(yearlySalary);
    }

    public void write(OutputStream outputStream)
                throws Exception {
        outputStream.write(header);
        outputStream.write(getSize());
        outputStream.write(commandChar);
        outputStream.write(name.getBytes());
        outputStream.write(0x00);
        outputStream.write(address.getBytes());
        outputStream.write(0x00);
        outputStream.write(city.getBytes());
        outputStream.write(0x00);
        outputStream.write(state.getBytes());
        outputStream.write(0x00);
        outputStream.write(yearlySalary.getBytes());
        outputStream.write(0x00);
        outputStream.write(footer);
    }
}

import java.io.OutputStream;

public class LoginCommand {

    private String userName;
    private String passwd;
    private static final byte[] header
                = {(byte)0xde, (byte)0xad};
    private static final byte[] commandChar = {0x01};
    private static final byte[] footer
= {(byte)0xbe, (byte)0xef};
```

```
    private static final int SIZE_LENGTH = 1;
    private static final int CMD_BYTE_LENGTH = 1;

    public LoginCommand(String userName, String passwd) {
        this.userName = userName;
        this.passwd = passwd;
    }

    private int getSize() {
        return header.length + SIZE_LENGTH + CMD_BYTE_LENGTH +
            footer.length + userName.getBytes().length + 1 +
            passwd.getBytes().length + 1;
    }

    public void write(OutputStream outputStream)
                throws Exception {
        outputStream.write(header);
        outputStream.write(getSize());
        outputStream.write(commandChar);
        outputStream.write(userName.getBytes());
        outputStream.write(0x00);
        outputStream.write(passwd.getBytes());
        outputStream.write(0x00);
        outputStream.write(footer);
    }
}
```

На рис. 21.1 классы команд представлены в виде блок-схемы UML.

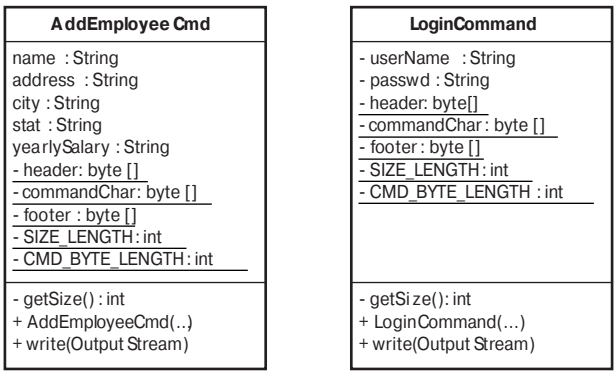


Рис. 21.1. Классы `AddEmployeeCmd` *и* `LoginCommand`

Похоже, что в приведенном выше коде имеется немало дублированных фрагментов, но что из этого? Объем кода все равно невелик. Если попытаться реорганизовать код, исключив дублирование и сделав его более компактным, то насколько это облегчит нам жизнь? Глядя на этот код, трудно сказать — насколько.

Попробуем выявить фрагменты дублированного кода и исключить его дублирование, чтобы посмотреть, к какому результату мы придем. После этого мы можем решить, насколько полезным оказывается исключение дублирования кода.

Прежде всего нам потребуется ряд тестов, которые мы могли бы выполнять после каждой реорганизации кода. К счастью, они у нас имеются. Ради краткости изложения они здесь не приводятся, но мы не должны забывать об их существовании.

Первые шаги

Когда я сталкиваюсь с дублированием кода, прежде всего пытаюсь отстраниться, чтобы получить полную картину происходящего. Делая это, я начинаю постепенно представлять себе классы, которые у меня в итоге получатся, а также внешний вид извлеченных фрагментов дублированного кода. Затем такое осмысление кода начинает казаться мне чрезмерным. Удаление небольших фрагментов дублированного кода на самом деле помогает лучше разглядеть в дальнейшем более крупные участки дублирования. Например, в методе `write` класса `LoginCommand` имеется следующий код:

```
outputStream.write(userName.getBytes());
outputStream.write(0x00);
outputStream.write(passwd.getBytes());
outputStream.write(0x00);
```

После каждой выводимой строки в этом коде выводится пустой символ (`0x00`). Мы можем извлечь такое дублирование, создав метод `writeField` (вывести поле), который воспринимает строку и выходной поток, а затем выводит эту строку в выходной поток, завершая ее пустым символом.

```
void writeField(OutputStream outputStream, String field) {
    outputStream.write(field.getBytes());
    outputStream.write(0x00);
}
```

С чего же начать

Реорганизуя код несколько раз подряд, чтобы исключить его дублирование, мы можем получить в конечном итоге разные структуры в зависимости от того, с чего мы начнем. Допустим, что у нас имеется следующий метод:

```
void c() { a(); a(); b(); a(); b(); b(); }
```

Его можно реорганизовать следующим образом:

```
void c() { aa(); b(); a(); bb(); }
```

или же так:

```
void c() { a(); ab(); ab(); b(); }
```

Какой же вариант выбрать? С точки зрения структуры кода это на самом деле не имеет никакого значения. Оба варианта группирования кода все равно выглядят предпочтительнее исходного, а если потребуется, то мы можем реорганизовать код в другие группы. Эти решения не окончательные. В данном случае основное внимание следует уделить используемым именам. Если для двух повторяющихся вызовов метода `a()` удастся подобрать имя, которое в данном контексте имеет больший смысл, чем имя для вызова метода `a()`, после которого следует вызов метода `b()`, то нужно использовать именно его.

Кроме того, начинать лучше всего с малого. Если можно удалить мелкие фрагменты дублированного кода, то это следует сделать в первую очередь, поскольку таким образом чаще всего проясняется общая картина.

Имея в своем распоряжении такой метод, мы можем заменить им каждую пару операторов вывода строки и пустого символа, периодически выполняя тесты, чтобы убедиться в том, что мы ничего не нарушили в коде. Ниже приведен код метода `write` из класса `LoginCommand` после внесенных изменений.

```
public void write(OutputStream outputStream)
            throws Exception {
    outputStream.write(header);
    outputStream.write(getSize());
    outputStream.write(commandChar);
    writeField(outputstream, username);
    writeField(outputStream, passwd);
    outputStream.write(footer);
}
```

Подобным способом устраняется дублирование кода в классе `LoginCommand`, но такой способ не совсем подходит для класса `AddEmployeeCmd`, где имеются аналогичные повторяющиеся последовательности вывода строки и пустого символа в таком же методе `write`. Оба класса представляют собой команды, и поэтому мы можем ввести для них суперкласс `Command`. Имея в своем распоряжении суперкласс, можно извлечь в него метод `writeField`, чтобы использовать данный метод в классах обеих команд, как показано на рис. 21.2.

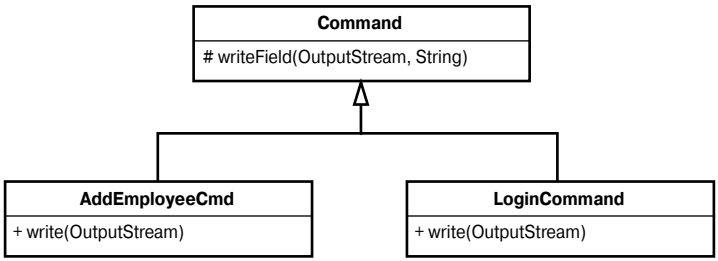


Рис. 21.2. Иерархия классов команд

Теперь мы можем вернуться к классу `AddEmployeeCmd` и заменить все операторы вывода строки и пустого символа вызовами метода `writeField`. После этого метод `write` из класса `AddEmployeeCmd` будет выглядеть следующим образом:

```
public void write(OutputStream outputStream)
            throws Exception {
    outputStream.write(header);
    outputStream.write(getSize());
    outputStream.write(commandChar);
    writeField(outputStream, name);
    writeField(outputStream, address);
    writeField(outputStream, city);
    writeField(outputStream, state);
    writeField(outputStream, yearlySalary);
    outputStream.write(footer);
}
```

А метод `write` из класса `LoginCommand` примет следующий вид:

```
public void write(OutputStream outputStream)
            throws Exception {
    outputStream.write(header);
    outputStream.write(getSize());
    outputStream.write(commandChar);
    writeField(outputstream, userName);
    writeField(outputStream, passwd);
    outputStream.write(footer);
}
```

Теперь код стал немного яснее, но это еще не все. Методы `write` из классов `AddEmployeeCmd` и `LoginCommand` действуют по одинаковому образцу: вывод заголовка, размера строки, символьного обозначения команды, затем целого ряда полей и, наконец, окончания. Мы можем извлечь отличия в этих методах, т.е. вывод полей, таким образом, чтобы метод `write` из класса `LoginCommand` принял следующий вид:

```
public void write(OutputStream outputStream)
            throws Exception {
    outputStream.write(header);
    outputStream.write(getSize());
    outputStream.write(commandChar);
    writeBody(outputstream);
    outputStream.write(footer);
}
```

Ниже приведен код, извлеченный в метод `writeBody` (вывод тела).

```
private void writeBody(OutputStream outputStream)
             throws Exception {
    writeField(outputstream, userName);
    writeField(outputStream, passwd);
}
```

Метода `write` из класса `AddEmployeeCmd` выглядит точно так же, но для него метод `writeBody` получается несколько иным.

```
private void writeBody(OutputStream outputStream) throws Exception {
    writeField(outputStream, name);
    writeField(outputStream, address);
    writeField(outputStream, city);
    writeField(outputStream, state);
    writeField(outputStream, yearlySalary);
}
```

Если два метода выглядят почти одинаково, извлеките их отличия в другие методы. После этого оба метода становятся совершенно одинаковыми и от одного из них можно просто избавиться.

Теперь методы `write` из обоих классов выглядят совершенно одинаково. Можем ли мы перенести метод `write` в класс `Command`? Пока еще не можем. Несмотря на полное сходство обоих методов `write`, в них используются данные, характерные для тех классов, к которым они принадлежат: переменные `header` (заголовок), `footer` (окончание) и `commandChar` (символьное обозначение команды). Если мы хотим объединить их в единый метод `write`, то нам придется организовать в нем вызовы методов из соот-

ветствующих подклассов для получения этих данных. Рассмотрим переменные в классах `AddEmployeeCmd` и `LoginCommand`.

```
public class AddEmployeeCmd extends Command {
    String name;
    String address;
    String city;
    String state;
    String yearlySalary;
    private static final byte[] header
                = {(byte)0xde, (byte)0xad};
    private static final byte[] commandChar = {0x02};
    private static final byte[] footer
                = {(byte)0xbe, (byte)0xef};
    private static final int SIZE_LENGTH = 1;
    private static final int CMD_BYTE_LENGTH = 1;
    ...
}

public class LoginCommand extends Command {
    private String userName;
    private String passwd;

    private static final byte[] header
                = {(byte)0xde, (byte)0xad};
    private static final byte[] commandChar = {0x01};
    private static final byte[] footer
                = {(byte)0xbe, (byte)0xef};
    private static final int SIZE_LENGTH = 1;
    private static final int CMD_BYTE_LENGTH = 1;
    ...
}
```

В обоих классах имеется немало общих данных. В частности, мы можем извлечь переменные `header`, `footer`, `SIZE_LENGTH` (длина строки) и `CMD_BYTE_LENGTH` (длина команды в байтах) в класс `Command`, поскольку у всех этих переменных одинаковые значения. Их следует сделать на время защищенными, чтобы перекомпилировать и протестировать код.

```
public class Command {
    protected static final byte[] header
                = {(byte)0xde, (byte)0xad};
    protected static final byte[] footer
                = {(byte)0xbe, (byte)0xef};
    protected static final int SIZE_LENGTH = 1;
    protected static final int CMD_BYTE_LENGTH = 1;
    ...
}
```

Итак, в обоих подклассах осталась одна сходная переменная `commandChar`, но у нее разные значения в каждом из них. В качестве простого выхода этого положения можно,

в частности, ввести в класс Command следующий абстрактный метод получения или так называемый получатель:

```
public class Command {
    protected static final byte[] header
                = {(byte)0xde, (byte)0xad};
    protected static final byte[] footer
                = {(byte)0xbe, (byte)0xef};
    protected static final int SIZE_LENGTH = 1;
    protected static final int CMD_BYTE_LENGTH = 1;
    protected abstract char [] getCommandChar();
    ...
}
```

Теперь мы можем заменить переменные commandChar в каждом из подклассов переопределением метода getCommandChar.

```
public class AddEmployeeCmd extends Command {
    protected char [] getCommandChar() {
        return new char [] { 0x02};
    }
    ...
}

public class LoginCommand extends Command {
    protected char [] getCommandChar() {
        return new char [] { 0x01};
    }
    ...
}
```

Вот теперь мы можем благополучно перенести метод write в класс Command. После этого класс Command примет следующий вид:

```
public class Command {
    protected static final byte[] header
                = {(byte)0xde, (byte)0xad};
    protected static final byte[] footer
                = {(byte)0xbe, (byte)0xef};
    protected static final int SIZE_LENGTH = 1;
    protected static final int CMD_BYTE_LENGTH = 1;

    protected abstract char [] getCommandChar();

    protected abstract void writeBody(OutputStream outputStream);

    protected void writeField(OutputStream outputStream,
                              String field) {
    outputStream.write(field.getBytes());
    outputStream.write(0x00);
    }

    public void write(OutputStream outputStream)
```

```
                throws Exception {
        outputStream.write(header);
        outputStream.write(getSize());
        outputStream.write(commandChar);
        writeBody(outputstream);
        outputStream.write(footer);
    }
}
```

Обратите внимание на то, что нам пришлось ввести абстрактный метод для метода `writeBody` и поместить его в класс `Command` (рис. 21.3).

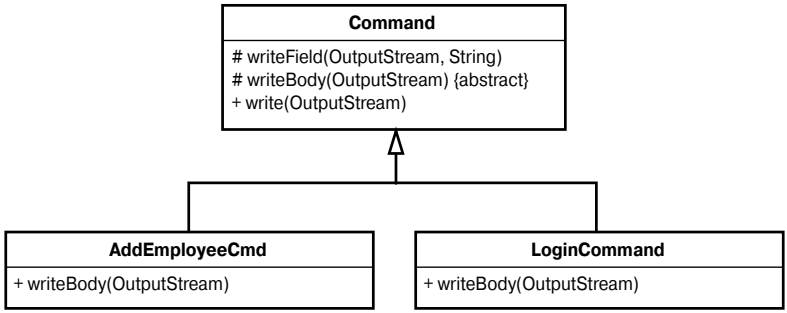


Рис. 21.3. Перенос методов `writeField`, `writeBody` и `write` в суперкласс

После переноса метода `write` в суперкласс нам остается только реорганизовать в каждом из подклассов методы `getSize`, метод `getCommandChar` и конструкторы. В настоящий момент класс `LoginCommand` выглядит следующим образом.

```
public class LoginCommand extends Command {
    private String userName;
    private String passwd;

    public LoginCommand(String userName, String passwd) {
        this.userName = userName;
        this.passwd = passwd;
}

protected char [] getCommandChar() {
    return new char [] { 0x01};
}

protected int getSize() {
    return header.length + SIZE_LENGTH + CMD_BYTE_LENGTH +
        footer.length + userName.getBytes().length + 1 +
        passwd.getBytes().length + 1;
    }
}
```

Это довольно стройный класс. Аналогично выглядит и класс `AddEmployeeCmd`. В обоих классах имеются лишь методы `getSize` и `getCommandChar` и больше ничего

существенного. Но присмотримся к методам `getSize` в этих классах внимательнее. Ниже приведен метод `getSize` из класса `LoginCommand`.

```
protected int getSize() {
    return header.length + SIZE_LENGTH +
        CMD_BYTE_LENGTH + footer.length +
        userName.getBytes().length + 1 +
        passwd.getBytes().length + 1;
}
```

А вот как он выглядит в классе `AddEmployeeCmd`.

```
private int getSize() {
    return header.length + SIZE_LENGTH +
        CMD_BYTE_LENGTH + footer.length +
        name.getBytes().length + 1 +
        address.getBytes().length + 1 +
        city.getBytes().length + 1 +
        state.getBytes().length + 1 +
        yearlySalary.getBytes().length + 1;
}
```

Что у этих методов общего и чем они отличаются? По-видимому, в обоих методах суммируется заголовок, длина строки, длина команды в байтах, окончание, а также длина каждого поля. А если извлечь отличия обоих методов в вычислении длины полей и поместить их в отдельный метод? Такой метод можно назвать `getBodySize()` — получить длину тела команды.

```
private int getSize() {
    return header.length + SIZE_LENGTH
        + CMD_BYTE_LENGTH + footer.length + getBodySize();
}
```

Сделав это, мы получим в каждом методе тот же самый код, выполняющий суммирование длины всех учетных данных, а также длины тела команды, представляющей собой сумму длин всех полей. После этого мы можем перенести метод `getSize` в класс `Command` и получить разные реализации метода `getBodySize` в каждом из подклассов (рис. 21.4).

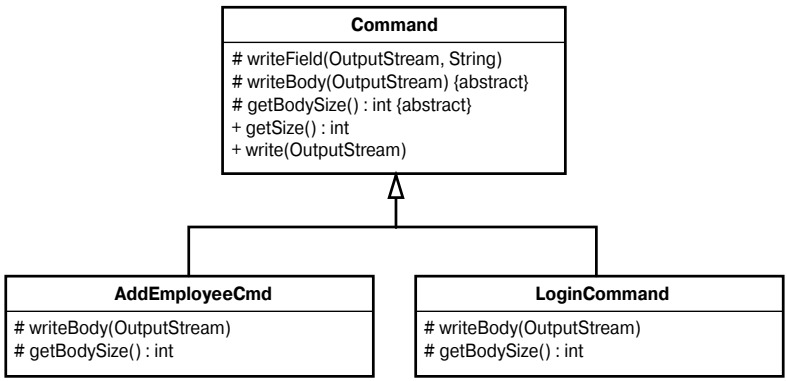


Рис. 21.4. Перенос метода `getBodySize` в суперкласс

Итак, мы пришли к тому, что теперь у нас имеется следующая реализация метода `getBody` в классе `AddEmployeeCmd`:

```
protected int getBodySize() {
    return name.getBytes().length + 1 +
        address.getBytes().length + 1 +
        city.getBytes().length + 1 +
        state.getBytes().length + 1 +
        yearlySalary.getBytes().length + 1;
}
```

Мы упустили из виду явное дублирование кода в этом методе. И хотя оно незначительно, постараемся устранить его полностью.

```
protected int getFieldSize(String field) {
    return field.getBytes().length + 1;
}

protected int getBodySize() {
    return getFieldSize(name) +
           getFieldSize(address) +
           getFieldSize(city) +
           getFieldSize(state) +
           getFieldSize(yearlySalary);
}
```

Если мы перенесем метод `getFieldSize` (получить длину поля) в класс `Command`, то сможем использовать его и в методе `getBodySize` из класса `LoginCommand`.

```
protected int getBodySize() {
    return getFieldSize(name) + getFieldSize(password);
}
```

Остались ли в рассматриваемом здесь коде еще какие-нибудь дублированные места? В действительности, они имеются, хотя их осталось немного. Оба класса `LoginCommand` и `AddEmployeeCmd` воспринимают список параметров, получают их длину и затем выводят их в выходной поток. За исключением переменной `commandChar`, все остальные отличия заключаются именно в этом списке. Что, если исключить дублирование за счет незначительного обобщения? Если мы объявим данный список в базовом классе, то сможем ввести его в конструктор каждого подкласса следующим образом:

```
class LoginCommand extends Command
{
    ...
    public AddEmployeeCmd(String name, String password) {
        fields.add(name);
        fields.add(password);
    }
    ...
}
```

Введя список `fields` (поля) в каждый подкласс, мы можем использовать один и тот же код для расчета длины тела команды.

```
int getBodySize() {
    int result = 0;
    for(Iterator it = fields.iterator(); it.hasNext(); ) {
        String field = (String)it.next();
        result += getFieldSize(field);
    }
    return result;
}
```

Аналогично, метод `writeBody` может выглядеть следующим образом:

```
void writeBody(Outputstream outputstream) {
    for(Iterator it = fields.iterator(); it.hasNext(); ) {
        String field = (String)it.next();
        writeField(outputStream, field);
    }
}
```

Оба приведенных выше метода можно перенести в суперкласс. Сделав это, мы можем практически полностью исключить дублирование в рассматриваемом здесь коде. Ниже показано, как после этого выглядит класс `Command`. В этом классе все методы, не доступные больше в подклассах, благоразумно сделаны частными.

```
public class Command {
    private static final byte[] header
                = {(byte)0xde, (byte)0xad};
    private static final byte[] footer
                = {(byte)0xbe, (byte)0xef};
    private static final int SIZE_LENGTH = 1;
    private static final int CMD_BYTE_LENGTH = 1;

    protected List fields = new ArrayList();
    protected abstract char [] getCommandChar();

    private void writeBody(Outputstream outputstream) {
        for(Iterator it = fields.iterator(); it.hasNext(); ) {
            String field = (String)it.next();
            writeField(outputStream, field);
        }
    }

    private int getFieldSize(String field) {
        return field.getBytes().length + 1;
    }

    private int getBodySize() {
        int result = 0;
        for(Iterator it = fields.iterator(); it.hasNext(); ) {
            String field = (String)it.next();
            result += getFieldSize(field);
        }
        return result;
    }
```

```
    private int getSize() {
        return header.length + SIZE_LENGTH
            + CMD_BYTE_LENGTH + footer.length
            + getBodySize();
    }

    private void writeField(OutputStream outputStream,
                            String field) {
        outputStream.write(field.getBytes());
        outputStream.write(0x00);
    }

    public void write(OutputStream outputStream)
                throws Exception {
        outputStream.write(header);
        outputStream.write(getSize());
        outputStream.write(commandChar);
        writeBody(outputstream);
        outputStream.write(footer);
    }
}
```

Классы `LoginCommand` и `AddEmployeeCmd` теперь выглядят необычно мелкими.

```
public class LoginCommand extends Command {
    public LoginCommand(String userName, String passwd) {
        fields.add(username);
        fields.add(passwd);
    }

    protected char [] getCommandChar() {
        return new char [] { 0x01};
    }
}

public class AddEmployeeCmd extends Command {
    public AddEmployeeCmd(String name, String address,
                          String city, String state,
                          int yearlySalary) {
        fields.add(name);
        fields.add(address);
        fields.add(city);
        fields.add(state);
        fields.add(Integer.toString(yearlySalary));
    }

    protected char [] getCommandChar() {
        return new char [] { 0x02 };
    }
}
```

На рис. 21.5 приведена блок-схема UML, демонстрирующая итог, к которому мы пришли после реорганизации кода, чтобы исключить его дублирование.

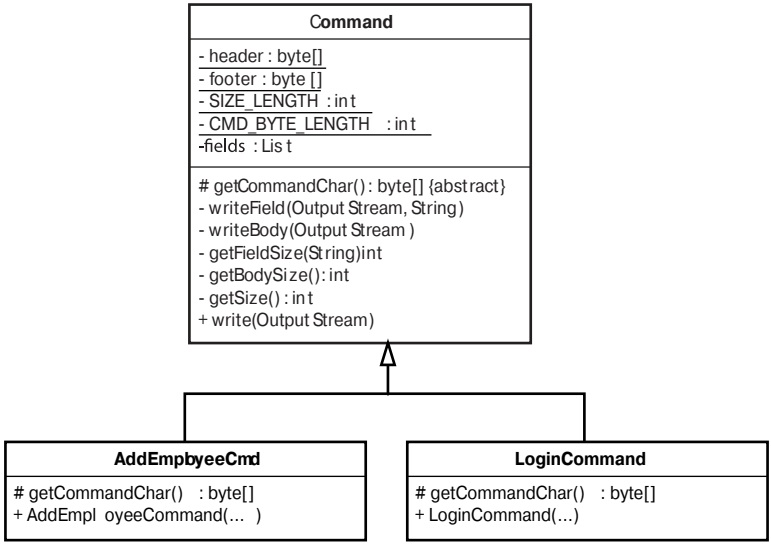


Рис. 21.5. Иерархия классов команд после исключения дублирования кода

Итак, к чему же мы пришли? Мы настолько исключили дублирование кода, что от классов команд остались одни оболочки, а все их функции перенесены в класс `Command`. И в этой связи возникает вопрос: нужно ли вообще разделять оба класса команд? Имеется ли этому какая-то другая альтернатива?

Мы могли бы избавиться от подклассов и ввести в класс `Command` статический метод, позволяющий посылать команды.

```
List arguments = new ArrayList();
arguments.add("Mike");
arguments.add("asdsad");
Command.send(stream, 0x01, arguments);
```

Но это добавило бы хлопот клиентам данного класса. В связи с этим очевидно одно: нам требуется послать два разных символьных обозначения команд, но мы не хотим переносить бремя этих хлопот на пользователей.

Вместо этого мы могли бы ввести другой статический метод для каждой команды, которую требуется послать.

```
Command.SendAddEmployee(stream,
        "Mike", "122 Elm St", "Miami", "FL", 10000);

Command.SendLogin(stream, "Mike", "asdsad");
```

Но тогда нам пришлось бы вносить изменения в клиентский код. В настоящий момент в рассматриваемом здесь коде имеется немало мест, где мы могли бы построить объекты классов `LoginCommand` и `AddEmployeeCmd`.

Возможно, нам лучше оставить классы команд в том виде, в каком они существуют теперь. Конечно, подклассы получились довольно мелкими, но так ли это важно? На самом деле не очень важно.

Можно ли считать дело сделанным? Нет, нельзя. Нам осталось еще незначительное изменение, которого не следовало делать раньше. Теперь мы можем переименовать подкласс

AddEmployeeCmd в AddEmployeeCommand, чтобы согласовать имена обоих подклассов и тем самым избежать путаницы при их использовании.

О сокращениях

Сокращения в именах классов и методов не всегда оказываются благом. Они удобны, если используются согласованно, но я лично стараюсь их избегать.

Разработчики одной из групп, с которой мне пришлось работать, однажды попытались использовать слова *manager* (управляющий) *management* (управление) в имени буквально каждого класса проектируемой системы. Такие условные обозначения не очень им помогли. Более того, используя сокращения этих слов в самых разных сочетания, они только усугубили положение. Например, одни классы были названы XXXXMgr, а другие — XXXXMngr. Когда я просматривал код, написанный этими разработчиками, более чем в половине случаев я так и не смог догадаться по суффиксу, какой именно класс в нем используется.

Итак, мы полностью исключили дублирование кода. Стал ли он от этого лучше или хуже? Проанализируем пару возможных ситуаций. Что нам делать, если потребуется ввести команду? Для этого нам достаточно выполнить подклассификацию класса Command и создать объект требуемой команды. Сравним эти действия с тем, что нам пришлось бы делать в первоначальном варианте кода. Мы могли бы создать новую команду, а затем вырезать и скопировать код из другой команды, изменив все ее переменные. Но если бы мы это сделали, то внесли бы дополнительное дублирование кода и только усугубили бы положение. Кроме того, такие действия могли бы привести к ошибкам. В частности, перепутав переменные, мы использовали бы их неправильно. Нам определенно потребовалось бы больше времени на все эти действия в прежнем варианте кода до исключения в нем дублирования.

А теперь посмотрим, насколько исключение дублирования кода делает его менее удобным в обращении. Если нам потребуется послать команды, состоящие из каких-то других элементов, кроме строк? Это задачу мы в известной степени уже решили. Класс AddEmployeeCommand теперь воспринимает целое значение, которое затем преобразуется в строку для отправки в качестве команды. Аналогичное преобразование мы можем сделать и с любым другим типом данных. Нам придется преобразовать его каким-то образом в строку, чтобы затем отправить. И это мы можем сделать в конструкторе любого нового подкласса.

А если команда имеет другой формат? Допустим, что нам требуется новый тип команды, в тело которой могут быть вложены другие команды. Для этого нам достаточно выполнить подклассификацию класса Command и переопределить его метод writeBody.

```
public class AggregateCommand extends Command
{
    private List commands = new ArrayList();
    protected char [] getCommandChar() {
        return new char [] { 0x03 };
    }

    public void appendCommand(Command newCommand) {
        commands.add(newCommand);
    }

    protected void writeBody(OutputStream out) {
```

```
        out.write(commands.getSize());
        for(Iterator it = commands.iterator(); it.hasNext(); ) {
            Command innerCommand = (Command)it.next();
            innerCommand.write(out);
        }
    }
}
```

Вся остальная часть кода остается без изменений и действует как и прежде. Только представьте, что бы нам пришлось делать, если бы мы не исключили дублирование в рассматриваемом здесь коде.

Этот последний пример ярко высвечивает следующее очень важное обстоятельство: когда исключается дублирование кода в классах, в итоге получаются очень компактные и конкретные методы. Каждый из них выполняет то, что не делают остальные методы, а это дает огромное преимущество, заключающееся в ортогональности.

Ортогональность — это замысловатое обозначение более простого понятия независимости. Так, если требуется изменить существующее поведение и для этого в коде имеется только одно место, это означает, что такому коду присуща ортогональность. Это все равно, как представить систему в виде большого ящика с множеством рукояток. Если на каждое поведение системы приходится одна рукоятка, то внести в нее изменения нетрудно. Если же в системе присутствует явное дублирование кода, на каждое ее поведение приходится уже более чем одна рукоятка. Вернемся к рассмотренному выше примеру вывода полей в выходной поток. Если бы в первоначальном варианте кода нам потребовалось использовать `0x01` вместо `0x00` в качестве признака окончания, то для этого нам пришлось бы просмотреть весь код и внести соответствующие изменения во многих его местах. А если бы потребовалось выводить два признака окончания `0x00` на каждое поле, ситуация оказалась бы еще хуже, поскольку на такое поведения не приходится единственная рукоятка, если прибегнуть к упомянутой выше аналогии проектируемой системы с большим ящиком. С другой стороны, в коде, который мы реорганизовали, нам достаточно поправить и переопределить метод `writeField`, если потребуется изменить порядок вывода полей, а для таких особых случаев, как агрегирование команд, нам было бы достаточно переопределить метод `writeBody`. Когда поведение локализовано в отдельных методах, его нетрудно изменить или же дополнить.

В рассмотренном ранее примере нам пришлось выполнить немало разных действий, включая перенос методов и переменных из одного класса в другой и разделение методов на части, но большая часть этих действий носила механический характер. Мы просто обращали внимание на дублирование кода и исключали его. Единственное, что можно было бы отнести к творчеству среди всех этих действий, — это выбор подходящих имен для новых методов. В первоначальном варианте кода вообще не упоминается о телах полей и команд, хотя эти понятия и присутствуют в самом коде. Например, некоторые переменные трактуются иначе и называются полями. В итоге нам удалось получить более аккуратную ортогональную структуру кода, но при этом у нас не возникало ощущения, будто мы что-то проектируем. Это было похоже, скорее, на анализ недостатков, обнаруженных в коде, и усовершенствование кода максимально близко к самой его сути.

Самое удивительное в том, что обнаруживается, когда вы начинаете усердно исключать дублирование кода, — это появление новых структур кода. Вы не планируете появление множества рукояток на ящике, символизирующем систему, — они просто появляются сами, а это не очень хорошо. Например, было бы неплохо, если бы в классе `Command` метод

```
public void write(OutputStream outputStream)
            throws Exception {
    outputStream.write(header);
    outputStream.write(getSize());
    outputStream.write(commandChar);
    writeBody(outputstream);
    outputStream.write(footer);
}
```

выглядел бы следующим образом:

```
public void write(OutputStream outputStream)
            throws Exception {
    writeHeader(outputStream);
    writeBody(outputstream);
    writeFooter(outputStream);
}
```

Теперь у нас имеется одна рукоятка для вывода заголовков и другая рукоятка для вывода окончаний. Мы можем добавлять рукоятки в систему по мере надобности, но лучше, если бы они появлялись естественным путем.

Исключение дублирования — эффективный способ дистиллирования кода. Оно не только помогает сделать структуру кода более гибкой, но и ускоряет и упрощает внесение в него изменений.

Принцип открытости-закрытости

Принцип открытости-закрытости (OCP) был впервые сформулирован Бертраном Мейером. Он состоит в том, что код должен быть открытым для расширения, но закрытым для модификации. Это означает, что если у нас имеется хорошая структура кода, то для ввода новых свойств нам не нужно существенно изменять код.

Обладает ли код, который мы реорганизовали в этой главе, подобными качествами? Да, обладает. Мы только что рассмотрели возможные варианты изменения этого кода, и во многих из них код пришлось бы изменить лишь в некоторых методах, а в ряде случаев для ввода новых свойств нам было бы достаточно выполнить подклассификацию. Разумеется, после подклассификации очень важно исключить дублирование (подробнее о том, как новые свойства вводятся с помощью подклассификации и интегрируются в код путем его реорганизации, речь идет в разделе "Программирование по разности" главы 8.)

Когда мы исключаем дублирование в коде, в нем зачастую начинает вполне естественным путем соблюдаться принцип открытости-закрытости.

Глава 22

Необходимо изменить гигантский метод, но нельзя написать для него тест

Одна из самых больших трудностей в работе с унаследованным кодом возникает в том случае, когда приходится иметь дело с крупными и длинными методами. Зачастую мы можем избежать реорганизации кода длинных методов, используя *почкование метода* или же *почкование класса*. Но даже если нам и удастся избежать реорганизации кода, не делать этого просто непростительно. База кода длинных методов настолько запутанна, что в ней нетрудно увязнуть, как в болоте. Когда требуется изменить подобные методы, сначала приходится разбираться в их назначении и функциях и только после этого вносить изменения. Как правило, для изменения длинных методов требуется больше времени, чем в том случае, если бы пришлось иметь дело с более понятным кодом.

Если длинные методы — сущее наказание, то гигантские методы еще хуже. Гигантским называется такой метод, который настолько длинен и сложен, что его не хочется даже трогать. Гигантские методы могут состоять из сотен или даже тысяч строк кода с настолько редкими отступами, что читать их просто невозможно. Когда приходится иметь дело с гигантскими методами, так и подмывает распечатать их на длинном рулоне бумаги, расстелить распечатку на полу в коридоре и улечься на нее вместе с коллегами, чтобы начать разбираться в этом чудовищном коде.

Однажды, когда мы с коллегой возвращались в свой отель после очередного совещания с группой разработчиков, он сказал мне: “Ты должен это увидеть”. Он повел меня в свой номер, вытащил переносной компьютер и показал мне метод, состоявший более чем из тысячи строк кода. Моему коллеге было известно, что я занимаюсь вопросами реорганизации кода, и поэтому он спросил меня: “Как же реорганизовать такой код?” И мы задумались над этим вопросом. Мы хорошо понимали, что самое главное — организовать тестирование кода, но откуда его начать в таком гигантском методе?

В этой главе я попытаюсь поделиться с вами опытом, который я с тех пор приобрел, имея дело с гигантскими методами.

Разновидности гигантских методов

Существуют две разновидности гигантских методов. Но провести четкое различия между ними удается далеко не всегда. Зачастую они представляют определенное сочетание основных разновидностей подобно гибридам.

Методы без отступов

Метод без отступов практически не содержит отступов в своем исходном коде. Он представляет собой последовательность фрагментов кода, очень похожую на размеченный список. Некоторые фрагменты кода могут присутствовать в таком методе с отступами, но большая часть его исходного кода не содержит отступы. Если посмотреть на метод без отступов прищурясь, то можно увидеть нечто, похожее на рис. 22.1.

```
void Reservation::extend(int additionalDays)
{
    int status = RIXInterface::checkAvailable(type, location,
startingDate);

    int identCookie = -1;
    switch(status) {
        case NOT_AVAILABLE_UPGRADE_LUXURY:
            identCookie = RIXInterface::holdReservation(Luxury,location,
                                                        startingDate,
                additionalDays +ddditionalDays);
            break;
        case NOT_AVAILABLE_UPGRADE_SUV:
        {
            int theDays = additionalDays + additionalDays;
            if (RIXInterface::getOpCode(customerID) != 0)
                theDays++;
            identCookie = RIXInterface::holdReservation(SUV,location,
                                                startingDate, theDays);
        }
        break;
        case NOT_AVAILABLE_UPGRADE_VAN:
            identCookie = RIXInterface::holdReservation(Van,
                location,startingDate, additionalDays + additionalDays);
            break;
        case AVAILABLE:
        default:
            RIXInterface::holdReservation(type,location,startingDate);
        break;
    }

    if (identCookie != -1 && state == Initial) {
        RIXInterface::waitlistReservation(type,location,startingDate);
    }

    Customer c = res_db.getCustomer(customerID);

    if (c.vipProgramStatus == VIP_DIAMOND) {
        upgradeQuery = true;
    }

    if (!upgradeQuery)
```

```
        RIXInterface::extend(lastCookie, days + additionalDays);
    else {
        RIXInterface::waitlistReservation(type,location,startingDate);
        RIXInterface::extend(lastCookie, days + additionalDays +1);
    }
    ...
}
```

Рис. 22.1. Метод без отступов

Это обычная форма метода без отступов. Если вам повезет, то вы сможете обнаружить в таком методе дополнительные строки, разделяющие отдельные части кода, или комментарии, обозначающие конкретное назначение этих частей кода. В идеальном случае вам удастся извлечь метод из каждой части исходного метода без отступов, но зачастую реорганизовывать такие методы очень трудно. Пробел между отдельными частями кода в таком методе может оказаться несколько обманчивым, поскольку временные переменные нередко объявляются в одних частях кода, а используются в других. Разделение подобного гигантского метода на части зачастую проще сделать обыкновенным копированием и вставкой кода. Несмотря на это, методы без отступов выглядят не так пугающе, как другие разновидности гигантских методов, главным образом, потому что отсутствие в них странных отступов позволяет нам лучше ориентироваться в их коде.

Запутанные методы

Запутанным называется такой метод, в котором преобладает одна крупная часть кода, набранная с отступом. В простейшем случае запутанный метод содержит один крупный условный оператор, как показано на рис. 22.2.

```
Reservation::Reservation(VehicleType type, int customerID, long
                         startingDate, int days, XLocation l)
: type(type), customerID(customerID), startingDate(startingDate),
days(days), lastCookie(-1),
state(Initial), tempTotal(0)
{
    location = l;
    upgradeQuery = false;

    if (!RIXInterface::available()) {
        RIXInterface::doEvents(100);
        PostLogMessage(0, 0, "delay on reservation creation");
        int holdCookie = -1;
        switch(status) {
            case NOT_AVAILABLE_UPGRADE_LUXURY:
                holdCookie = RIXInterface::holdReservation(Luxury,l,
                                                           startingDate);
                if (holdCookie != -1) {
                    holdCookie |= 9;
                }
                break;
            case NOT_AVAILABLE_UPGRADE_SUV:
```

```
                holdCookie = RIXInterface::holdReservation(SUV,l,
                                                           startingDate);
                break;
            case NOT_AVAILABLE_UPGRADE_VAN:
                holdCookie = RIXInterface::holdReservation(Van,l,
                                                           startingDate);
                break;
            case AVAILABLE:
            default:
                RIXInterface::holdReservation;
                state = Held;
                break;
        }
    }
    ...

}
```

Рис. 22.2. Простой запутанный метод

Но подобного рода запутанность оказывается практически такого же свойства, как и у метода без отступов. Запутанные методы, требующие настоящего умения, чтобы полностью разобраться в них, имеют форму, приведенную на рис. 22.3.

Для того чтобы выяснить, имеете ли вы дело с запутанным методом, попробуйте выровнять блоки какого-нибудь длинного метода. Если после этого у вас начнет кружиться голова, значит, вам попался действительно запутанный метод.

```
Reservation::Reservation(VehicleType type, int customerID, long starting-
Date, int days, XLocation l)
: type(type), customerID(customerID), startingDate(startingDate),
days(days), lastCookie(-1),
state(Initial), tempTotal(0)
{
    location = l;
    upgradeQuery = false;

    while(!RIXInterface::available()) {
        RIXInterface::doEvents(100);
        PostLogMessage(0, 0, "delay on reservation creation");
        int holdCookie = -1;
        switch(status) {
            case NOT_AVAILABLE_UPGRADE_LUXURY:
                holdCookie =
                RIXInterface::holdReservation(Luxury,l,startingDate);
                if (holdCookie != -1) {
                    if (l == GIG && customerID == 45) {
                        // Особый #1222
                        while (RIXInterface::notBusy()) {
                            int code =
                             RIXInterface::getOpCode(customerID);
                            if (code == 1 || customerID > 0)) {
                                PostLogMessage(1, 0, "QEX PID");
```

```
                                for (int n = 0; n < 12; n++) {
                                    int total = 2000;
                                    if (state == Initial || state == Held)
                                    {
                                     total += getTotalByLocation(location);
                                     tempTotal = total;
                                     if (location == GIG && days > 2)
                                        {
                                            if (state == Held)
                                                total += 30;
                                        }
                                    }
                                    RIXInterface::serveIDCode(n, total);
                                }
                            } else {
                                RIXInterface::serveCode(customerID);
                            }
                        }
                    }
                }
                break;
            case NOT_AVAILABLE_UPGRADE_SUV:
                holdCookie =
                RIXInterface::holdReservation(SUV,1,startingDate);
               break;
            case NOT_AVAILABLE_UPGRADE_VAN:
                holdCookie =
                RIXInterface::holdReservation(Van,1,startingDate);
                break;
            case AVAILABLE:
            default:
                RIXInterface::holdReservation(type,1,startingDate);
                state = Held;
                break;
        }
    }
    ...
}
```

Рис. 22.3. Очень запутанный метод

Большая часть гигантских методов занимает промежуточное положение между двумя крайними разновидностями методов без отступов и запутанных методов. Так, во многих запутанных методах имеются длинные части кода без отступов, но поскольку они глубоко вложены, для них очень трудно написать тесты, чтобы выявить их поведение. В этом отношении работа с запутанными методами представляет особые трудности.

Для реорганизации кода длинных методов имеет большое значение наличие или отсутствие инструментального средства реорганизации кода. Практически все подобные средства поддерживают реорганизацию кода извлечением метода, поскольку такая поддержка открывает очень большие возможности. Ведь если инструментальное средство реорганизации кода способно извлекать методы безопасно, то для проверки извлеченного

кода тесты не требуются. Такое средство проводит анализ кода автоматически, и поэтому остается лишь научиться правильно использовать извлеченный код, чтобы привести метод в удобную форму для последующей работы с ним.

Если же поддержка извлечения метода отсутствует, то привести гигантский метод в порядок будет труднее. Обращаться с таким методом приходится более осторожно, поскольку это связано с выполнением тестов, которые могут быть размещены по местам.

Обращение с гигантскими методами при поддержке автоматической реорганизации кода

Пользуясь инструментальным средством, извлекающим методы автоматически, следует ясно осознавать его возможности. Современные инструментальные средства реорганизации кода выполняют простое извлечение методов и целый ряд других видов реорганизации кода, но они не проводят при этом все те дополнительные операции, которые обычно нам требуются, когда мы разделяем крупные методы на части. Например, у нас нередко возникает искушение переупорядочить операторы, чтобы сгруппировать их для последующего извлечения. Ни одно из современных инструментальных средств данного типа не проводит анализ, необходимый для того, чтобы выполнить такое переупорядочение безопасно. И это прискорбно, поскольку подобная операция чревата программными ошибками.

Для эффективного применения инструментальных средств реорганизации кода в работе с гигантскими методами изменения целесообразно вносить по отдельности с помощью такого средства, избегая всех остальных видов правки исходного кода. Такой подход можно сравнить с правкой кода одной рукой, держа другую за спиной, тем не менее он позволяет провести отчетливую границу между теми изменениями, которые безопасны изначально, и теми изменениями, о которых этого нельзя сказать заранее. При такой реорганизации кода следует избегать даже таких простых правок, как упорядочение операторов и разделение выражений на части. Если инструментальное средство поддерживает переименование переменных, то и прекрасно, а если не поддерживает, то его можно отложить напоследок.

Автоматическую реорганизацию кода следует выполнять только специально предназначенным для этого инструментальным средством. После ряда автоматических реорганизаций кода обычно тесты размещаются по местам, чтобы проверить с их помощью правки кода, сделанные вручную.

При извлечении кода преследуются следующие основные цели.

1. Отделение логики от неудобных зависимостей.
2. Введение швов, упрощающих размещение тестов по местам для дополнительной реорганизации кода.

Обратимся к следующему примеру.

```
class CommoditySelectionPanel
{
    ...
    public void update() {
        if (commodities.size() > 0
                && commodities.GetSource().equals("local")) {
```

```
            listbox.clear();
            for (Iterator it = commodities.iterator();
                    it.hasNext(); ) {
                Commodity current = (Commodity)it.next();
                if (commodity.isTwilight()
                        && !commodity.match(broker))
                    listbox.add(commodity.getView());
            }
        }
        ...
    }
    ...
}
```

В этом методе имеется немало мест для правки. Самое удивительное, что определенного рода фильтрация происходит в классе панели, который должен бы идеально подходить только для отображения. Распутать такой код не так-то просто. Если приступить к написанию тестов для данного метода в его исходном виде, то тесты можно было бы написать для проверки состояния объекта спискового окна, но это вряд ли помогло бы нам продвинуться далеко в улучшении структуры кода.

При наличии инструментальной поддержки реорганизации кода мы можем начать с переименования отдельно извлекаемых частей верхнего уровня в данном методе, одновременно разрывая зависимости. Вот как может выглядеть код после ряда извлечений.

```
class CommoditySelectionPanel
{
    ...
    public void update() {
        if (commoditiesAreReadyForUpdate()) {
            clearDisplay();
            updateCommodities();
        }
        ...
    }

    private boolean commoditiesAreReadyForUpdate() {
        return commodities.size() > 0
                && commodities.GetSource().equals("local");
    }
    private void clearDisplay() {
        listbox.clear();
    }

    private void updateCommodities() {
        for (Iterator it = commodities.iterator(); it.hasNext(); ) {
            Commodity current = (Commodity)it.next();)
            if (singleBrokerCommodity(commodity)) {
                displayCommodity(current.getView());
            }
        }
    }
```

```
    private boolean singleBrokerCommodity(Commodity commodity) {
        return commodity.isTwilight() && !commodity.match(broker);
    }

    private void displayCommodity(CommodityView view) {
        listbox.add(view);
    }

    ...
}
```

Откровенно говоря, код в методе `update` мало чем отличается структурно. В нем по-прежнему присутствует оператор `if` и все вложенные в него функции, но теперь они делегируются другим методам. Метод `update` служит в качестве скелета для извлеченного из него кода. А что можно сказать об именах? Похоже, что они выглядят не очень убедительно, не так ли? Впрочем, они могут служить в качестве неплохой отправной точки. По крайней мере, они дают представление о коде на более высоком уровне и вводят швы для разрывания зависимостей. Теперь мы можем выполнить *подклассификацию и переопределение метода* для распознавания методов `displayCommodity` (отобразить товар) и `clearDisplay` (очистить отображение). После этого мы можем подумать о создании класса отображения и переносе в него этих методов, опираясь на тесты. Но в данном случае лучше было бы рассмотреть возможность переноса методов `update` и `updateCommodities` (обновить товары) в другой класс, оставив методы `displayCommodity` и `clearDisplay` в классе панели, чтобы воспользоваться тем обстоятельством, что данный класс относится к панели для отображения. А переименовать методы мы сможем в дальнейшем, когда они займут свои места. После дополнительной реорганизации кода его структура может выглядеть так, как показано на рис. 22.4.

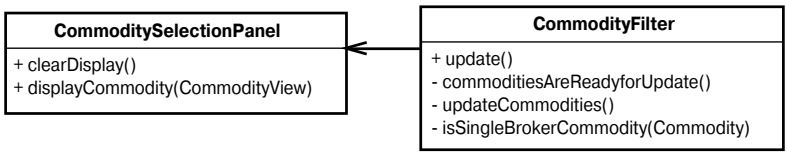


Рис. 22.4. Класс логики, извлеченной из класса `CommoditySelectionPanel`

Пользуясь инструментальным средством автоматической реорганизации кода, не забывайте о том, что большую часть черновой работы можно сделать безопасно, уделив внимание деталям после размещения тестов на местах. Не обращайте особого внимания на методы, которые, на первый взгляд, не подходят для класса. Они зачастую указывают на необходимость извлечь их впоследствии в новый класс. Подробнее о том, как это делается, см. в главе 20.

Трудности реорганизации кода вручную

Имея в своем распоряжении инструментальную поддержку автоматической реорганизации кода, вам не нужно делать ничего особенного, чтобы разделить на части крупный метод. Хорошие инструментальные средства реорганизации кода проверяют каждую реорганизацию кода, которую вы пытаетесь провести, и не допускают те ее разновидности, которые нельзя выполнить безопасно. Но если у вас нет подходящего инструментального

средства реорганизации кода, то, работая с крупными методами, вы должны соблюдать точность и аккуратность, пользуясь тестами как самым действенным средством.

Гигантские методы сильно затрудняют тестирование, реорганизацию кода и ввод новых свойств. Если удастся создать экземпляры класса, содержащего такой метод, в средствах тестирования, то можно попытаться придумать ряд контрольных примеров, позволяющих уверенно разделить данный метод на части. Если же логика гигантского метода оказывается слишком сложной, то работа с ним превращается в сущий кошмар. Правда, для выхода из подобных ситуаций имеются соответствующие приемы. Но прежде чем переходить к ним, рассмотрим те ошибки, которые мы можем совершить, извлекая методы.

Ниже перечислены лишь некоторые возможные ошибки, но они наиболее типичны.

1. Мы можем забыть передать переменную извлеченному методу. Об отсутствии переменной обычно сообщает компилятор, если только у нее не такое же имя, как и у переменной экземпляра, но мы можем просто посчитать, что переменная должна быть локальной, и объявить ее в новом методе.
2. Мы можем присвоить извлеченному методу имя, которое скрывает или переопределяет метод с таким же именем в базовом классе.
3. Мы можем сделать ошибку, передавая параметры или присваивая возвращаемые значения. Такая ошибка может оказаться довольно нелепой, как, например, возврат неверного значения. Менее заметная ошибка возникает при возврате или восприятии неверных типов данных в новом методе.

Подобного рода ошибки могут возникнуть и по ряду других причин. Способы, рассматриваемые в этом разделе, делают извлечение методов менее рискованным занятием, когда на местах отсутствуют подходящие тесты.

Внедрение переменной распознавания

Реорганизация кода не всегда предполагает ввод новых свойств, но это совсем не означает, что мы не можем добавлять код. Иногда ввод переменной в класс помогает распознать в методе условия, которые требуется реорганизовать. А по завершении необходимой реорганизации кода от такой переменной можно избавиться, чтобы сделать код более ясным. Такой прием называется *внедрением переменной распознавания*. Обратимся к конкретному примеру. Рассмотрим метод из класса Java под названием `DOMBuilder` (построитель домена). Нам требуется поправить этот метод, но, к сожалению, у нас нет для этого подходящего инструментального средства реорганизации кода.

```
public class DOMBuilder
{
    ...
    void processNode(XDOMNSnippet root, List childNodes)
    {
        if (root != null) {
            if (childNodes != null)
                    root.addNode(new XDOMNSnippet(childNodes));
                root.addChild(XDOMNSnippet.NullSnippet);
        }
        List paraList = new ArrayList();
        XDOMNSnippet snippet = new XDOMNReSnippet();
        snippet.setSource(m_state);
```

```
        for (Iterator it = childNodes.iterator();
                it.hasNext();) {
            XDOMNNode node = (XDOMNNode)it.next();
            if (node.type() == TF_G || node.type() == TF_H ||
                    (node.type() == TF_GLOT && node.isChild())) {
                paraList.addNode(node);
            }
            ...
        }
        ...
    }
...
}
```

По-видимому, в данном примере основные действия выполняются в методе `XDOMNSnippet` (фрагмент домена X). Это означает, что нам придется написать такие тесты, которые позволяют передавать разные значения данному методу. Но на самом деле бо́льшая часть действий происходит косвенным образом, и этому их можно распознать лишь очень косвенно. В подобной ситуации в качестве вспомогательного средства лучше внедрить переменные распознавания. В частности, мы можем внедрить переменную распознавания для того, чтобы выяснить, что узел вводится в список параметров (`paraList`), когда у него надлежащий тип узла.

```
public class DOMBuilder
{
   public boolean nodeAdded = false;
    ...
    void processNode(XDOMNSnippet root, List childNodes)
    {
        if (root != null) {
            if (childNodes != null)
                    root.addNode(new XDOMNSnippet(childNodes));
                root.addChild(XDOMNSnippet.NullSnippet);
        }
        List paraList = new ArrayList();
        XDOMNSnippet snippet = new XDOMNReSnippet();
        snippet.setSource(m_state);
        for (Iterator it = childNodes.iterator();
                it.hasNext(); ) {
            XDOMNNode node = (XDOMNNode)it.next();
            if (node.type() == TF_G || node.type() == TF_H ||
                    (node.type() == TF_GLOT && node.isChild())) {
                paraList.add(node);
                nodeAdded = true;
            }
            ...
        }
        ...
    }
    ...
}
```

Но даже при наличии в коде такой переменной нам все равно придется придумать способ передачи входных данных, чтобы получить контрольный пример, покрывающий упомянутое выше условие. Сделав это, мы сможем извлечь данную часть логики, а наши тесты должны по-прежнему проходить.

Ниже приведен тест, показывающий, что узел вводится, когда у него тип узла `TF_G`:

```
void testAddNodeOnBasicChild()
{
    DOMBuilder builder = new DomBuilder();
    List children = new ArrayList();
    children.add(new XDOMNNode(XDOMNNode.TF_G));
    builder.processNode(new XDOMNSnippet(), children);

    assertTrue(builder.nodeAdded);
}
```

А приведенный ниже тест, показывает, что узел не должен быть введен, если у него неверный тип узла.

```
void testNoAddNodeOnNonBasicChild()
{
    DOMBuilder builder = new DomBuilder();
    List children = new ArrayList();
    children.add(new XDOMNNode(XDOMNNode.TF_A));
    builder.processNode(new XDOMNSnippet(), children);
    assertTrue(!builder.nodeAdded);
}
```

Разместив эти тесты на местах, мы должны почувствовать бо́льшую уверенность, извлекая тело условия, определяющего порядок ввода узлов. Теперь мы можем скопировать это условие полностью. Тест, который мы только что написали, показывает, что узел вводится при выполнении данного условия.

```
public class DOMBuilder
{
    void processNode(XDOMNSnippet root, List childNodes)
    {
        if (root != null) {
            if (childNodes != null)
                    root.addNode(new XDOMNSnippet(childNodes));
                root.addChild(XDOMNSnippet.NullSnippet);
        }
        List paraList = new ArrayList();
        XDOMNSnippet snippet = new XDOMNReSnippet();
        snippet.setSource(m_state);
        for (Iterator it = childNodes.iterator();
                it.hasNext();) {
            XDOMNNode node = (XDOMNNode)it.next();
            if (isBasicChild(node)) {
                paraList.addNode(node);
                nodeAdded = true;
            }
            ...
```

```
        }
        ...
    }
    private boolean isBasicChild(XDOMNNode node) {
        return node.type() == TF_G
            || node.type() == TF_H
            || node.type() == TF_GLOT && node.isChild());
    }
    ...
}
```

В дальнейшем мы можем удалить переменную распознавания.

В данном примере использована логическая переменная для того, чтобы просто выяснить, вводится ли узел и после извлечения условия. Это дает возможность уверенно извлечь все тело условия, не внося в код ошибки и не вынуждая тестировать всю логику данного условия. Подобные тесты позволяют быстро проверять условия, чтобы убедиться в том, что они по-прежнему составляют часть ветви программы и после их извлечения. Дополнительные рекомендации относительно организации тестирования при извлечении методов приведены в разделе "Нацеленное тестирование" главы 13.

Используемые *переменные распознавания* целесообразно хранить в отдельном классе в течение целого ряда реорганизаций кода, а удалять их следует только по завершении всех операций, связанных с реорганизацией кода. Я обычно поступаю именно так, чтобы лучше видеть все тесты, написанные мной для извлечения, и без труда отменить их, если я найду возможность для извлечения методов другим способом. По завершении я удаляю эти тесты или же реорганизую их так, чтобы они проверяли извлеченные мной методы, а не исходный метод.

Переменные распознавания служат отличным средством для разделения гигантских методов на части. С их помощью можно не только реорганизовывать код глубоко внутри запутанных методов, но и постепенно сделать подобные методы менее запутанными. Так, если имеется метод, большая часть кода которого глубоко вложена в ряд условных операторов, то с помощью переменных распознавания можно извлечь условные операторы верхнего уровня или же тела этих условных операторов в новые методы. Переменные распознавания могут быть также использованы как для дальнейшей работы с вновь извлеченными методами, так и для окончательного распутывания кода.

Извлечение только того, что хорошо известно

Еще один стратегический прием, которым можно воспользоваться, работая с гигантскими методами, состоит в том, чтобы начинать с малого, находя небольшие фрагменты кода, которые можно уверенно извлечь без тестирования, а затем покрывать их дополнительными тестами. Здесь необходимо прояснить понятие "небольшой" фрагмент кода, поскольку каждый понимает его по-своему. В контексте рассматриваемого здесь способа под "небольшим" фрагментом кода подразумеваются две, три или самое большее пять строк кода, которым можно без особого труда присвоить имя. Извлекая столь небольшие фрагменты кода, следует уделять особое внимание *связующему числу*, которое обозначает число значений, передаваемых и возвращаемых извлекаемым методом. Так, если извлечь метод `max` из приведенного ниже метода, то его связующее число будет равно `3`.

```
void process(int a, int b, int c) {
    int maximum;
```

```
    if (a > b)
        maximum = a;
    else
        maximum = b;
    ...
}
```

Ниже приведен код после извлечения.

```
void process(int a, int b, int c) {
    int maximum = max(a,b);
    ...
}
```

Связующее число 3 извлеченного метода обозначает следующее: две входящие переменные и одну исходящую переменную. Извлекая методы, рекомендуется добиваться как можно меньшего связующего числа, поскольку в этом случае совершить ошибку труднее. Выбирая код для извлечения, старайтесь находить небольшое число строк, подсчитывая число входящих и исходящих переменных. В это число не входят переменные экземпляров, поскольку они не передаются извлекаемому методу через интерфейс.

Главная опасность при извлечении метода состоит в ошибке преобразования типов, когда, например, вместо значения `double` передается значение `int`. Вероятность такой ошибки окажется меньше, если извлекать только те методы, у которых малое связующее число. Определив метод кода, который можно извлечь, мы должны оглянуться назад и найти места, где объявляется каждая переменная, передаваемая извлекаемому методу, чтобы убедиться в правильности получаемой сигнатуры данного метода.

Если извлечения с малым связующим числом оказываются более безопасными, то еще безопаснее должны быть извлечения с нулевым связующим числом. Мы можем продвинуться далеко вперед, извлекая из гигантского метода только такие методы, которые не воспринимают и не передают никаких значений. Подобные методы на самом деле представляют собой команды, выполняющие определенные действия, например предписать объекту как-то изменить свое состояние, а возможно, и глобальное состояние. Тем не менее при попытке присвоить имя таким фрагментам кода мы начинаем лучше представлять себе их назначение и предполагаемое воздействие на объект. Такое представление постепенно составляется в более глубокое представление, позволяющее взглянуть на структуру кода с иной, более продуктивной точки зрения.

Извлекая только то, что хорошо известно, следите за тем, чтобы извлекаемые фрагменты кода не были слишком крупными. А если их связующее число окажется больше 0, зачастую имеет смысл воспользоваться переменными распознавания. После извлечения метода напишите для его проверки ряд тестов.

Применяя такой прием к небольшим фрагментам кода, нелегко заметить прогресс в улучшении гигантского метода, но этот прогресс намечается исподволь. Всякий раз, когда вы возвращаетесь к гигантскому методу для извлечения из него очередного небольшого фрагмента кода, вы знаете, что тем самым делаете этот метод чуть более ясным и понятным. Постепенно вы начинаете лучше представлять себе границы области действия данного метода и направления, в которых вам нужно двигаться дальше.

Когда в моем распоряжении нет подходящего инструментального средства реорганизации кода, я обычно начинаю извлекать методы с нулевым связующим числом только для того, чтобы получить ясное представление обо всей структуре кода. И зачастую это служит мне неплохим началом для тестирования и дальнейшей работы с кодом.

Если вам приходится иметь дело с методом без отступов, то вы можете посчитать, что способны извлечь немало методов с нулевым связующим числом и что для этой цели подходит едва ли не каждый фрагмент кода в методе без отступов. Иногда такие фрагменты кода действительно удается обнаружить, но чаще всего в них используются временные переменные, объявленные раньше этих фрагментов. Бывает и так, что "фрагментарную" структуру метода без отступов приходится отвергать и искать методы с малым связующим числом как внутри отдельных фрагментов кода, так и среди нескольких фрагментов.

Подбирание зависимостей

Иногда в гигантском методе обнаруживается код, выполняющий в нем второстепенные функции. Такой код, безусловно, нужен, но он оказывается ужасно сложным, и если случайно нарушить его, то такое нарушение сразу же станет очевидным. Но несмотря на все это, отказаться от идеи разделить на части основную логику гигантского метода не представляется возможным. В подобных случаях полезным оказывается способ, называемый *подбиранием зависимостей*. Его суть состоит в следующем: сначала вы пишете тесты для логики, которую вам нужно сохранить, а затем извлекаете логику, которую эти тесты не покрывают. Сделав это, вы, по крайней мере, будете уверены в том, что сохраняете важное поведение. Рассмотрим следующий простой пример.

```
void addEntry(Entry entry) {
    if (view != null && DISPLAY == true) {
        view.show(entry);
    }
    ...
    if (entry.category().equals("single")
            || entry.category("dual")) {
        entries.add(entry);
        view.showUpdate(entry, view.GREEN);
    }
    else {
        ...
    }
}
```

Если мы совершим ошибку в приведенном выше коде отображения, то быстро обнаружим ее. Но ошибку в логике метода `add` не так-то просто обнаружить. В подобных случаях мы можем написать тесты для метода `add` и проверить, проводится ли сложение при правильных условиях. А когда мы будем уверены в том, что такое поведение полностью покрывается тестами, то сможем извлечь код отображения, хорошо зная, что его извлечение не повлияет на сложение отображаемого элемента.

В известном смысле подбирание зависимостей является компромиссным решением. Вы сохраняете один ряд видов поведения и работаете с другим безо всяких предохранительных средств. Но в приложении не все виды поведения равнозначны. Одни из них важнее, чем другие, что и выясняется во время работы с кодом.

Подбирание зависимостей особенно полезно в тех случаях, когда один вид важного поведения тесно переплетается с другим. Имея в своем распоряжении надежные тесты для проверки важного поведения, вы можете смело править код, который эти тесты не покрывают, но при этом вы сохраняете основное поведение.

Вынос объекта метода

Переменные распознавания являются очень полезным средством в арсенале разработчика программного обеспечения, но иногда в самом коде можно обнаружить переменные, идеально подходящие для распознавания, хотя они и локальны для изменяемого метода. Если бы они были переменными экземпляров, то после выполнения метода они могли бы служить для распознавания. Локальные переменные можно превратить в переменные экземпляров, но, как правило, это приводит к путанице. Устанавливаемое состояние может оказаться общим только для гигантского метода и тех методов, которые из него извлекаются. И несмотря на то, что оно переустанавливается всякий раз, когда вызывается гигантский метод, очень трудно понять, в каких именно переменных оно сохраняется, если требуется вызвать методы, извлеченные независимо один от другого.

В качестве альтернативного подхода может служить *вынос объекта метода*, впервые предложенный Уардом Каннингхэмом и воплощающий в себе принцип вымышленной абстракции. Вынося объект метода, вы создаете класс, единственной ответственностью которого является выполнение функций гигантского метода. Параметры такого метода становятся параметрами конструктора нового класса, а код гигантского метода можно перенести в метод под названием `run` или `execute` из нового класса. После переноса кода в новый класс появляется отличная возможность для реорганизации кода. В таком методе временные переменные могут быть преобразованы в переменные экземпляров для распознавания при разделении метода на части.

Вынос объекта метода — довольно радикальная мера, но, в отличие от внедрения переменной распознавания, переменные, используемые для этой цели, выбираются из выходного кода. Это позволяет создавать тесты, которые можно сохранять. Конкретный пример применения *выноса объекта метода* приведен на с. 9.

Стратегия обращения с гигантскими методами

Способы и приемы, описанные в этой главе, помогают разбивать на части гигантские методы для дополнительной реорганизации кода и ввода новых переменных. В этом разделе приведены некоторые рекомендации относительно выбора компромиссных вариантов структуры кода при выполнении подобной работы.

Скелетное представление методов

Если метод содержит условный оператор, то имеются два варианта выбора — извлечь условный оператор вместе с телом метода или же извлечь их по отдельности. Рассмотрим следующий пример.

```
if (marginalRate() > 2 && order.hasLimit()) {
    order.readjust(rateCalculator.rateForToday());
    order.recalculate();
}
```

Если извлечь условный оператор и тело метода в два разных метода, то появится больше возможностей для последующей реорганизации логики данного метода.

```
if (orderNeedsRecalculation(order)) {
    recalculateOrder(order, rateCalculator);
}
```

Такой прием называется *скелетным представлением*, потому что от метода остается только скелет, т.е. управляющая структура и делегирование полномочий другим методам.

Поиск последовательностей

Если метод содержит условный оператор, то имеются две возможности — извлечь условный оператор вместе с телом метода или же извлечь их по отдельности. Рассмотрим еще один пример.

```
...
if (marginalRate() > 2 && order.hasLimit()) {
    order.readjust(rateCalculator.rateForToday());
    order.recalculate();
}
...
```

Если извлечь условный оператор и тело метода в один и тот же метод, то появится больше возможностей для выявления общих последовательностей операций.

```
...
recalculateOrder(order, rateCalculator);
...

void recalculateOrder(Order order,
                      RateCalculator rateCalculator) {
    if (marginalRate() > 2 && order.hasLimit()) {
        order.readjust(rateCalculator.rateForToday());
        order.recalculate();
    }
}
```

Остальная часть метода может оказаться не более чем последовательностью операций, следующих одна за другой, и поэтому положение прояснится, если удастся выявить такую последовательность.

Не противоречит ли эта рекомендация предыдущей? В какой-то степени противоречит. Но на самом деле во время работы с гигантскими методами нередко приходится обращаться то к *скелетному представлению методов*, то к *поиску последовательностей*. Я, например, прибегаю к скелетному представлению методов, когда чувствую, что управляющую структуру придется реорганизовать после ее прояснения. А находить последовательности я пытаюсь, когда чувствую, что выявление перекрывающихся последовательностей позволит прояснить код.

Работая с методами без отступов, я склонен находить последовательности, а имея дело с запутанными методами, я отдаю предпочтение скелетному представлению. Но выбор конкретной стратегии на самом деле зависит от того, насколько хорошо вы представляете себе структуру кода, приступая к извлечению методов.

Извлечение в текущий класс

Иногда, приступая к извлечению методов из гигантского метода, можно заметить, что некоторые фрагменты извлекаемого кода на самом деле принадлежат другим классам. На это, в частности, может указывать имя, выбираемое для извлекаемого фрагмента кода. Так, если возникает искушение воспользоваться именем одной из переменных, применяемых в извлекаемом фрагменте кода, то, скорее всего, этот код принадлежит тому же классу, что и сама переменная. Характерным тому примером служит приведенный ниже фрагмент кода.

```
if (marginalRate() > 2 && order.hasLimit()) {
    order.readjust(rateCalculator.rateForToday());
    order.recalculate();
}
```

Этому фрагменту кода можно было бы вполне присвоить имя `recalculateOrder` (пересчитать заказ). Но если в этом имени используется слово `Order`, то не лучше ли перенести данный фрагмент кода в класс `Order` и назвать его `recalculate`. Если же в классе `Order` уже имеется метод с таким именем, то данный фрагмент кода, скорее всего, придется переименовать, чтобы как-то различать разные виды перерасчета. Так или иначе, но рассматриваемый здесь фрагмент, по-видимому, принадлежит классу `Order`.

Невзирая на искушение извлечь код непосредственно в другой класс, делать этого не рекомендуется. Несмотря на то, что имя `recalculateOrder` выбрано неудачно, оно дает возможность выполнить ряд легко отменяемых извлечений, выяснить, тот ли код был извлечен, а затем двигаться дальше. Ведь мы можем всегда перенести извлеченный метод в другой класс, если впоследствии обнаружим более подходящее направление для внесения в код изменений. А между тем извлечение в текущий класс дает нам возможность для дальнейшего продвижения вперед по пути, который меньше чреват ошибками.

Извлечение небольшими фрагментами

Как упоминалось ранее в этой главе, но следует подчеркнуть еще раз, код следует извлекать сначала небольшими фрагментами. Перед извлечением небольшого фрагмента кода из гигантского метода создается впечатление, будто оно мало что дает. Но после извлечения большего количества фрагментов кода формируется иное представление об исходном методе. В частности, можно обнаружить последовательность операций, которая раньше была незаметной, или же выявить более совершенный способ организации данного метода. Обнаруживая эти пути, можно выбирать их для продвижения вперед. Это намного лучшая стратегия, чем разделение метода на крупные фрагменты с самого начала. Ведь очень часто сделать это оказывается не так просто, как кажется на первый взгляд, да и не безопасно. При этом можно легко упустить детали, но именно в деталях кроется работоспособность кода.

Будьте готовы к повторному извлечению

Для разделения гигантского метода на части имеется столько же способов, сколько и для разрезания пирога. После ряда последовательных извлечений кода обычно обнаруживаются более совершенные способы для внедрения новых свойств. Иногда самым лучшим решением оказывается отмена того или иного извлечения и его повторное выполнение.

Но это совсем не означает, что предыдущие извлечения кода оказались напрасной тратой времени. Они все равно принесли нечто очень важное — ясное представление о старой структуре кода и лучших путях для продвижения вперед.

Глава 23

Как узнать, нарушают ли что-нибудь изменения в коде

Код — это своеобразный строительный материал. Многие материалы, из которых изготавливаются предметы, например металл, дерево или пластмасса, теряют со временем свои качества и разрушаются. Код ведет себя иначе. Если не трогать его, то он никогда не разрушится. Так, если не подвергать носитель информации воздействию космических лучей или механическому износу, то хранящаяся на нем информация может оказаться сбойной лишь в том случае, если кто-нибудь ее отредактирует. Если же многократно запускать механизм, сделанный из металла, то он в конце концов сломается. А если многократно запускать на выполнение код, то он будет так же многократно выполняться.

Эти свойства кода возлагают тяжкое бремя на разработчиков программного обеспечения. Мы сами являемся первоисточником сбоев в программном обеспечении, хотя это может легко сделать кто угодно. Насколько легко изменить код? Механически сделать это очень просто. Всякий может открыть текстовый редактор и набрать в нем все что угодно — от стихотворения до бессмысленного набора слов, причем некоторые тексты, набранные подобным образом, компилируются (с условиями состязаний по написанию запутанного кода на языке C можно ознакомиться по адресу `www.ioccc.org`). Но если серьезно, то нарушить нормальную работу программы оказывается поразительно легко. Приходилось ли вам вылавливать таинственные программные ошибки, причиной которых оказывался случайно набранный символ, например, когда на клавиатуру упала книжка, которую вы упустили из рук, пытаясь передать ее коллеге? В этом отношении код очень хрупкий материал.

В этой главе рассматриваются различные способы снижения риска нарушить код при его правке. Одни из этих способов механические, а другие — психологические. Тем не менее они требуют к себе особого внимания, особенно если приходится разрывать зависимости в унаследованном коде для размещения тестов по местам.

Сверхосторожная правка

Что же мы делаем, когда правим код? Какие цели мы при этом преследуем? Обычно эти цели благие. Нам требуется ввести новое свойство или же исправить программную ошибку. Знать эти цели, конечно, хорошо, но как воплотить их в жизнь?

Каждое нажатие клавиши на клавиатуре может привести к одному из двух — изменить поведение программы или же не изменить его. Так, если набрать комментарии к программе, то они не изменят ее поведение, а если набрать строковый литерал, то поведение, скорее всего, изменится, за исключением тех случаев, когда этот литерал оказывается в коде, который вообще не вызывается. Но если набранный строковый литерал используется в вызове метода, то он, безусловно, изменяет поведение. Таким образом, нажатие клавиши пробела при форматировании исходного кода может формально считаться реорганизацией кода, хотя и очень незначительной. Иногда набор самого кода может быть также отнесен

к реорганизации кода. В то же время изменение числового литерала в выражении, используемом в коде, не может считаться реорганизацией кода, поскольку это *функциональное изменение*, о чем не следует забывать, вводя исходный текст кода.

Суть программирования в том и состоит, чтобы точно знать, к чему приводит каждое нажатие клавиши. Это не означает, что мы должны быть всеведущими, но такое знание действительно помогает ясно осознавать, каким образом мы воздействуем на программу, набирая ее исходный текст, чтобы исключить программные ошибки. В этом отношении весьма эффективной оказывается *разработка посредством тестирования*. Если код удается ввести в средства тестирования и проверить его с помощью тестов менее чем за секунду, то тесты можно выполнять в любой удобный момент, чтобы сразу же узнать о воздействиях на код в связи с внесенными в него изменениями.

Можно предположить, что со временем будет создана интегрированная среда разработки, позволяющая указывать набор тестов, выполняемых при каждом нажатии клавиши. Это был бы невероятно эффективный способ получения ответной реакции на изменения в коде.

Когда-нибудь это неизбежно случится. Ведь современные интегрированные среды разработки уже теперь позволяют проверять синтаксис при каждом нажатии клавиши, изменяя цвет выделения кода, когда в нем обнаруживаются ошибки. Поэтому следующим этапом станет правка кода с автоматически запускаемым тестированием.

Тесты способствуют сверхосторожной правке кода, как, впрочем, и парное программирование. Но не кажется ли сверхосторожная правка чрезмерной? Конечно, всякое излишество чрезмерно. Но самое главное, что сверхосторожная правка не разочаровывает, поскольку она позволяет отрешиться от внешнего мира и полностью сосредоточиться на самом коде. Кроме того, сверхосторожная правка не так утомляет, как отсутствие всякой ответной реакции, когда возникает опасение, что код был нарушен совершенно неумышленно. И тогда приходится мучительно восстанавливать в памяти последовательность внесенных изменений и думать о том, как затем убедить себя, что все было сделано так, как надо.

Правка с единственной целью

Я вполне допускаю, что первый опыт программирования у каждого оказывается разным, но когда я стал заниматься программированием, я был увлечен легендами об особо одаренных программистах, которые могли держать в голове состояние всей системы, писать на ходу правильный код и сразу же знать, насколько правильным окажется изменение кода. Очевидно, что у людей разные способности удерживать в памяти большие фрагменты информации и мелкие детали. Я, например, знаю многие малоизвестные свойства языка программирования C++ и даже могу вспомнить в нужный момент все подробности метамодели UML, но в конечном итоге я пришел к выводу, что знать в мельчайших деталях метамодель UML для программирования совсем не обязательно и просто бесполезно.

Одаренность может проявляться по-разному. Возможно, держать в уме состояние всей системы и полезно, но этого явно недостаточно для принятия оптимального решения. Оценивая свои профессиональные навыки, я считаю, что в настоящий момент я умею программировать намного лучше, чем прежде, хотя и знаю каждый язык, с которым работаю, не во всех деталях. Главным навыком для программирования, на мой взгляд, является рас-

судительность, а всякие попытки работать, как особо одаренные программисты, ни к чему хорошему не приводят.

Возможно, вам приходилось испытывать нечто подобное: приступив к работе над каким-нибудь свойством системы, вы затем решаете немного поправить и реорганизовать код, то тут же у вас возникает мысль, что код мог бы выглядеть иначе, и тогда вы останавливаетесь в нерешительности. Но свойство, над которым вы работаете, все равно нужно довести до логического завершения, и поэтому вы возвращаетесь к тому месту, где вы правили код. Вы решаете сделать сначала вызов метода, а затем переходите к самому методу, но тут же обнаруживаете, что метод должен выполнять еще кое-то что, и тогда вы начинаете изменять его, откладывая первоначальное изменение, на что ваш коллега, работающий с вами в паре, немедленно реагирует, восклицая: "Погоди! Покончи сначала с тем, а уж потом принимайся за это!" Вы чувствуете себя как беговая лошадь на дистанции, а ваш коллега вместо того, чтобы помогать вам, только подгоняет вас как жокей, а еще хуже — как азартный игрок на трибуне, сделавший на вас ставку.

Нечто подобное нередко случается в группах разработчиков. Пара разработчиков настолько увлекается программированием, что три последние четверти написанного ими кода требуют основательной правки первой четверти кода, который они нарушили. На первый взгляд, такая ситуация кажется ужасной, но иногда это становится даже интересно. Вы и ваш партнер испытываете охотничий азарт, затравив зверя в его логове. Но стоит ли овчинка выделки? Рассмотрим этот вопрос под несколько иным углом зрения.

Допустим, что вам нужно внести изменения в некоторый метод. У вас уже имеется класс в средствах тестирования, и поэтому вы можете приступить к внесению изменений в код. Но затем у вас возникает мысль о том, что нужно изменить и другой метод, и тогда вы переходите к нему. Его код требует упорядочения, и поэтому вы начинаете переформатирование одной или двух строк, чтобы сделать код яснее. Глядя на ваши действия, ваш коллега спрашивает: "Что ты делаешь?", на что вы отвечаете: "Я просто проверил, нужно ли нам изменять метод X". Тогда ваш коллега заявляет: "Послушай, давай делать все по порядку". Он записывает имя метода X на обрывке бумаги рядом с компьютерной клавиатурой, а вы возвращаетесь к правке кода, чтобы завершить ее. Выполнив тесты, вы обнаруживаете, что все они проходят. Затем вы переходите к другому методу. Анализируя его, вы убеждаетесь в необходимости внести в него изменения, и тогда вы приступаете к написанию очередного теста. Написав еще немного кода, вы вновь выполняете тесты и далее внедряете код. И тогда вы и ваш коллега слышат протестующее восклицание одного из двух других программистов, работающих за соседним столом: "Погодите! Исправьте сначала то, чтобы мы могли сделать это!" Они уже несколько часов работают над своей задачей, потратив на ее решение немало сил. Внесенные вами изменения могут помешать им интегрировать написанный ими код, и тогда их работа затянется еще на несколько часов.

Когда я работаю с кодом, то постоянно повторяю как молитву следующее: "Программирование — это искусство делать все по порядку". Работая в паре с коллегой, я всегда прошу его регулярно интересоваться тем, что я делаю. Если я отвечаю, что занимаюсь сразу несколькими делами, то мы выбираем что-то одно. Аналогичным образом я контролирую работу своего коллеги. Откровенно говоря, при такой организации труда работа над кодом продвигается быстрее. Программируя, можно очень легко потратить впустую немало времени, не добившись в итоге ничего, кроме отчаянных попыток сделать код работоспособным, вместо того чтобы действовать обдуманно, хорошо понимая, что именно делает код.

Сохранение сигнатур

Правя код, мы можем совершить ошибки самыми разными путями: неправильно набрать исходный текст, использовать неверный тип данных, ввести одну переменную, имея на самом деле в виду другую и т.д., и т.п. Особенно чревата ошибками реорганизация кода, поскольку она зачастую предполагает весьма радикальные изменения в коде. Мы копируем отдельные элементы кода из одного места в другое, создаем новые классы и методы, т.е. вносим намного более масштабные изменения, чем только ввод новой строки кода.

Для того чтобы сохранять ситуацию под контролем, как правило, пишутся тесты. Когда тесты находятся на местах, мы можем выловить многие ошибки, совершаемые нами при изменении кода. К сожалению, во многих системах нам приходится так или иначе реорганизовывать код лишь для того, чтобы сделать систему тестируемой в достаточной степени для последующей реорганизации кода. Такие первоначальные виды реорганизации кода (способы разрыва зависимостей, перечисляемые в главе 25) обычно не предполагают тестирование, и поэтому должны выполняться с особой осторожностью.

Когда я только начинал пользоваться подобными способами, у меня постоянно возникало искушение сделать сразу очень много. Когда мне требовалось извлечь все тело метода вместо того, чтобы просто скопировать и вставить аргументы при объявлении метода, то я выполнял и другую правку кода. Так, если мне нужно было извлечь тело метода и сделать его статическим, т.е. *раскрыть статический метод*, подобный следующему:

```
public void process(List orders,
                    int dailyTarget,
                    double interestRate,
                    int compensationPercent) {
    ...
    // далее следует сложный код
    ...
}
```

то я извлекал его приведенным ниже способом, создавая по ходу дела пару вспомогательных классов:

```
public void process(List orders,
                    int dailyTarget,
                    double interestRate,
                    int compensationPercent) {
    processOrders(new OrderBatch(orders),
                  new CompensationTarget(dailyTarget,
                  interestRate * 100,
                  compensationPercent));
}
```

Разумеется, все это я делал с самыми благими намерениями улучшить структуру кода, разрывая зависимости, но добиться нужного результата мне так и не удавалось. Вместо этого я совершал нелепые ошибки, которые не мог выловить сразу в отсутствие подходящих тестов, и поэтому я находил их лишь намного позже того момента, когда их действительно следовало отыскать.

Разрывая зависимости для тестирования кода, следует соблюдать особую осторожность. В частности, рекомендуется при всякой возможности *сохранять сигнатуры*. Копируя, вырезая и вставляя сигнатуры целых методов из одного места в другое, можно свести к минимуму вероятность появления любых ошибок.

В предыдущем примере полученный в итоге код выглядел бы следующим образом.

```
public void process(List orders,
                    int dailyTarget,
                    double interestRate,
                    int compensationPercent) {
    processOrders(orders, dailyTarget, interestRate,
                      compensationPercent);
}

private static void processOrders(List orders,
                          int dailyTarget,
                          double interestRate,
                          int compensationPercent) {
    ...
}
```

Правка аргументов, которую мне пришлось для этого выполнить, оказалась довольно простой. Она, по сути, заключалась в следующем.

1. Сначала я скопировал весь список аргументов в буфер копирования, вырезания и вставки.

   ```
   List orders,
   int dailyTarget,
   double interestRate,
   int compensationPercent
   ```

2. Затем я набрал объявление нового метода.

   ```
   private void processOrders() {
   }
   ```

3. Далее я вставил содержимое буфера в объявление нового метода.

   ```
   private void processOrders(List orders,
                              int dailyTarget,
                              double interestRate,
                              int compensationPercent) {
   }
   ```

4. Затем я набрал вызов нового метода.

   ```
   processOrders();
   ```

5. Далее я вставил содержимое буфера в вызов нового метода.

   ```
   processOrders(List orders,
                 int dailyTarget,
                 double interestRate,
                 int compensationPercent);
   ```

6. И наконец, я удалил типы данных, оставив только имена аргументов.

   ```
   processOrders(orders,
                 dailyTarget,
                 interestRate,
                 compensationPercent);
   ```

Когда вы выполняете подобные действия часто, они становятся автоматическими и придают вам бо́льшую уверенность в изменении кода. При этом вы можете сосредоточить основное внимание на ряде других неотложных вопросов, которые бывают причиной появления ошибок при разрыве зависимостей, например, на следующем вопросе: скрывает ли новый метод другой метод с такой же сигнатурой имени в базовом классе?

Сохранение сигнатур можно применять и в других случаях, в том числе при объявлении новых методов, а также при создании методов экземпляров для всех аргументов метода при реорганизации кода типа *выноса объекта метода*.

Упор на компилятор

Основное назначение компилятора — транслировать исходный код в другую форму, но в языках программирования со статическими типами данных компилятору можно найти и другое применение. Используя его свойство проверки типов данных, можно выявить изменения, которые необходимо внести в код. Такой способ называется *упором на компилятор*. Покажем, как это делается на примере программы C++, в которой имеются две глобальные переменные:

```
double domestic_exchange_rate;
double foreign_exchange_rate;
```

В файле исходного кода той же самой программы имеется ряд методов, в которых используются эти переменные, но нам нужно найти какой-то способ изменить их при тестировании, для чего вполне подходит *инкапсуляция глобальных ссылок*.

Для этого создается класс, где размещаются объявления упомянутых выше переменных и объявляется переменная данного класса.

```
class Exchange
{
public:
    double domestic_exchange_rate;
    double foreign_exchange_rate;
};

Exchange exchange;
```

Далее выполняется компиляция, чтобы выявить все места, где компилятор не сможет обнаружить переменные `domestic_exchange_rate` (курс местной валюты) и `foreign_exchange_rate` (курс иностранной валюты), после чего они изменяются вручную таким образом, чтобы обеспечить доступ к ним из объекта `exchange`. Ниже приведены фрагменты кода до и после подобных изменений, где переменная

```
total = domestic_exchange_rate * instrument_shares;
```

принимает такой вид:

```
total = exchange.domestic_exchange_rate * instrument_shares;
```

Главное достоинство такого способа состоит в том, что вы сами предоставляете компилятору возможность направить вас по пути изменений, которые требуется внести в код. Но это совсем не означает, что вам уже не нужно думать об этих изменениях. Просто в некоторых случаях вы можете частично переложить бремя своих забот на компилятор. Для

этого очень важно понимать, что именно компилятор способен обнаружить и на что он не способен, чтобы не доверяться ему слепо.

Упор на компилятор делается в два этапа.

1. Изменение в объявлении, чтобы вызвать ошибки компиляции.
2. Последовательное выявление этих ошибок в результатах компиляции и внесение изменений.

Упор на компилятор можно делать для внесения структурных изменений в программу, как в приведенном ранее примере *инкапсуляции глобальных ссылок*. Делая упор на компилятор, можно также инициировать изменения типов данных. Чаще всего изменение типа осуществляется в объявлении переменной, где ей присваивается тип интерфейса вместо класса. А с помощью обнаруженных ошибок определяются методы, которые должны присутствовать в таком интерфейсе.

Упор на компилятор не всегда оказывается практичным. Если для компоновки требуется немало времени, то, возможно, более практичным окажется поиск тех мест, где требуется внести изменения. Способы решения подобной проблемы представлены в главе 7. Но когда возможность сделать упор на компилятор все же появляется, этот способ действительно приносит практическую пользу. Впрочем, пользоваться им следует весьма осторожно и осмотрительно, чтобы не внести малозаметные программные ошибки.

Когда упор делается на компилятор, наибольшую возможность для выявления ошибок предоставляет такое свойство языка программирования, как наследование. Рассмотрим следующий пример. Допустим, что в классе Java имеется следующий метод `getX()`:

```
public int getX() {
    return x;
}
```

Нам требуется обнаружить все примеры применения данного метода, чтобы закомментировать их.

```
/*
public int getX() {
    return x;
} */
```

А теперь выполним компиляцию. Нетрудно догадаться, что компиляция не даст никаких ошибок. Означает ли это, что метод `getX()` не используется? Не обязательно. Если метод `getX()` объявляется в суперклассе как конкретный метод, то, закомментировав его в текущем классе, мы просто создадим условия для его применения в суперклассе. Аналогичная ситуация может возникнуть с переменными и наследованием.

Упор на компилятор является весьма эффективным способом, но в то же время следует хорошо осознавать его ограничения. В противном случае неизбежны серьезные ошибки.

Парное программирование

Вам, вероятно, уже приходилось слышать о *парном программировании*. Так, если вы руководствуетесь в своей практической деятельности принципами экстремального программирования, то, скорее, всего прибегаете и к парному программированию. Это замечательный способ повышения качества труда и распространения знаний и опыта в коллективе разработчиков.

Если же вы еще не пробовали заниматься парным программированием, то настоятельно рекомендуется сделать это. В частности, я настоятельно советую вам работать в паре с коллегой, применяя способы разрывания зависимостей, рассматриваемые в этой книге.

Работая в одиночку, очень легко совершить ошибку, даже не заметив нарушений в программе. И здесь очень пригодится пара внимательных глаз коллеги. Следует откровенно признать, что работа с унаследованным кодом подобна хирургическому вмешательству, а врачи, как известно, не оперируют без ассистентов.

Подробнее о парном программировании можно узнать из книги *Pair Programming Illuminated* (Парное программирование в правильном свете) Лори Уильямса (Laurie Williams) и Роберта Кесслера (Robert Kessler), Addison-Wesley, 2002.

Глава 24

Сдаемся — дальнейшее улучшение невозможно

Работать с унаследованным кодом нелегко. От него просто нельзя отказаться. Но, для того чтобы работа с ним имела какой-то смысл, следует, прежде всего, решить для себя, насколько она вам как программисту по плечу, хотя все, конечно, зависит от конкретной ситуации. Такую работу зачастую приходится выполнять, как говорится, ради куска хлеба. Но для занятия программированием должны и быть и другие побудительные причины.

Если вам повезло, то заниматься программированием вы начали просто ради интереса. Вам было любопытно сесть впервые за компьютер со всеми возможностями, которые в то время у него были, и попробовать написать какую-нибудь полезную программу. Это было интересно как с познавательной точки зрения, так и с точки зрения выбора будущей профессии, поскольку сулило весьма радужные перспективы.

Но не все приходят к программированию таким путем, и не у всех это связано с любопытством. Если вам и вашим коллегам действительно интересно заниматься программированием, то вам должно быть все равно, с какой системой работать, поскольку у вас есть возможность поэкспериментировать с ней. В противном случае вас ждет разочарование и такое ощущение, будто вы заслуживаете лучшей участи.

Нередко у тех, кому приходится работать с унаследованными системами, возникает желание работать с новыми системами, поскольку они считают, что проектировать систему с самого начала намного интереснее, но ведь и в проектировании новых систем имеются свои трудности. Мне не раз приходилось наблюдать за тем, как существующая система постепенно становится запущенной и трудно контролируемой. А те, кто ее поддерживают, просто приходят в отчаяние от того, как долго нужно вносить в нее изменения. Из лучших специалистов (а иногда виновников всех бед) составляют отдельную группу, призванную решать задачу создания системы, которая должна прийти на смену существующей, обладая более совершенной архитектурой. Поначалу все идет гладко. Хорошо зная недостатки старой архитектуры, разработчики из этой группы трудятся над построением новой, а тем временем остальные разработчики дорабатывают старую систему, которая находится в эксплуатации, и поэтому они получают периодически запросы на устранение программных ошибок, а иногда и на ввод новых свойств. Руководство трезво оценивает каждое новое свойство и решает, следует ли вводить его в старую систему или же клиент может подождать появления этого свойства в новой системе. Но зачастую клиенты не могут ждать, и поэтому изменения приходится вносить сразу в обе системы. На группу, разрабатывающую новую систему, возлагается двойная обязанность, поскольку ей придется заменять систему, которая постоянно изменяется. Проходят месяцы и становится ясно, что заменить систему, которая уже находится в эксплуатации, не удастся. Напряжение нарастает, разработчики трудятся сутками и без выходных, и все ждут, что их усилия окупятся впоследствии для всей организации сторицей.

Таким образом, работа над новой системой оказывается не такой уж и новаторской и интересной по сравнению с усовершенствованием старой системы. Для работы с унаследованным кодом очень важно найти мотивацию. Несмотря на то, что каждый програм-

мист — индивидуальность, вряд ли что-нибудь может заменить работу в благоприятной обстановке с коллегами, которых вы уважаете и которые знают, как можно сделать работу интересной. Лучших своих друзей я приобрел на работе и до сих пор общаюсь с ними, обсуждая свои новые или интересные открытия в программировании.

В работе помогает также общение с более широким кругом людей. В наше время проще, чем когда-либо прежде, связаться с другими программистами, чтобы учиться у них и делиться с ними секретами своего ремесла. Для этого достаточно подписаться на списки рассылки, посещать конференции, пользоваться всеми ресурсами в Интернете и быть постоянно в курсе самых последних новинок в области разработки программного обеспечения.

Глубокое уныние может воцариться в группе разработчиков даже в том случае, если все они ответственно подходят к своим обязанностям и стремятся делать работу как можно лучше. Иногда такое состояние возникает потому, что база кода оказывается настолько крупной, что разработчики могут работать с ней годами, но не сделать ее лучше и на 10%. Но является ли это поводом для уныния? Мне приходилось консультировать группы разработчиков, которые каждый рабочий день принимали как вызов и возможность хотя бы немного улучшить код и в то же время находили свою работу интересной. Но мне приходилось иметь дело и с такими группами, которые постигало уныние, хотя база кода у них была в намного лучшем состоянии. В конечном итоге все дело в том, как мы сами относимся к своей работе.

Попробуйте разработать какое-нибудь приложение с помощью тестирования вне работы. Это может быть какая угодно программа, написанная ради интереса. Постарайтесь почувствовать разницу в работе над мелкими и крупными проектами. Если вам удастся быстро ввести фрагменты кода в программные средства тестирования, то такую же возможность вы сможете найти и в том проекте, в котором вы заняты на работе.

Если в вашей группе царит состояние моральной подавленности и оно связано с низким качеством кода, попробуйте следующее: выберите самый неприятный ряд классов в проекте и подвергните их тестированию. Если вы и ваши коллеги вместе справитесь с самой трудной задачей, то сумеете изменить ситуацию к лучшему и улучшить моральную атмосферу в коллективе. Мне не раз приходилось наблюдать, как это происходит на практике.

Как только вы почувствуете, что база кода поддается вашему контролю, то сразу же начнете писать качественный код в большом количестве. И тогда работать станет веселее и интереснее.

Часть III

Методы разрыва зависимостей

Глава 25

Способы разрыва зависимостей

В этой главе представлен целый ряд способов разрыва зависимостей. И хотя этот перечень далеко не полный, приведенные в нем способы неоднократно опробованы мной в практике работы с группами разработчиков по разрыву зависимостей в классах, чтобы подвергнуть их тестированию. Формально все эти способы относятся к видам реорганизации кода, поскольку каждый из них предполагает сохранение поведения. Но в отличие от многих других видов реорганизации кода, практикуемых в настоящее время в программировании, рассматриваемые здесь виды реорганизации кода предназначены для выполнения без тестирования, чтобы разместить тесты по местам. Если, выполняя их, точно следовать процедуре, то вероятность совершения ошибки окажется очень малой. Но это не означает, что они совершенно безопасны. Вероятность совершить ошибку все же существует, поэтому пользоваться ими следует очень осторожно, обратившись за рекомендациями к материалу главы 23.

Рекомендации, приведенные в этой главе, помогут вам безопасно использовать рассматриваемые здесь способы для размещения тестов по местам. В этом случае вы сможете внести более радикальные изменения в код с большей уверенностью в том, что ничего в нем не нарушаете.

Рассматриваемые здесь способы не могут сразу сделать структуру кода лучше. В действительности, некоторые из них способны даже вызвать у вас сильные сомнения в их действенности, если у вас имеется хорошее представление о структуре кода. Тем не менее эти способы позволяют подготовить методы, классы и совокупности классов к тестированию, и благодаря этому система становится более удобной для сопровождения. А для того чтобы сделать структуру кода более ясной, вы можете далее воспользоваться видами реорганизации кода с поддержкой тестирования.

Некоторые виды реорганизации кода, представленные в этой главе, описаны в упоминавшейся ранее книге Мартина Фаулера, *Рефакторинг. Улучшение существующего кода*. В эту главу они включены с другой процедурой и приспособлены для безопасного применения без тестирования.

Адаптация параметра

Когда я вношу изменения в код, нередко сталкиваюсь с большими трудностями, связанными с параметрами методов. В одних случаях мне оказывается трудно создать требующийся параметр, а в других — мне нужно проверять воздействие метода на параметр. Но чаще всего трудности связаны с классом метода. Если класс оказывается одним из тех, что я видоизменяю, то я могу воспользоваться *извлечением интерфейса*, чтобы разорвать зависимость. Извлечение интерфейса зачастую оказывается наилучшим вариантом выбора, когда дело доходит до разрыва зависимостей.

Как правило, для разрыва зависимостей, препятствующих тестированию, нам требуется сделать что-нибудь простое, т.е. нечто такое, что сводит к минимуму вероятность появления ошибок. Но в некоторых случаях способ извлечения интерфейса действует не са-

мым лучшим образом. Так, если тип параметра оказывается довольно низкого уровня или же специфическим для применяемой технологии реализации, то извлечение интерфейса окажется неэффективным или просто невозможным.

Если извлечение интерфейса оказывается невозможным по отношению к классу параметра или же если подделать параметр нельзя, то воспользуйтесь *адаптацией параметра*.

Рассмотрим следующий пример.

```
public class ARMDispatcher
{
    public void populate(HttpServletRequest request) {
        String [] values
            = request.getParameterValues(pageStateName);
        if (values != null && values.length > 0)
        {
            marketBindings.put(pageStateName + getDateStamp(),
                               values[0]);
        }
        ...
    }
    ...
}
```

В этом классе метод `populate` (заполнить) воспринимает `HttpServletRequest` (запрос HTTP-сервлета) в качестве параметра. При этом `HttpServletRequest` является интерфейсом из стандартной платформы J2EE для Java. Если бы нам потребовалось протестировать метод `populate` в том виде, в каком он выглядит теперь, то нам пришлось бы создать класс, реализующий интерфейс `HttpServletRequest`, и предоставить какой-нибудь способ для его заполнения значениями параметров, которые требуется возвращать при тестировании. Как следует из текущей документации на набор инструментальных средств разработки Java SDK, в интерфейсе `HttpServletRequest` имеются объявления около 23 методов, не считая объявлений из его суперинтерфейса, которые нам пришлось бы реализовать. Для сужения интерфейса, предоставляющего только нужные нам методы, было бы неплохо воспользоваться извлечением интерфейса, но мы не можем извлечь один интерфейс из другого. В Java нам пришлось бы расширить интерфейс `HttpServletRequest` до извлекаемого интерфейса, но видоизменить стандартный интерфейс у нас нет никакой возможности. К счастью, у нас имеются другие возможности.

Для стандартной платформы J2EE доступно несколько библиотек имитирующих объектов. Если мы загрузим одну из них, то сможем использоваться имитацией интерфейса `HttpServletRequest` и выполнить требующееся нам тестирование. Благодаря этому мы сэкономим немало времени, поскольку, выбрав такой путь, мы не будем тратить зря время на формирование фиктивного запроса сервлета вручную. Итак, мы, по-видимому, нашли подходящее решение или еще не нашли?

Когда я разрываю зависимости, всегда стараюсь заглянуть немного вперед, чтобы посмотреть, к какому результату это приведет. Затем я выясняю, насколько меня устроят последствия такого шага. В данном случае выходной код не должен особенно измениться, и поэтому нам придется немало потрудиться, чтобы сохранить прикладной интерфейс `HttpServletRequest` на месте. Имеется ли другой способ улучшить код и упростить разрывание зависимости? На самом деле имеется. Мы можем заключить входящий пара-

метр в оболочку и полностью разорвать зависимость от данного прикладного интерфейса. Когда мы сделаем это, код будет выглядеть следующим образом:

```
public class ARMDispatcher
    public void populate(ParameterSource source) {
        String value = source.getParameterForName(pageStateName);
        if (value != null) {
            marketBindings.put(pageStateName + getDateStamp(),
                               value);
        }
        ...
    }
}
```

Что же мы сделали в этом коде? Мы ввели новый интерфейс под названием `ParameterSource` (Источник параметров). В данный момент у него имеется единственный метод под названием `getParameterForName` (получить параметр по имени). В отличие от метода `getParameterValues` (получить параметр по значениям) из интерфейса `HttpServletRequest`, метод `getParameterForName` возвращает только первый параметр. Мы написали метод именно таким образом, поскольку в данном контексте нас интересует только первый параметр.

> Пользуйтесь интерфейсами, которые передают виды ответственности, а не детали реализации. Благодаря этому код легче читать и проще сопровождать.

Ниже приведен фиктивный класс, реализующий интерфейс `ParameterSource`. Мы можем использовать его в нашем тесте.

```
class FakeParameterSource implements ParameterSource
{
    public String value;

    public String getParameterForName(String name) {
        return value;
    }
}
```

А в выходном коде источник параметров выглядит следующим образом.

```
class ServletParameterSource implements ParameterSource
{
    private HttpServletRequest request;

    public ServletParameterSource(HttpServletRequest request) {
        this.request = request;
    }

    String getParameterValue(String name) {
        String [] values = request.getParameterValues(name);
        if (values == null || values.length < 1)
            return null;
        return values[0];
    }
}
```

На первый взгляд, может показаться, что мы улучшаем код только ради самого улучшения, но для баз унаследованного кода весьма характерным оказывается отсутствие каких-либо уровней абстракции, когда самый важный в системе код зачастую перемежается с вызовами низкоуровневого интерфейса API. Мы уже знаем, до какой степени это может затруднить тестирование, но ведь дело не только в самом тестировании. Код труднее понимать, если он испещрен обширными интерфейсами с десятками неиспользуемых методов. А при наличии узких абстракций, нацеленных только на то, что требуется, код действует лучше, оставляя более отчетливый шов.

Если мы остановим свой выбор на использовании в данном примере интерфейса `ParameterSource`, то в конечном итоге развяжем логику заполнения и конкретные источники. Так образом, мы уже не будем связаны с конкретными интерфейсами платформы J2EE.

Адаптация параметра представляет собой один из тех случаев, когда мы не *сохраняем сигнатуры*, поэтому пользовать ею следует с особой осторожностью.

Адаптация параметра может оказаться рискованной, если упрощенный интерфейс, создаваемый для класса параметра, слишком отличается от текущего интерфейса данного параметра. Если мы не будем действовать аккуратно при внесение подобных изменений, то в конечном итоге можем внести в код едва заметные программные ошибки. Как всегда, не следует забывать о главной цели — разорвать зависимости настолько, чтобы разместить тесты по местам. Поэтому мы должны склоняться в сторону таких изменений, которые придадут нам больше уверенности, чем тех изменений, которые приведут к лучшей структуре кода. Такие изменения могут последовать после тестов. В данном примере, возможно, придется изменить интерфейс `ParameterSource`, чтобы клиентам не пришлось проверять пустое значение при вызове его методов (подробнее об этом см. *шаблон пустого объекта*).

Самое главное — это безопасность. Имея в своем распоряжении тесты, вы можете более уверенно вносить радикальные изменения в код.

Процедура

Для того чтобы воспользоваться адаптацией параметра, выполните следующую процедуру.

1. Создайте новый интерфейс, который вы будете использовать в методе. Сделайте его как можно более простым и коммуникативным, но постарайтесь не создавать интерфейс, который потребует не сложных изменений в этом методе, а самых простых.
2. Создайте средство реализации нового интерфейса в выходном коде.
3. Создайте фиктивное средство реализации данного интерфейса.
4. Составьте простой контрольный пример, передав фиктивный объект методу.
5. Внесите в метод требуемые изменения, чтобы использовать новый параметр.
6. Выполните тест, чтобы проверить возможность тестирования метода с помощью фиктивного объекта.

Вынос объекта метода

Во многих приложениях очень трудно работать с длинными методами. Если в средствах тестирования получается экземпляр класса, содержащий подобные методы, то зачастую можно сразу же приступать к написанию тестов. Но иногда для того, чтобы сделать класс отдельно доступным для получения экземпляра, приходится немало потрудиться, причем весь этот труд может оказаться неоправданным для изменений, которые требуется внести в код. Если же метод достаточно мал и в нем не используются данные экземпляра, то для тестирования внесенных в код изменений может вполне подойти *раскрытие статического метода*.

С другой стороны, если метод оказывается довольно крупным и в нем используются данные и методы экземпляров, то следует рассмотреть возможность для *выноса объекта метода*. В основу этого вида реорганизации кода положен перенос длинного метода в новый класс. Объекты, которые создаются с помощью этого нового класса, называются объектами метода, поскольку они воплощают в себе код единственного метода. После выноса объекта метода написать тесты для нового класса, как правило, оказывается легче, чем для старого метода. Локальные переменные в старом методе могут стать переменными экземпляров в новом классе. Благодаря этому нередко упрощается разрыв зависимостей и перевод кода в лучшее состояние.

Ниже приведен пример кода, написанного на языке C++ (крупные фрагменты класса были исключены из него ради сохранения древовидной структуры).

```
class GDIBrush
{
public:
    void draw(vector<point>& renderingRoots,
              ColorMatrix& colors,
              vector<point>& selection);
    ...

    private:
        void drawPoint(int x, int y, COLOR color);
        ...

    };

    void GDIBrush::draw(vector<point>& renderingRoots,
                        ColorMatrix& colors,
                        vector<point>& selection)
    {
        for(vector<points>::iterator it = renderingRoots.begin();
                it != renderingRoots.end();
                ++it) {
            point p = *it;

            ...
            drawPoint(p.x, p.y, colors[n]);
        }
        ...

}
```

В классе GDIBrush (кисть из интерфейса графических устройств) имеется длинный метод под названием draw (рисовать). Написать тесты для него не так-то просто, как, впрочем, и создать экземпляр класса GDIBrush в средствах тестирования. Поэтому воспользуемся выносом объекта метода, чтобы перенести метод draw в новый класс.

Прежде всего, следует создать новый класс, выполняющий функции рисования. Ему можно присвоить имя Renderer (визуализатор). После создания этот класс должен получить общедоступный конструктор. В качестве аргументов этого конструктора должна быть ссылка на исходный класс, а также аргументы исходного метода. Нам придется *сохранить сигнатуры* последнего.

```
class Renderer
{
public:
    Renderer(GBIBrush *brush,
             vector<point>& renderingRoots,
             ColorMatrix &colors,
             vector<point>& selection);
    ...
}
```

После создания конструктора следует ввести переменные экземпляров для каждого аргумента этого конструктора и затем инициализировать их. С этой целью воспользуемся рядом операций копирования, вырезания и вставки, чтобы сохранить также и сигнатуры.

```
class Renderer
{
private:
    GDIBrush *brush;
    vector<point>& renderingRoots;
    ColorMatrix& colors;
    vector<point>& selection;

public:
    Renderer(GDIBrush *brush,
            vector<point>& renderingRoots,
            ColorMatrix& colors,
            vector<point>& selection)
        : brush(brush), renderingRoots(renderingRoots),
          colors(colors), selection(selection)
        {}

};
```

Глядя на этот код, вы можете сказать: “Похоже, что мы этим ничего не добились. Несмотря на воспринимаемую ссылку на класс GDIBrush, мы по-прежнему не можем получить ни один из его экземпляров в средствах тестирования. Что же это нам дает?” Не спешите с выводами, поскольку дело еще не доведено до конца.

После создания конструктора мы можем ввести в класс еще один метод, выполняющий те же функции, что и метод draw(). Этому методу можно присвоить то же самое имя draw().

```
class Renderer
{
```

```
private:
    GDIBrush *brush;
    vector<point>& renderingRoots;
    ColorMatrix& colors;
    vector<point>& selection;

public:
    Renderer(GDIBrush *brush,
             vector<point>& renderingRoots,
             ColorMatrix& colors,
             vector<point>& selection)
        : brush(brush), renderingRoots(renderingRoots),
          colors(colors), selection(selection)
        {}

    void draw();
};
```

А теперь добавим тело метода `draw()` к классу `Renderer`. Для этого скопируем старый метод `draw()` в новый и сделаем *упор на компилятор*.

```
void Renderer::draw()
{
    for(vector<points>::iterator it = renderingRoots.begin();
            it != renderingRoots.end();
            ++it) {
        point p = *it;
        ...
        drawPoint(p.x, p.y, colors[n]);
    }
    ...

}
```

Если метод `draw()` из класса `Renderer` содержит какие-либо ссылки на переменные или методы экземпляров из класса `GDIBrush`, то компиляция окажется неудачной. Для того чтобы компиляция прошла удачно, мы можем создать методы получения (так называемые получатели) для переменных и сделать общедоступными методы, от которых зависит успех компиляции. В данном случае имеется единственная зависимость от частного метода `drawPoint` (рисовать точку). Сделав его общедоступным в классе `GDIBrush`, мы сумеем обращаться к нему по ссылке на класс `Renderer` и тем самым скомпилировать код.

Теперь мы можем передать полномочия метода `draw()` из класса `GDIBrush` новому методу из класса `Renderer`.

```
void GDIBrush::draw(vector<point>& renderingRoots,
                    ColorMatrix &colors,
                    vector<point>& selection)
{
    Renderer renderer(this, renderingRoots,
                      colors, selection);
    renderer.draw();
}
```

А теперь вернемся к зависимости от класса GDIBrush. Если нам не удается получить экземпляр класса GDIBrush в средствах тестирования, то мы можем воспользоваться *извлечением интерфейса*, чтобы полностью разорвать зависимость от класса GDIBrush. Это означает, что мы можем создать пустой интерфейсный класс и реализовать его в классе GDIBrush. В данном случае этому классу может быть присвоено имя PointRenderer (визуализатор точек), поскольку метод drawPoint, к которому нам нужно обращаться из класса Renderer, находится в классе GDIBrush. Далее заменим ссылку на класс GDIBrush, содержащуюся в классе Renderer, ссылкой на интерфейсный класс PointRenderer, выполним компиляцию и предоставим компилятору возможность указать нам на те методы, которые должны быть в интерфейсе. Ниже приведен код в его окончательном виде.

```
class PointRenderer
{
    public:
        virtual void drawPoint(int x, int y, COLOR color) = 0;
};

class GDIBrush : public PointRenderer
{
public:
    void drawPoint(int x, int y, COLOR color);
    ...
};

class Renderer
{
private:
    PointRender *pointRenderer;
    vector<point>& renderingRoots;
    ColorMatrix& colors;
    vector<point>& selection;

public:
    Renderer(PointRenderer *renderer,
             vector<point>& renderingRoots,
             ColorMatrix& colors,
             vector<point>& selection)
        : pointRenderer(pointRenderer),
          renderingRoots(renderingRoots),
          colors(colors), selection(selection)
        {}

    void draw();
};

void Renderer::draw()
{
    for(vector<points>::iterator it = renderingRoots.begin();
            it != renderingRoots.end();
            ++it) {
        point p = *it;
```

```
        ...
        pointRenderer->drawPoint(p.x,p.y,colors[n]);
    }
    ...

}
```

На рис. 25.1 показана структура этого кода в виде блок-схемы UML.

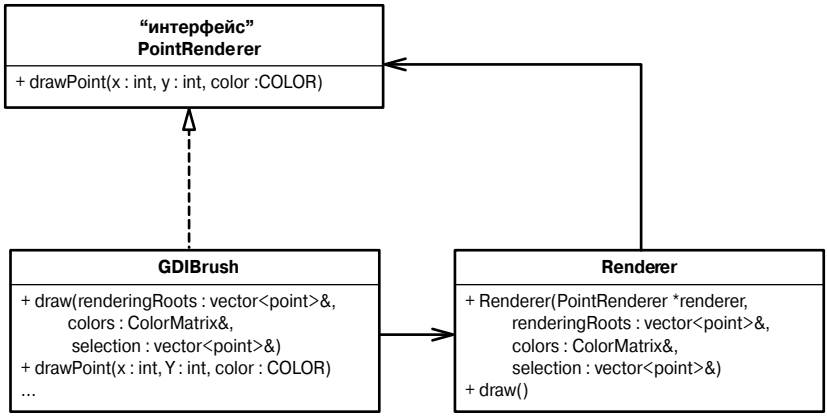


Рис. 25.1. Иерархия класса `GDIBrush` после выноса объекта метода

Полученный нами конечный результат выглядит несколько странно. У нас теперь имеется класс (`GDIBrush`), реализующий новый интерфейс (`PointRenderer`), единственное назначение которого состоит в том, что он используется объектом (`Renderer`), создаваемым данным классом. Возможно, вам станет немного не по себе, когда вы заметите, что мы сделали общедоступными детали, которые были частными в исходном классе, чтобы воспользоваться рассматриваемым здесь способом. Ведь метод `drawPoint`, который раньше был частным в классе `GDIBrush`, теперь открыт вовне. Но самое интересное, что это еще не все.

Со временем еще бо́льшую неприязнь у вас будет вызывать тот факт, что нельзя получить экземпляр исходного класса в средствах тестирования, а для этого вам придется разрывать зависимости. И тогда вам придется рассмотреть другие возможности. Например, должен ли класс `PointRenderer` быть непременно интерфейсом? Может ли он быть классом, содержащим класс `GDIBrush`? Если да, то, возможно, стоит перейти к структуре кода, основанной на этом новом понятии объектов `Renderer`.

Это лишь один из самых простых видов реорганизации кода, которую мы можем выполнить, подготавливая класс к тестированию. Получающаяся в итоге структура кода может желать много лучшего.

Имеется несколько разновидностей выноса объекта метода. В простейшем случае в исходном методе вообще не используются переменные или методы экземпляров из исходного класса, и поэтому нам не нужно передавать ему ссылку на исходный класс. В других случаях в исходном методе используются только данные из исходного класса. Иногда имеет смысл поместить эти данные в новый класс и передать его объекту метода в качестве аргумента.

В этом разделе приведен пример, демонстрирующий самый худший случай. Нам нужно использовать методы из исходного класса, и поэтому мы прибегаем к *извлечению интерфейса* и начинаем строить некоторую абстракцию между объектом метода и исходным классом.

Процедура

Для того чтобы безопасно вынести объект метода без тестирования, выполните следующую процедуру.

1. Создайте класс, который будет содержать код метода.
2. Создайте конструктор для данного класса и *сохраните сигнатуры*, чтобы предоставить ему точную копию аргументов, используемых методом. Если в методе используются данные экземпляров или же методы из исходного класса, то введите ссылку на исходный класс в качестве первого аргумента конструктора.
3. Объявите переменную экземпляра для каждого аргумента в конструкторе, присвоив тот же самый тип, что и у переменной. Сохраните сигнатуры, скопировав все аргументы непосредственно в класс и отформатировав их в виде объявлений переменных экземпляров. Присвойте все аргументы переменным экземпляров в конструкторе.
4. Создайте пустой исполняемый метод в новом классе. Зачастую такой метод называется `run()`. В приведенном выше примере он был назван `draw`.
5. Скопируйте тело старого метода в исполняемый метод и выполните компиляцию, сделав *упор на компилятор*.
6. Сообщения об ошибках, выдаваемые компилятором, должны указывать на те места в методе, где по-прежнему используются методы или переменные из старого класса. В каждом из этих случаев сделайте все, что требуется для компиляции метода. В одних случаях для этого достаточно изменить вызов, чтобы использовать ссылку на исходный класс, а в других — возможно, придется сделать методы общедоступными в исходном классе или же внедрить методы получения (получатели), чтобы не делать общедоступными переменные экземпляров.
7. После компиляции нового класса вернитесь к исходному методу и измените его таким образом, чтобы он создавал экземпляр нового класса и передавал ему свои полномочия.
8. Если требуется, воспользуйтесь *извлечением интерфейса*, чтобы разорвать зависимость от исходного класса.

Расширение определения

В некоторых языках программирования можно объявить тип данных в одном месте и определить его в другом. Наиболее очевидно такая возможность проявляется в языках C и C++, где функция или метод могут быть объявлены в каком-то другом месте (как правило, в файле реализации). Если есть такая возможность, то воспользуемся ею для разрыва зависимостей. Рассмотрим следующий пример.

```
class CLateBindingDispatchDriver : public CDispatchDriver
{
public:
                CLateBindingDispatchDriver ();
    virtual    ~CLateBindingDispatchDriver ();

    ROOTID      GetROOTID (int id) const;

    void        BindName (int id,
                          OLECHAR FAR *name);
    ...

private:
    CArray<ROOTID, ROOTID& > rootids;

};
```

В приведенном выше коде объявляется небольшой класс из приложения C++. Его пользователи сначала создают объекты класса `CLateBindingDispatchDriver` (драйвер диспетчеризации динамического связывания на C), а затем обращаются к методу `BindName`, чтобы связать имена с идентификационными номерами объектов. Нам требуется предоставить другой способ связывания имен, когда мы используем данный класс в тесте. В коде C++ мы можем воспользоваться для этой цели *расширением определения*. Метод `BindName` был объявлен в заголовочном файле класса. Как же нам дать ему другое определение для тестирования? Мы можем включить заголовок, содержащий определение данного класса, в тестовый файл и предоставить альтернативные определения для методов перед выполнением наших тестов.

```
#include "LateBindingDispatchDriver.h"

CLateBindingDispatchDriver::CLateBindingDispatchDriver() {}

CLateBindingDispatchDriver::~CLateBindingDispatchDriver() {}

ROOTID GetROOTID (int id) const { return ROOTID(-1); }

void BindName(int id, OLECHAR FAR *name) {}


TEST(AddOrder,BOMTreeCtrl)
{
    CLateBindingDispatchDriver driver;
    CBOMTreeCtrl ctrl(&driver);

    ctrl.AddOrder(COrderFactory::makeDefault());
    LONGS_EQUAL(1, ctrl.OrderCount());
}
```

Когда мы определяем эти методы непосредственно в тестовом файле, предоставляем определения, которые будут использоваться в тесте. Кроме того, мы можем предоставить пустые тела для методов, которые нас не интересуют, или же ввести методы распознавания, чтобы пользоваться ими во всех наших тестах.

Когда мы пользуемся расширением определения в коде C или C++, то просто вынуждены создавать отдельный исполняемый файл для тестов, в которых применяются расширенные определения. В противном случае они войдут в конфликт с настоящими определениями во время компоновки. Еще один недостаток данного способа состоит в том, что в этом случае у нас имеются два разных ряда определений методов класса: одно — в тестовом исходном файле, а другое — в готовом исходном файле. Это может существенно затруднить сопровождение приложения и отладку, если не настроить правильно соответствующую среду. По этим причинам расширением определения не рекомендуется пользоваться, кроме самых худших случаев зависимости. Но даже в этих случаях данный способ рекомендуется только для разрыва первоначальных зависимостей. Впоследствии, когда класс удается быстро протестировать, дублированные определения можно удалить.

Процедура

Для того чтобы воспользоваться расширением определения в коде C или C++, выполните следующую процедуру.

1. Выявите класс с определениями, которые требуется заменить.
2. Убедитесь в том, что определения методов находятся в исходном, а не в заголовочном файле. Если же они окажутся в заголовочном файле, то перенесите их в исходный файл.
3. Включите заголовочный файл в тестовый исходный файл того класса, который вы тестируете.
4. Убедитесь в том, что исходные файлы для данного класса не включены в компоновку.
5. Выполните компоновку, чтобы обнаружить недостающие методы.
6. Добавьте постепенно определения методов в тестовый исходный файл, чтобы завершить компоновку.

Инкапсуляция глобальных ссылок

Когда вы пытаетесь протестировать код со сложными зависимостями от глобальных объектов, то у вас, по существу, имеются три варианта выбора: попробовать изменить действие глобальных объектов при тестировании, установить связь с другими глобальными объектами или же инкапсулировать глобальные объекты для дополнительной развязки. Последний вариант называется *инкапсуляцией глобальных ссылок*. Рассмотрим следующий пример кода C++.

```
bool AGG230_activeframe[AGG230_SIZE];
bool AGG230_suspendedframe[AGG230_SIZE];

void AGGController::suspend_frame()
{
    frame_copy(AGG230_suspendedframe,
               AGG230_activeframe);
    clear(AGG230_activeframe);
    flush_frame_buffers();
}
```

```
void AGGController::flush_frame_buffers()
{
    for (int n = 0; n < AGG230_SIZE; ++n) {
        AGG230_activeframe[n] = false;
        AGG230_suspendedframe[n] = false;
    }
}
```

В этом примере представлен код, обрабатывающий ряд глобальных массивов. Методу `suspend_frame` требуется доступ к активным и приостановленным кадрам. На первый взгляд, кадры могут быть сделаны членами класса `AGGController`, но кадры используются и в ряде других классов, не показанных в приведенном выше коде. Что же тогда делать?

Первое, что приходит на ум, — это передать кадры методу `suspend_frame` в качестве параметров, используя *параметризацию метода*, но тогда нам придется передать их в качестве параметров и любым другим методам, которые вызываются методом `suspend_frame` и которым они требуются. В этом случае всему помехой служит метод `flush_frame_buffer` (очистить буфер кадров).

Следующий вариант состоит в том, чтобы передать оба типа кадров классу `AGGController` в виде аргументов его конструктора. Конечно, это можно сделать, но прежде следует обратить внимание на другие места, где эти типы кадров используются. Если окажется, что они используются совместно, то их можно связать вместе.

Если несколько глобальных объектов всегда используются или видоизменяются рядом, то они принадлежат одному классу.

Самым лучшим выходом из данного положения является анализ данных (т.е. активных и приостановленных кадров) и выбор подходящего имени для нового класса, который будет их содержать. Иногда это оказывается не так-то просто сделать. Ведь нужно осмыслить назначение данных в структуре кода, чтобы понять, для чего они нужны. Если мы создадим новый класс, то в конечном итоге переместим в него методы, но не исключено, что для этих методов где-то уже имеется код, в котором используются анализируемые нами данные.

Присваивая классу имя, подумайте о методах, которые в конечном итоге окажутся в нем. Такое имя должно быть вполне, хотя и не обязательно идеально, подходящим. Ведь класс можно всегда переименовать впоследствии.

Возвращаясь к анализу приведенного выше кода, можно предположить, что со временем методы `frame_copy` (копировать кадр) и `clear` (очистить) могут быть перенесены в новый класс, который мы собираемся создать. Имеется ли что-нибудь общее у активных и приостановленных кадров? По-видимому, имеется. Метод `suspend_frame` можно было бы перенести из класса `AGGController` в новый класс, если бы он содержал массивы `suspended_frame` и `active_frame`. Как бы мы назвали этот новый класс? Мы могли бы назвать его просто `Frame` (кадр) и объявить, что для каждого кадра имеется буфер активных кадров и буфер приостановленных кадров. Для этого нам пришлось бы немного изменить используемые понятия и переименовать переменные, а взамен мы получили бы более развитый класс, скрывающий детали.

Выбранное имя класса может уже использоваться. В таком случае следует рассмотреть возможность переименовать тот объект, в котором это имя уже используется.

Итак, реализуем наши замыслы по порядку. Прежде всего создадим новый класс, который будет выглядеть следующим образом.

```
class Frame
{
public:
    // объявить переменную AGG230_SIZE как постоянную
    // enum { AGG230_SIZE = 256 };

    bool AGG230_activeframe[AGG230_SIZE];
    bool AGG230_suspendedframe[AGG230_SIZE];
};
```

Мы намеренно оставили прежними имена данных, чтобы упростить дело на следующем этапе. А теперь объявим глобальным экземпляр класса `Frame`:

```
Frame frameForAGG230;
```

Далее закомментируем исходные объявления данных и попытаемся выполнить компоновку:

```
// bool AGG230_activeframe[AGG230_SIZE];
// bool AGG230_suspendedframe[AGG230_SIZE];
```

На данном этапе мы получим всевозможные типы ошибок компиляции, указывающие на то, что массивы `AGG230_activeframe` и `AGG230_suspendedframe` не существуют, и предупреждающие нас о неприятных последствиях. Если система компоновки поведет себя не очень дружелюбно, то выдаст около 10 страниц ошибок по поводу неразрешенных связей. Но огорчаться не стоит, ведь мы и не ожидали получить ничего другого, не так ли?

Для того чтобы избавиться ото всех этих ошибок, мы можем исправить каждую из них, поместив `frameForAGG230.` перед каждой ссылкой, вызывающей ошибку.

```
void AGGController::suspend_frame()
{
    frame_copy(frameForAGG230.AGG230_suspendedframe,
               frameForAGG230.AGG230_activeframe);
    clear(frameForAGG20.AGG230_activeframe);
    flush_frame_buffer();
}
```

Сделав это, мы получим еще более скверный код, тем не менее он будет компилироваться и правильно работать, а следовательно, данное преобразование выполнено с сохранением поведения. Покончив с этим, мы можем передать объект `Frame` через конструктор класса `AGGController` и тем самым добиться разделения, которое так необходимо нам для продвижения вперед.

Обращение по ссылке к члену класса вместо простой глобальной переменной — это лишь первый шаг. В дальнейшем нам придется рассмотреть возможность следующего выбора: *ввести статический установщик* или же параметризировать код, используя *параметризацию конструктора* либо *параметризацию метода*.

Итак, мы ввели новый класс, добавив в него глобальные переменные и сделав их общедоступными. Почему мы поступили именно так, а не иначе? Мы могли бы начать с создания фиктивного объекта `Frame`, передать его полномочия классу `AGGController`

и перенести в настоящий класс `Frame` всю логику, в которой используются эти глобальные переменные. Безусловно, мы могли бы все это сделать, но не сразу. Более того, без тестов на местах мы должны постараться как можно меньше трогать логику, постепенно продвигаясь к размещению тестов по местам. Мы должны не только не трогать логику, но и постараться отделить ее, введя швы, позволяющие вызывать один метод вместо другого или получать доступ к одному фрагменту данных, а не к другому. В дальнейшем, когда у нас появится больше тестов на местах, мы сумеем безболезненно перенести поведение из одного класса в другой.

Передав кадр классу `AGGController`, мы можем кое-что переименовать, чтобы немного прояснить ситуацию. Ниже приведен код, полученный в результате такой реорганизации.

```
class Frame
{
public:
    enum { BUFFER_SIZE = 256 };
    bool activebuffer[BUFFER_SIZE];
    bool suspendedbuffer[BUFFER_SIZE];
};

Frame frameForAGG230;

void AGGController::suspend_frame()
{
    frame_copy(frame.suspendedbuffer,
               frame.activebuffer);
    clear(frame.activeframe);
    flush_frame_buffer();
}
```

На первый взгляд, приведенный выше код не претерпел особых усовершенствований, но это был очень важный шаг. Ведь после переноса данных в класс мы добились нужного нам разделения и подготовили почву для улучшения кода со временем. В какой-то момент нам, возможно, понадобится даже класс `FrameBuffer` (буфер кадров).

> Используя инкапсуляцию глобальных ссылок, начинайте с данных и небольших методов. Более крупные методы можно перенести в новый класс, когда на местах окажется больше тестов.

В приведенном выше примере показано, как инкапсулируются глобальные ссылки по отношению к глобальным данным. То же самое можно сделать и с функциями, не являющимися членами в программах, написанных на C++. Когда приходится работать с интерфейсом API на языке C, нередко встречаются вызовы глобальных функций, рассеянные по всему коду, который требуется изменить. Единственным швом при этом оказывается связь вызовов с соответствующим функциями. В таком случае для разделения можно воспользоваться *подстановкой связи*, хотя для организации еще одного шва инкапсуляция глобальных ссылок дает лучше структурированный код.

Обратимся к еще одному примеру. Во фрагменте кода, который нам требуется протестировать, имеются вызовы двух функций — `GetOption(const string optionName)` и `setOption(string name, Option option)`. Это не более чем свободные функции,

не присоединяемые ни к одному классу, но они часто используются, например, в следующем коде.

```
void ColumnModel::update()
{
    alignRows();
    Option resizeWidth = ::GetOption("ResizeWidth");
    if (resizeWidth.isTrue()) {
        resize();
    }   else {
        resizeToDefault();
    }
}
```

В подобных случаях мы могли бы прибегнуть к таким проверенным способам, как *параметризация метода* и *извлечение и переопределение получателя*, но если вызовы функций распределены среди множества методов и классов, то проще воспользоваться инкапсуляцией глобальных ссылок. Для этого создадим следующий новый класс.

```
class OptionSource
{
public:
    virtual    ~OptionSource() = 0;
    virtual    Option GetOption(const string& optionName) = 0;
    virtual void    SetOption(const string& optionName,
                          const Option& newOption) = 0;
};
```

Этот класс содержит абстрактные методы для каждой из трех интересующих нас свободных функций. Далее выполним подклассификацию, чтобы сымитировать данный класс. В данном случае мы могли бы организовать карту или вектор в фиктивном подклассе, чтобы хранить в нем разные опции, которые мы использовали бы во время тестирования. В фиктивный подкласс мы могли бы ввести метод `add` или просто конструктор, воспринимающий карту, т.е. то, что удобнее всего для тестирования. Имея в своем распоряжении фиктивный подкласс, мы можем теперь создать источник для настоящих опций.

```
class ProductionOptionSource : public OptionSource
{
public:
    Option GetOption(const string& optionName);
    void SetOption(const string& optionName,
                   const Option& newOption) ;
};

Option ProductionOptionSource::GetOption(
       const string& optionName)
{
    ::GetOption(optionName);
}

void ProductionOptionSource::SetOption(
         const string& optionName,
         const Option& newOption)
```

```
{
    ::SetOption(optionName, newOption);
}
```

Для инкапсуляции ссылок на свободные функции создайте интерфейсный класс с фиктивными и выходными подклассами. Каждая из этих функций в выходном коде должна просто делегировать свои полномочия глобальной функции.

Такая реорганизация кода оказалась эффективной. Мы внедрили шов и в итоге делегировали простые полномочия функции из интерфейса API. Сделав это, мы можем теперь параметризировать класс, чтобы он воспринял объект класса `OptionSource` (источник опций). Благодаря этому мы можем использовать не фиктивный объект при тестировании, а настоящий объект в реальных условиях эксплуатации.

В приведенном выше примере мы поместили функции в класс и сделали их виртуальными. Можно ли было поступить по-другому? Да, можно. Мы могли бы создать свободные функции, делегирующие свои полномочия другим свободным функциям, или же ввести их в новый класс в качестве статических функций, но ни один из этих подходов не дал бы нам качественных швов. Поэтому нам пришлось бы воспользоваться *компоновочным швом* или же *швом предварительной обработки*, чтобы заменить одну реализацию другой. Когда же мы применяем подход, заключающийся в создании виртуальной функции и параметризации класса, швы получаются явными и легко управляемыми.

Процедура

Для инкапсуляции глобальных ссылок выполните следующую процедуру.

1. Выявите глобальные объекты, которые требуется инкапсулировать.
2. Создайте класс, в котором требуется обращаться к глобальным объектам по ссылке.
3. Скопируйте глобальные объекты в этот класс. Если некоторые из них окажутся переменными, организуйте их инициализацию в данном классе.
4. Закомментируйте исходные объявления глобальных объектов.
5. Объявите глобальный экземпляр нового класса.
6. Сделайте *упор на компилятор*, чтобы выявить все неразрешенные ссылки на старые глобальные объекты.
7. Предварите каждую неразрешенную ссылку именем глобального экземпляра нового класса.
8. В тех местах, где могут потребоваться фиктивные объекты, воспользуйтесь *вводом статического установщика*, *параметризацией конструктора*, *параметризацией метода* или же *заменой глобальной ссылки получателем*.

Раскрытие статического метода

Работать с классами, экземпляры которых нельзя получить в средствах тестирования, очень трудно. Поэтому здесь рассматривается способ, который упрощает работу лишь в некоторых случаях. Так, если имеется метод, в котором не используются данные или ме-

тоды экземпляров, то его можно сделать статическим, а затем протестировать, не прибегая к получению экземпляра класса. Рассмотрим пример кода, написанного на языке Java.

Допустим, что у нас имеется класс `validate` (проверить достоверность) и нам требуется добавить в него новое условие проверки достоверности. К сожалению, получить экземпляр класса, в котором находится этот метод, очень трудно. Избавив вас от необходимости анализировать весь класс в целом, приведем ниже только интересующий нас метод.

```
class RSCWorkflow
{
    ...
    public void validate(Packet packet)
            throws InvalidFlowException {
        if (packet.getOriginator().equals( "MIA")
                || packet.getLength() > MAX_LENGTH
                || !packet.hasValidCheckSum()) {
            throw new InvalidFlowException();
        }
        ...
    }
    ...
}
```

Что нужно сделать для того, чтобы протестировать этот метод? Если проанализировать данный метод внимательнее, то можно заметить, что в нем используется целый ряд других методов из класса `Packet` (пакет). В самом деле, было бы целесообразно перенести метод `validate` в класс `Packet`, но такой перенос представляется в настоящий момент едва ли не самым рискованным действием, поскольку мы определенно не сумеем *сохранить сигнатуры*. Без автоматической поддержки для переноса методов зачастую оказывается лучше разместить соответствующие тесты по местам. И в этом помощь может оказать *раскрытие статического метода*. Как только тесты окажутся на местах, мы сможем внести необходимые нам изменения в код и затем с намного большей уверенностью перенести метод.

> Когда вы разрываете зависимости без тестирования, *сохраняйте сигнатуры* методов при всякой возможности. В этом случае вы можете вырезать, копировать и вставлять целые сигнатуры методов с меньшим риском внести ошибки в код.

Код в данном примере не зависит от каких-либо переменных или методов экземпляров. А если бы метод `validate` оказался общедоступным и статическим? Приведенный ниже оператор можно было бы написать в любом месте кода и проверить пакет на достоверность.

```
RSCWorkflow.validate(packet);
```

Тот, кто создавал данный класс, даже и не думал о том, что кому-нибудь потребуется когда-нибудь сделать метод статическим и намного менее общедоступным. Насколько это плохо? Совсем не плохо. Инкапсуляция — это, конечно, очень хорошо для класса, но статическая область класса на самом деле не является частью класса. В некоторых языках программирования она может быть даже частью другого класса, иногда еще называемого метаклассом отдельного класса.

Когда метод статический, заранее известно, что у него нет доступа к любым частным данным класса. В этом случае он оказывается просто служебным методом. Если же сделать

метод общедоступным, то для него можно написать тесты. Эти тесты могут оказать необходимую поддержку при последующем переносе метода.

Статические методы на самом деле действуют так, как если бы они принадлежали другому классу. Статические данные существуют в течение всего срока действия программы, а не срока действия экземпляра, причем они доступны без экземпляра.

Статические части класса можно рассматривать в качестве своего рода "района сосредоточения" элементов кода, которые не полностью принадлежат классу. Если в коде обнаруживается метод, в котором вообще не используются данные экземпляров, то такой метод целесообразно сделать статическим и оставить его в этом хорошо заметном состоянии до тех пор, пока не будет окончательно решено, к какому классу он действительно принадлежит.

Ниже приведен код класса `RSCWorkflow` после извлечения в статический метод тела метода `validate`.

```
public class RSCWorkflow {
    public void validate(Packet packet)
           throws InvalidFlowException {
        validatePacket(packet);
    }

    public static void validatePacket(Packet packet)
           throws InvalidFlowException {
        if (packet.getOriginator() == "MIA"
                || packet.getLength() <= MAX_LENGTH
                || packet.hasValidCheckSum()) {
            throw new InvalidFlowException();
        }
        ...
    }
    ...
}
```

В некоторых языках программирования раскрытие статического метода можно осуществить более простым способом. Вместо извлечения статического метода из исходного можно просто сделать статическим исходный метод. Если же метод используется другими классами, то он по-прежнему остается доступным без экземпляра своего класса. Ниже приведен соответствующий пример.

```
RSCWorkflow workflow = new RCSWorkflow();
...
// вызов статического метода, похожий на вызов
// нестатического метода
workflow.validatePacket(packet);
```

Но в ряде других языков при компиляции такого кода будет выдано предупреждающее сообщение. Поэтому код лучше всего перевести в такое состояние, в котором подобные предупреждающие сообщения при компиляции не возникают.

Если вас беспокоит, что кто-нибудь может использовать статический метод ненадлежащим образом и тем самым вызвать впоследствии осложнения, связанные с зависимостью, то раскройте статический метод, используя необщий режим доступа. В таких языках, как

Java и C#, где поддерживается внутренняя область видимости или же область действия внутри пакета, доступ к статическому методу может быть ограничен или же его можно сделать защищенным и обращаться к нему с помощью *тестирующего подкласса*. В C++ имеются аналогичные возможности: статический метод можно сделать защищенным или же использовать пространство имен.

Процедура

Для раскрытия статического метода выполните следующую процедуру.

1. Напишите тест с доступом к методу, который требуется раскрыть в качестве статического общедоступного метода отдельного класса.
2. Извлеките тело метода в статический метод. Не забудьте *сохранить сигнатуры*. Данному методу придется присвоить другое имя. Для выбора имени нового метода воспользуйтесь именами параметров в качестве образца. Так, если метод под названием `validate` воспринимает объект `Packet`, то его тело можно извлечь в статический метод и присвоить последнему имя `validatePacket`.
3. Выполните компиляцию.
4. Если обнаружатся ошибки, связанные с доступом к данным или методам экземпляров, то обратите внимание на эти свойства и выясните, можно ли сделать их также статическими. Если можно, то сделайте их статическими, чтобы компиляция прошла успешно.

Извлечение и переопределение вызова

Иногда зависимости, препятствующие тестированию кода, оказываются достаточно локализованными. В коде может быть единственный вызов метода, который нам требуется заменить. И если нам удастся разорвать зависимость от вызова этого метода, то мы сможем исключить странные побочные эффекты тестирования или распознавания значений, передаваемых при вызове. Рассмотрим следующий пример.

```
public class PageLayout
{
    private int id = 0;
    private List styles;
    private StyleTemplate template;
    ...
    protected void rebindStyles() {
        styles = StyleMaster.formStyles(template, id);
        ...
    }
    ...
}
```

В классе `PageLayout` (компоновка страницы) делается вызов статического метода `formStyles` (стили форм) из класса `StyleMaster` (мастер стилей). Возвращаемое значение присваивается переменной экземпляра `styles`. Что же нам предпринять, если требуется распознать воздействия методом `formStyles` или же отделить зависимость от

класса `StyleMaster`? Для этого можно, в частности, извлечь вызов в новый метод и переопределить его в тестирующем подклассе. Такой способ называется *извлечением и переопределением вызова*. Ниже приведен код после извлечения.

```
public class PageLayout
{
    private int id = 0;
    private List styles;
    private StyleTemplate template;
    ...
    protected void rebindStyles() {
        styles = formStyles(template, id);
    ...
    }

    protected List formStyles(StyleTemplate template,
                              int id) {
        return StyleMaster.formStyles(template, id);
    }
    ...
}
```

Теперь, когда у нас имеется собственный локальный метод `formStyles`, мы можем переопределить его, чтобы разорвать зависимость. В данном примере нам не нужны стили для тех функций кода, которые мы тестируем в настоящий момент, и поэтому мы можем просто возвратить пустой список.

```
public class TestingPageLayout extends PageLayout {
    protected List formStyles(StyleTemplate template,
                              int id) {
       return new ArrayList();
    }
    ...
}
```

По мере разработки тестов нам потребуются разные стили, и поэтому, изменив данный метод, мы можем настроить его на конкретный возвращаемый результат.

Извлечение и переопределение вызова является очень полезным видом реорганизации кода. Я лично пользуюсь им очень часто. Это идеальный способ разрыва зависимостей от глобальных переменных и статических методов. Я обычно прибегаю к нему, если отсутствуют разные обращения к одной и той же глобальной переменной. Если же они присутствуют, то вместо данного способа я зачастую пользуюсь *заменой глобальной ссылки получателем* или же *параметризацией конструктора*.

Если в вашем распоряжении имеется инструментальное средство автоматической реорганизации кода, то извлечение и переопределение вызова не составит особого труда. Вы можете сделать это, используя *извлечение метода*. В противном случае воспользуйтесь приведенной ниже процедурой, с помощью которой можно безопасно извлечь любой вызов — даже если у вас отсутствуют тесты на местах.

Процедура

Для извлечения и переопределения вызова метода выполните следующую процедуру.

1. Выявите вызов, который требуется извлечь. Найдите объявление соответствующего метода. Скопируйте сигнатуру этого метода, чтобы *сохранить сигнатуры*.
2. Создайте новый метод в текущем классе. Присвойте ему скопированную сигнатуру.
3. Скопируйте вызов нового метода и замените им данный вызов.
4. Введите тестирующий подкласс и переопределите новый метод.

Извлечение и переопределение фабричного метода

Создание объектов в конструкторах может доставить немало хлопот, когда класс требуется подвергнуть тестированию. Иногда действия, происходящие в таких объектах, не должны совершаться в средствах тестирования. А в других случаях требуется просто разместить на месте объект распознавания, но сделать это не удается, поскольку создание такого объекта жестко закодировано в конструкторе.

Инициализацию, жестко закодированную в конструкторах, иногда очень трудно обойти при тестировании.

Рассмотрим следующий пример.

```
public class WorkflowEngine
{
    public WorkflowEngine () {
        Reader reader
            = new ModelReader(
                  AppConfig.getDryConfiguration());

        Persister persister
            = new XMLStore(
                AppConfiguration.getDryConfiguration());

        this.tm = new TransactionManager(reader, persister);
        ...
    }
    ...
}
```

В конструкторе класса `WorkflowEngine` (механизм последовательности операций) создается объект `TransactionManager` (администратор транзакций). Если бы этот объект создавался в каком-нибудь другом месте, то нам было бы легче осуществить разделение. Одна из имеющихся у нас возможностей состоит в *извлечении и переопределении фабричного метода*.

```
public class WorkflowEngine
{
```

```
    public WorkflowEngine () {
        this.tm = makeTransactionManager();
        ...
    }

    protected TransactionManager makeTransactionManager() {
        Reader reader
           = new ModelReader(
               AppConfiguration.getDryConfiguration());

        Persister persister
            = new XMLStore(
                AppConfiguration.getDryConfiguration());
        return new TransactionManager(reader, persister);
    }
    ...
}
```

Извлечение и переопределение фабричного метода является довольно эффективным средством, но оно очень зависит от используемого языка программирования. В частности, его нельзя осуществить в C++, поскольку в C++ не допускаются вызовы виртуальных функций для разрешения функций в производных классах. А в Java и многих других языках это допускается. В качестве неплохой альтернативы для C++ может стать замена переменной экземпляра, а также *извлечение и переопределение получателя*. Эта проблема обсуждается на примере *замены переменной экземпляра*.

Имея в своем распоряжении такой фабричный метод, мы можем выполнить подклассификацию и переопределить этот метод, чтобы возвратить новый объект администратора транзакций, когда он нам понадобится.

```
public class TestWorkflowEngine extends WorkflowEngine
{
    protected TransactionManager makeTransactionManager() {
        return new FakeTransactionManager();
    }
}
```

Процедура

Для извлечения и переопределения фабричного метода выполните следующую процедуру.

1. Выявите создание объекта в конструкторе.
2. Извлеките все, что относится к созданию объекта, в фабричный метод.
3. Создайте тестирующий подкласс и переопределите в нем фабричный метод, чтобы исключить зависимости от трудно тестируемых типов данных.

Извлечение и переопределение получателя

Извлечение и переопределение фабричного метода является весьма эффективным способом разделения зависимостей от типов данных, но он подходит не для всех случаев. Серьезным пробелом в его применимости является C++, где не допускается вызов виртуальной функции в производном классе из конструктора базового класса. Правда, из этого положения есть выход, если речь идет только о создании объекта в конструкторе.

Суть такой реорганизации кода заключается в том, чтобы ввести метод получения или так называемый получатель для переменной экземпляра, которую требуется заменить фиктивным объектом. Затем код реорганизуется для использования получателя в каждом месте отдельного класса. После этого можно выполнить подклассификацию и переопределить получатель, чтобы предоставить альтернативные объекты для тестирования.

В рассматриваемом здесь примере мы создаем администратор транзакций в конструкторе. Нам нужно организовать код так, чтобы этот администратор транзакций можно было использовать в классе при эксплуатации и распознавать его при тестировании. Итак, начнем со следующего кода.

```
// WorkflowEngine.h
class WorkflowEngine
{
private:
    TransactionManager *tm;
public:
    WorkflowEngine ();
    ...
}

// WorkflowEngine.cpp
WorkflowEngine::WorkflowEngine()
{
    Reader *reader
        = new ModelReader(
            AppConfig.getDryConfiguration());

    Persister *persister
        = new XMLStore(
            AppConfiguration.getDryConfiguration());

    tm = new TransactionManager(reader, persister);
    ...
}
```

А вот какой код получается в итоге.

```
// WorkflowEngine.h
class WorkflowEngine
{
private:
    TransactionManager *tm;

protected:
```

```
        TransactionManager *getTransaction() const;

public:
                            WorkflowEngine ();
    ...
}

// WorkflowEngine.cpp
WorkflowEngine::WorkflowEngine()
:tm (0)
{
    ...
}

TransactionManager *getTransactionManager() const
{
    if (tm == 0) {
        Reader *reader
            = new ModelReader(
                AppConfig.getDryConfiguration());

        Persister *persister
            = new XMLStore(
                AppConfiguration.getDryConfiguration());

        tm = new TransactionManager(reader,persister);
        }
        return tm;
}
...
```

Прежде всего следует ввести получатель по требованию, создающий объект администратора транзакций по первому же вызову, а затем заменить все примеры применения переменной вызовами этого получателя.

Получатель по требованию — это метод, который внешне выглядит как обычный получатель, но отличается тем, что создает объект по первому же вызову. Для этой цели получатель по требованию обычно содержит логику, аналогичную приведенной ниже. Обратите внимание на то, как инициализируется переменная `thing` (нечто).

```
Thing getThing() {
    if (thing == null) {
        thing = new Thing();
    }
    return thing;
}
```

Получатели по требованию используются также в *шаблоне проектирования одиночки*.

Имея в своем распоряжении такой получатель, мы можем выполнить подклассификацию и переопределение, чтобы внедрить еще один объект.

```
class TestWorkflowEngine : public WorkflowEngine
{
public:
    TransactionManager *getTransactionManager()
                        { return &transactionManager; }
    FakeTransactionManager transactionManager;
};
```

Извлекая и переопределяя получатель, следует иметь в виду срок действия объекта, особенно в таких языках программирования без поддержки "сборки мусора", как C++. Непременно организуйте удаление тестового экземпляра объекта согласованно с удалением аналогичного экземпляра в выходном коде.

В тесте мы можем, если требуется, получить без особого труда доступ к фиктивному администратору транзакций.

```
TEST(transactionCount, WorkflowEngine)
{
    auto_ptr<TestWorkflowEngine> engine(new TestWorkflowEngine);
    engine.run();
    LONGS_EQUAL(0,
        engine.transactionManager.getTransactionCount());
}
```

Недостаток извлечения и переопределения получателя заключается, в частности, в том, что существует вероятность того, что кто-нибудь воспользуется переменной до ее инициализации. По этой причине целесообразно организовать класс так, чтобы весь его код использовал получатель.

Извлечение и переопределение получателя — это способ, подходящий не для каждого случая. Так, если в объекте имеется лишь один проблематичный метод, намного проще воспользоваться *извлечением и переопределением вызова*. Тем не менее извлечение и переопределение получателя лучше выбирать при наличии многих проблематичных методов в одном и том же объекте. Если от всех этих проблем можно избавиться извлечением и переопределением получателя, то такой способ дает явное преимущество.

Процедура

Для извлечения и переопределения получателя выполните следующую процедуру.

1. Выявите объект, для которого требуется получатель.
2. Извлеките всю логику, необходимую для создания объекта в получателе.
3. Замените все примеры применения объекта вызовами получателя и инициализируйте ссылку на объект пустым значением во всех конструкторах.
4. Введите в получатель логику первого вызова, по которой объект создается и присваивается по ссылке всякий раз, когда она указывает на пустое значение.
5. Создайте подкласс и переопределите получатель, чтобы предоставить альтернативный объект для тестирования.

Извлечение средства реализации

Пользоваться *извлечением интерфейса*, конечно, удобно, но при этом возникают трудности с присвоением имен. Когда мне нужно извлечь интерфейс, нередко оказывается, что имя, которое я собираюсь использовать, уже присвоено какому-нибудь классу. Если я извлекаю интерфейс, работая в интегрированной среде разработки, поддерживающей переименование классов, то присвоение имен не доставляет мне никаких хлопот. В противном случае мне остается одно из двух:

- придумать какое-нибудь нелепое имя;
- проанализировать интересующие меня методы и выяснить, являются ли они подмножеством общедоступных методов в отдельном классе. Если они таковыми являются, то мне придется придумать другое имя для нового интерфейса.

Как правило, я стараюсь не составлять имя нового интерфейса из префикса *I* и имени соответствующего класса, если только такое условное обозначение уже не принято в базе кода. Ведь нет ничего хуже, чем работать с незнакомым фрагментом кода, в котором одна половина имен определенного типа начинается с префикса *I*, а другая — не начинается. При этом совершенно не ясно, пропущен ли префикс *I* по ошибке или же это сделано намеренно.

Присвоение имен — очень важная часть проектирования программного обеспечения. Выбирая удачные имена, вы способствуете лучшему пониманию проектируемой системы и упрощаете работу с ней. А если вы выбираете неудачные имена, то тем самым затрудняете понимание системы, усложняя ее последующее сопровождение.

Если имя класса идеально подходит для именования интерфейса и в моем распоряжении отсутствуют инструментальные средства автоматической реорганизации кода, то я пользуюсь *извлечением средства реализации*, чтобы добиться нужного мне разделения. Для того чтобы извлечь средство реализации класса, мы превращаем класс в интерфейс, выполняя его подклассификацию и перенося все его конкретные методы в подкласс. Рассмотрим следующий пример кода, написанного на C++.

```
// ModelNode.h
class ModelNode
{
private:
    list<ModelNode *>    m_interiorNodes;
    list<ModelNode *>    m_exteriorNodes;
    double               m_weight;
    void                 createSpanningLinks();

public:
    void addExteriorNode(ModelNode *newNode);
    void addInternalNode(ModelNode *newNode);
    void colorize();
    ...

};
```

Прежде всего следует полностью скопировать объявление класса ModelNode (узел модели) в другой заголовочный файл и изменить его имя на ProductionModelNode (выходной узел модели). Ниже приведена часть объявления для скопированного класса.

```
// ProductionModelNode.h
class ProductionModeNode
{
private:
    list<ModelNode *>    m_interiorNodes;
    list<ModelNode *>    m_exteriorNodes;
    double               m_weight;
    void                 createSpanningLinks();
public:
    void addExteriorNode(ModelNode *newNode);
    void addInternalNode(ModelNode *newNode);
    void colorize();
    ...
};
```

Далее следует вернуться к заголовку ModelNode и извлечь из него все объявления необщедоступных переменных и методов. После этого мы можем сделать все оставшиеся общедоступные методы чисто виртуальными, т.е. абстрактными.

```
// ModelNode.h
class ModelNode
{
public:
    virtual void addExteriorNode(ModelNode *newNode) = 0;
    virtual void addInternalNode(ModelNode *newNode) = 0;
    virtual void colorize() = 0;
    ...
};
```

На данном этапе ModelNode представляет собой чистый интерфейс. Он содержит только абстрактные методы. А поскольку мы имеем дело с кодом, написанным на C++, то должны также объявить чисто виртуальный деструктор и определить для него файл реализации.

```
// ModelNode.h
class ModelNode
{
public:
    virtual        ~ModelNode () = 0;
    virtual void    addExteriorNode(ModelNode *newNode) = 0;
    virtual void    addInternalNode(ModelNode *newNode) = 0;
    virtual void    colorize() = 0;
    ...
    };

// ModelNode.cpp
ModelNode::~ModelNode()
{}
```

Теперь мы можем вернуться к классу ProductionModelNode и сделать так, чтобы он наследовал новый интерфейсный класс.

```
#include "ModelNode.h"
class ProductionModelNode : public ModelNode
{
private:
    list<ModelNode *>    m_interiorNodes;
    list<ModelNode *>    m_exteriorNodes;
    double               m_weight;
    void                 createSpanningLinks();

public:
    void addExteriorNode(ModelNode *newNode);
    void addInternalNode(ModelNode *newNode);
    void colorize();
    ...

};
```

На данном этапе класс ProductionModelNode должен компилироваться без ошибок. Если скомпоновать остальную часть системы, то в ней, скорее всего, обнаружатся места, где кто-нибудь может попытаться получить экземпляры объектов ModelNode. В этих местах можно провести замену, чтобы вместо объектов ModelNode создавались объекты ProductionModelNode. В данном виде реорганизации кода мы заменяем создание объектов одного конкретного класса созданием объектов другого класса и пытаемся выяснить, можно ли воспользоваться фабрикой для дальнейшего сокращения зависимостей.

Процедура

Для извлечения средства реализации выполните следующую процедуру.

1. Сделайте копию объявления исходного класса. Присвойте новому классу другое имя. Для извлекаемых классов полезно выбрать подходящие условные обозначения. Я, например, часто добавляю к имени нового класса префикс Production, чтобы указать на то, что новый класс относится к выходному коду, где он реализует интерфейс.
2. Преобразуйте исходный класс в интерфейс, удалив все необщедоступные методы и переменные.
3. Сделайте абстрактными все оставшиеся общедоступные методы. Если вы работаете с кодом на языке C++, убедитесь в том, что ни один из методов, которые вы делаете абстрактными, не переопределяется невиртуальными методами.
4. Проверьте, все ли виды импорта и включения действительно необходимы в файле интерфейса. Зачастую многие из них можно удалить. Для их обнаружения вы можете сделать *упор на компилятор*. Удалите их по очереди и перекомпилируйте код, чтобы проверить, требуется ли дальнейшей удаление.
5. Сделайте реализацию выходного класса новым интерфейсом.
6. Скомпилируйте выходной класс, чтобы непременно реализовать сигнатуры всех методов в интерфейсе.

7. Скомпилируйте остальную часть системы, чтобы обнаружить все места, где создаются экземпляры исходного класса. Замените в этих местах создание экземпляров исходного класса на создание экземпляров выходного класса.
8. Повторите компиляцию и проведите тестирование кода.

Более сложный пример

Извлечение средства реализации осуществляется относительно просто, когда в иерархии наследования исходного класса отсутствуют родительские или порожденные классы. В противном случае нам приходится действовать более изощренно. На рис. 25.2 показана иерархия наследования класса `ModelNode`, но на этот раз она реализована на языке Java.

В данной иерархии классы `Node`, `ModelNode` и `LinkageNode` (связующий узел) являются конкретными. В классе `ModelNode` используются защищенные методы из класса `Node`. Кроме того, он предоставляет доступ к методам, используемым в его подклассе `LinkageNode`. Для извлечения средства реализации требуется конкретный класс, который можно преобразовать в интерфейс, что итоге дает интерфейс и конкретный класс.

В такой ситуации можно сделать следующее: извлечь средство реализации из класса `Node`, расположив класс `ProductionNode` ниже класса `Node` в иерархии наследования, а затем изменить отношение наследования таким образом, чтобы класс `ModelNode` наследовал класс `ProductionNode`, а не класс `Node`. На рис. 25.3 показано, как будет выглядеть иерархия наследования после извлечения средства реализации.

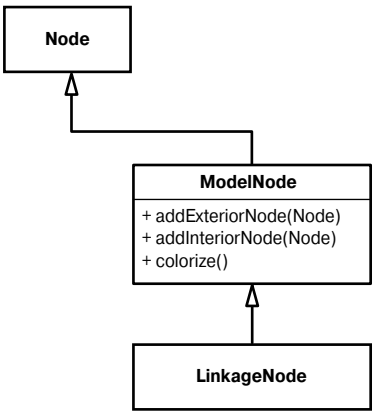


Рис. 25.2. Класс `ModelNode` вместе со своим суперклассом и подклассом

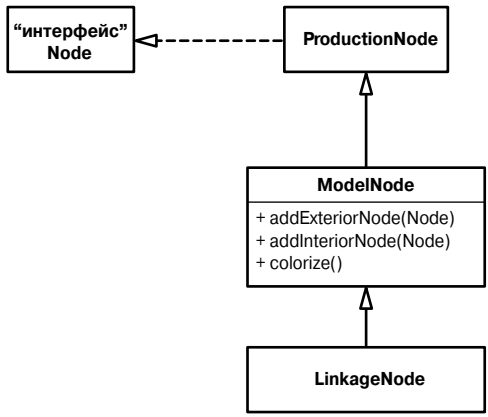


Рис. 25.3. Иерархия наследования после извлечения средства реализации из класса `Node`

Далее извлечем средство реализации из класса `ModelNode`. У класса `ModelNode` уже имеется подкласс, и поэтому мы вводим класс `ProductionNode` в иерархию между классами `ModelNode` и `LinkageNode`. Благодаря этому мы можем сделать `ModelNode` расширением интерфейса `Node`, как показано на рис. 25.4.

После такого встраивания класса в иерархию невольно возникает вопрос: не лучше ли воспользоваться *извлечением интерфейса* и выбрать другие имена для интерфейсов? Ведь это намного более очевидный способ реорганизации кода.

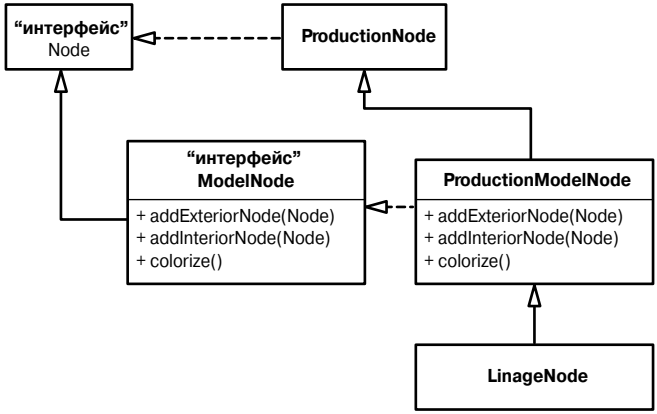


Рис. 25.4. Извлечение средства реализации из класса `ModelNode`

Извлечение интерфейса

Во многих языках программирования *извлечение интерфейса* оказывается одним из самых безопасных способов разрывания зависимостей. Если вы сделаете что-нибудь не так, то компилятор немедленно уведомит вас об этом, а следовательно, вероятность того, что вы внесете программную ошибку, сводится к минимуму. Суть рассматриваемого здесь способа состоит в создании интерфейса для класса с объявлениями всех методов, которые требуется использовать в определенном контексте. Сделав это, можно реализовать интерфейс для распознавания или разделения и передать фиктивный объект в тот класс, который требуется протестировать.

Извлечь интерфейс можно, как минимум, тремя способами, хотя и с некоторыми “особенностями”, на которые следует обратить внимание. Первый способ состоит в использовании инструментального средства автоматической реорганизации кода, если, конечно, таковое имеется в наличии. Подобные инструментальные средства обычно предоставляют в той или иной степени возможность выбрать методы из класса и ввести имя нового интерфейса. Хорошие инструментальные средства запрашивают, следует ли искать в коде места, где можно изменить ссылки на использование нового интерфейса. Такие инструментальные средства позволяют сэкономить немало времени.

Без поддержки для автоматического извлечения интерфейса можно воспользоваться вторым способом, извлекая интерфейс поэтапно, руководствуясь процедурой, описываемой в этом разделе.

И третий способ извлечения интерфейса состоит в том, чтобы скопировать или вырезать из класса сразу несколько методов и поместить их объявления в интерфейс. Это менее безопасный, чем два предыдущих, но все же надежный и зачастую единственный практический способ извлечь интерфейс, когда поддержка для его автоматического извлечения отсутствует, а для компоновки требуется слишком много времени.

Попробуем извлечь интерфейс вторым способом, обсудив по ходу дела некоторые особенности, на которые следует обращать внимание.

Допустим, что нам требуется извлечь интерфейс, чтобы протестировать класс `PaydayTransaction` (транзакция в расчетный день). На рис. 25.5 показан класс

PaydayTransaction и одна из его зависимостей от класса TransactionLog (регистрация транзакций).

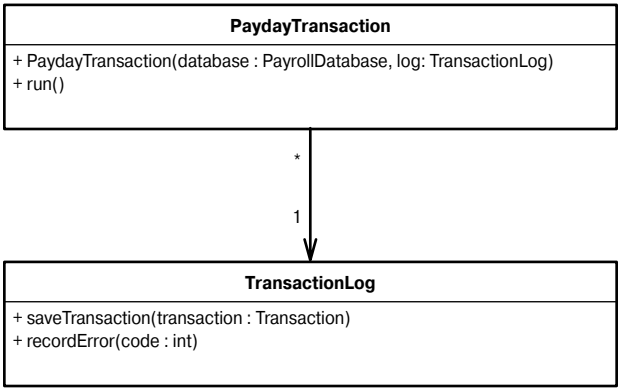


Рис. 25.5. Зависимость класса PaydayTransaction от класса TransactionLog

Ниже приведена первая попытка достичь желаемой цели в контрольном примере.

```
void testPayday()
{
    Transaction t = new PaydayTransaction(getTestingDatabase());
    t.run();

    assertEquals(getSampleCheck(12),
                 getTestingDatabase().findCheck(12));
}
```

Но для того чтобы скомпилировать этот метод, нам придется передать ему какой-то объект типа TransactionLog. Для этого организуем обращение к классу FakeTransactionLog (фиктивная регистрация транзакций), который пока еще не существует.

```
void testPayday()
{
    FakeTransactionLog aLog = new FakeTransactionLog();
    Transaction t = new PaydayTransaction(
                        getTestingDatabase(),
                        aLog);
    t.run();

    assertEquals(getSampleCheck(12),
                 getTestingDatabase().findCheck(12));
}
```

Для того чтобы скомпилировать этот метод, нам придется извлечь интерфейс для класса TransactionLog, реализовать в классе FakeTransactionLog интерфейс, а затем сделать так, чтобы класс PaydayTransaction воспринимал интерфейс FakeTransactionLog.

Прежде всего извлечем интерфейс. С этой целью создадим пустой класс под названием TransactionRecorder (регистратор транзакций). Если вас интересует, откуда взялось это имя, прочитайте следующее примечание.

Присвоение имен интерфейсам

Интерфейсы являются относительно новыми конструкциями в программировании. Они имеются в Java, .NET и многих других языках. А в C++ их приходится имитировать, создавая класс, содержащий лишь чисто виртуальные функции.

Когда интерфейсы были впервые внедрены в языки программирования, некоторые разработчики начали именовать их, добавляя префикс *I* к имени класса, из которого они извлекались. Так, если имеется класс `Account`, то извлекаемый из него интерфейс может быть назван `IAccount`. Преимущество такого способа присвоения имен заключается в том, что, извлекая интерфейс, не нужно придумывать ему имя — достаточно присоединить префикс *I* к имени соответствующего класса. А недостаток такого способа присвоения имен заключается в том, что в базе кода появляются одни имена с префиксом *I*, а другие без такового, и поэтому нужно точно знать, действительно ли код относится к интерфейсу. В идеальном случае это не имеет особого значения. Но если удалить из имени префикс *I*, чтобы вернуться к обычному классу, то такое переименование окажется слишком радикальным, и без соответствующих изменений имя не будет верно отражать назначение конструкции кода.

При разработке новых классов проще всего выбирать для них простые имена — даже если это и крупные абстракции. Так, если проектируется приложение для учета, то начинать лучше всего с класса под названием `Account`. В какой-то момент после тестирования написанного кода новых функций может возникнуть потребность сделать класс `Account` интерфейсом. С этой целью можно создать подкласс ниже данного класса и перенести в него все методы, превратив тем самым `Account` в интерфейс. В этом случае переименовывать все ссылки на `Account` в коде не придется.

В случаях, аналогичных примеру класса `PaydayTransaction`, где уже имеется подходящее имя для интерфейса (`TransactionLog`), можно поступить точно также. Недостаток такого способа заключается в том, что для переноса методов и данных в подкласс требуется предпринять целый ряд шагов. Но если это не очень рискованно, то иногда оказывается вполне оправданным. Такой способ называется *извлечением средства реализации*.

Если у меня нет достаточного количества тестов и мне нужно извлечь интерфейс, чтобы разместить по местам больше тестов, то я зачастую пытаюсь придумать подходящее новое имя для интерфейса. Иногда для этого нужно немного подумать. Без инструментальных средств, автоматически переименовывающих классы, целесообразно тщательно подбирать имена еще до того, как количество мест, где они указываются в коде, сильно возрастет.

```
interface TransactionRecorder
{
}
```

А теперь вернемся к реализации нового интерфейса в классе `TransactionLog`.

```
public class TransactionLog implements TransactionRecorder
{
    ...
}
```

Далее создадим пустой класс FakeTransactionLog.

```
public class FakeTransactionLog implements TransactionRecorder
{
}
```

Теперь весь код должен компилироваться правильно, поскольку мы ввели только ряд новых классов и изменили существующий класс таким образом, чтобы реализовать пустой интерфейс.

Итак, мы подготовили почву для радикальной реорганизации кода. Изменим тип каждой ссылки в тех местах, где нам требуется использовать интерфейс. Так, в классе PaydayTransaction имеется ссылка на класс TransactionLog, которую нам нужно заменить ссылкой на интерфейс TransactionRecorder. Сделав это и скомпилировав код, мы обнаружим целый ряд мест, в которых вызываются методы из интерфейса TransactionRecorder, поэтому избавимся от ошибок компиляции по очереди, введя объявления методов в интерфейс TransactionRecorder и определения пустых методов в класс FakeTransactionLog. Ниже приведен соответствующий пример.

```
public class PaydayTransaction extends Transaction
{
    public PaydayTransaction(PayrollDatabase db,
                             TransactionRecorder log) {
        super(db, log);
    }

    public void run() {
        for(Iterator it = db.getEmployees(); it.hasNext(); ) {
            Employee e = (Employee)it.next();
            if (e.isPayday(date)) {
                e.pay();
            }
        }
        log.saveTransaction(this);
    }
    ...
}
```

В данном примере из интерфейса TransactionRecorder вызывается только метод saveTransaction (сохранить транзакцию). А поскольку этого метода пока еще нет в интерфейсе TransactionRecorder, то мы получаем ошибку компиляции. Для того чтобы успешно скомпилировать тестовый код, нам достаточно ввести метод saveTransaction в интерфейс TransactionRecorder и в класс FakeTransactionLog.

```
interface TransactionRecorder
{
    void saveTransaction(Transaction transaction);
}

public class FakeTransactionLog implements TransactionRecorder
    {
        void saveTransaction(Transaction transaction) {
    }
}
```

Вот, собственно, и все. Нам не пришлось создавать настоящий класс `TransactionLog` для выполнения наших тестов.

Проанализировав данный пример, вы можете возразить: "Пожалуй, это еще не все. Ведь мы не ввели метод `recordError` в интерфейс и фиктивный класс". На самом деле метод `recordError` (регистрировать ошибки) отсутствует в классе `TransactionLog`. Если бы нам потребовалось извлечь интерфейс полностью, мы добавили бы в него и сигнатуру данного метода, но дело в том, что он нам не нужен для тестирования. Конечно, было бы неплохо иметь интерфейс, покрывающий все общедоступные методы данного класса, но если пойти по такому пути, нам пришлось бы приложить намного больше труда, чем требуется для тестирования данного фрагмента кода. Не забывайте, что если вы замыслили такую структуру кода, в которой должен быть ряд ключевых абстракций с интерфейсами, полностью покрывающими ряд общедоступных методов в их классах, то такую структуру можно реализовать постепенно. Иногда лучше отложить внесение изменений в код до тех пор, пока не появится больше тестов, покрывающих эти изменения.

Когда вы извлекаете интерфейс из отдельного класса, не извлекайте из него все общедоступные методы подряд. Сделайте *упор на компилятор*, чтобы обнаружить те методы, которые действительно используются в тестируемом коде.

Как правило, интерфейс извлекается просто. Настоящие трудности возникают лишь в том случае, если приходится иметь дело с невиртуальными методами. В Java ими могут оказаться статические методы. В таких языка, как C# и C++, также допускаются невиртуальные методы экземпляров. Подробнее об этом — в приведенной ниже врезке.

Процедура

Для извлечения интерфейса выполните следующую процедуру.

1. Создайте новый интерфейс с подходящим для его использования именем. Не спешите вводить в него методы.
2. Реализуйте интерфейс в том классе, из которого он извлекается. При этом вы ничего не нарушите, поскольку в интерфейсе отсутствуют методы. Но для надежности убедитесь в этом сами, скомпилировав и выполнив тест.
3. Измените место, где требуется использовать объект, чтобы вместо исходного класса этим местом оказался интерфейс.
4. Выполните компиляцию и введите в интерфейс новое объявление каждого метода, об ошибке использования которого сообщает компилятор.

Извлечение интерфейса и невиртуальные функции

Если в коде встречается такой характерный для многих языков вызов, как, например, `bondRegistry.newFixedYield(client)`, очень трудно сказать, является ли данный метод статическим, виртуальным или же невиртуальным методом экземпляра. В тех языках, где допускаются невиртуальные методы экземпляров, возможны определенные осложнения, если извлечь интерфейс из какого-нибудь класса и добавить в него сигнатуру невиртуального метода из этого класса. Обычно если у класса отсутствуют подклассы, то метод можно сделать виртуальным и затем извлечь интерфейс из этого класса. В этом случае все должно пройти гладко. Но если у класса имеются подклассы, то перенос сиг-

натуры метода в интерфейс может привести к нарушению кода. Рассмотрим следующий пример кода, написанного на C++, где имеется класс с невиртуальным методом.

```
class BondRegistry
{
public:
    Bond *newFixedYield(Client *client) { ... }
};
```

У этого класса имеется подкласс, содержащий метод с таким же самым именем и сигнатурой.

```
class PremiumRegistry : public BondRegistry
{
public:
    Bond *newFixedYield(Client *client) { ... }
};
```

Если извлечь интерфейс из класса `BondRegistry`:

```
class BondProvider
{
public:
    virtual Bond *newFixedYield(Client *client) = 0;
};
```

и реализовать его в классе `BondRegistry` (реестр облигаций):

```
class BondRegistry : public BondProvider { … };
```

то код может оказаться нарушенным, если передать объект класса `PremiumRegistry` (реестр премий) по ссылке следующим образом:

```
void disperse(BondRegistry *registry) {
    ...
    Bond *bond = registry->newFixedYield(existingClient);
    ...
}
```

Перед извлечением интерфейса метод `newFixedYield` был вызван из класса `BondRegistry`, поскольку во время компиляции типом переменной реестра был `BondRegistry`. Если мы сделаем метод `newFixedYield` (новый фиксированный доход) виртуальным в процессе извлечения интерфейса, то тем самым изменим поведение, и тогда произойдет вызов метода из класса `PremiumRegistry`. Если в C++ сделать виртуальным метод из базового класса, то методы, которые переопределяют его в подклассах, также окажутся виртуальными. Подобное затруднение отсутствует в Java или C#. Так, в Java все методы экземпляров являются виртуальными. А в C# дело обстоит чуть более благополучно, поскольку ввод интерфейса не оказывает влияния на существующие вызовы невиртуальных методов.

Как правило, создание в порожденном классе метода с такой же сигнатурой, как и у невиртуального метода в базовом классе, в C++ не приветствуется, поскольку это может привести к недоразумению. Если требуется получить доступ к невиртуальной функции через интерфейс и она отсутствует в классе, не имеющем подклассов, то лучше всего ввести новый виртуальный метод с новым именем. Этот метод может делегировать свои полномочия невиртуальному или даже статическому методу. Нужно лишь убедиться в том, что данный метод правильно обращается с подклассами, расположенными ниже того класса, из которого извлекается интерфейс.

Ввод делегата экземпляра

Разработчики пользуются статическими методами в классах по самым разным причинам. К числу самых распространенных причин относится реализация *шаблона проектирования (синглтона)*, а также создание служебных классов.

Служебные классы можно часто обнаружить во многих структурах кода. Это классы, у которых отсутствуют какие-либо переменные или методы экземпляров. Вместо этого они состоят из ряда статических методов и констант.

Служебные классы создаются по самым разным причинам. Чаще всего они создаются в тех случаях, когда трудно найти общую абстракцию для ряда методов. Примером тому служит класс `Math` из набора инструментальных средств разработки программ на языке Java (JDK). В этом классе содержатся статические методы для тригонометрических (косинуса, синуса, тангенса) и многих других функций. Когда разработчики создают языки программирования из объектов "вниз по иерархии", они обеспечивают правильный порядок выполнения подобных функций в числовых примитивах. Например, вызов метода `sin()` из объекта `1` или любого другого числового объекта дает правильный результат. На момент написания этой книги в Java не поддерживались математические методы в примитивных типах объектов, и поэтому удобным выходом из этого положения был служебный класс, хотя это и особый случай. А в большинстве остальных случаев для этой цели могут использоваться обычные классы с данными и методами экземпляров.

Если в проекте используются статические методы, то они вряд ли доставят хлопоты, если, конечно, не содержат нечто такое, зависимость от чего трудно преодолеть в тесте (это так называемый *статический захват*). На первый взгляд, в подобных случаях можно было бы воспользоваться *объектным швом*, чтобы заменить такое поведение на другое при вызове статических методов. Но что же делать на самом деле?

В качестве выхода из данного положения можно, в частности, попытаться ввести в класс делегирование методов экземпляров. При этом нужно найти способ замены вызовов статических методов вызовами методов экземпляров объекта. Рассмотрим следующий пример.

```
public class BankingServices
{
    public static void updateAccountBalance(int userID,
                                    Money amount) {
        ...
    }
    ...
}
```

Это пример класса, содержащего только статические методы. Здесь показан лишь один статический метод, но рассматриваемый здесь способ распространяется и на все остальные методы данного типа. Мы можем ввести в такой класс метод экземпляра и делегировать его полномочия статическому методу.

```
public static void updateAccountBalance(int userID,
                                Money amount) {
    ...
}

public void updateBalance(int userID, Money amount) {
```

```
    updateAccountBalance(userID, amount);
    }
    ...
}
```

В данном случае мы ввели метод экземпляра под названием `updateBalance` (обновить остаток) и сделали его делегатом статического метода `updateAccountBalance` (обновить остаток на счете).

Теперь мы можем заменить в вызывающей части кода ссылки:

```
public class SomeClass
{
    public void someMethod() {
        ...
        BankingServices.updateAccountBalance(id, sum);
    }
}
```

на следующее:

```
public class SomeClass
{
    public void someMethod(BankingServices services) {
        ...
        services.updateBalance(id,sum);
    }
    ...
}
```

Следует иметь в виду, что все это осуществимо лишь в том случае, если нам удастся найти способ для внешнего создания используемого нами объекта класса `BankingServices` (банковские услуги). Для этого потребуется дополнительная реорганизация кода, но в языках с контролем статических типов мы можем сделать *упор на компилятор*, чтобы получить объект на месте.

Рассматриваемый здесь способ достаточно просто применяется ко многим статическим методам, но в служебных классах это может оказаться не совсем удобным. Ведь класс с пятью или десятью статическими методами и только одним или двумя методами экземпляров выглядит довольно странно. Он будет выглядеть еще более странно, если эти методы будут только делегировать свои полномочия статическим методам. Но применяя данный способ, мы можем без особого труда получить на месте шов и заменить одни виды поведения на другие для тестирования. Со временем мы можем дойти до того, что каждый вызов из служебного класса будет проходить через делегирующие методы. В таком случае тела статических методов можно перенести в методы экземпляров, а сами статические методы удалить.

Процедура

Для ввода делегата экземпляра выполните следующую процедуру.

1. Выявите статический метод, который проблематично использовать в тесте.
2. Создайте метод экземпляра для выбранного статического метода в том же классе. Не забудьте *сохранить сигнатуры*. Сделайте метод экземпляра делегатом статического метода.

3. Найдите в тестируемом коде места, где используется данный статический метод. Воспользуйтесь *параметризацией метода* или другим способом разрыва зависимостей, чтобы предоставить экземпляр в том месте, где вызывается данный статический метод.
4. Замените вызов исходного статического метода, затрудняющего тестирование, вызовом его делегата из экземпляра, введенного в п.3.

Ввод статического установщика

Может быть я и пурист, но мне не нравятся глобально видоизменяемые данные. Мой опыт работы с группами разработчиков показывает, что такие данные обычно являются наиболее очевидным препятствием на пути к тестированию системы по частям. Пытаясь вынести ряд классов в средства тестирования, вы неожиданно обнаруживаете, что некоторые из них нужно установить в определенное состояние, чтобы как-то воспользоваться ими. Настраивая средства тестирования, приходится просматривать список глобальных объектов, которые нужно установить в состояние, необходимое для условий тестирования. Если в квантовой механике такое "призрачное действие на расстоянии" пока еще не удалось обнаружить, то в программировании оно известно уже давно.

Этим недостатком грешат многие системы, несмотря на все огорчения, которые доставляют глобальные данные. В одних системах они совершенно неосознанно объявлены где-нибудь как переменные, а в других — они надевают личину одиночек в строгом соответствии с шаблоном проектирования одиночки. Но в любом случае получить подделку на месте для распознавания не составляет особого труда. Если переменная является явно глобальной и находится вне класса или же просто общедоступной статической переменной, то можно запросто заменить объект. Если же имеется ссылка типа `const` или `final`, то, возможно, придется снять защиту и оставить в коде комментарий, что это делается только для тестирования и не должно быть доступно в выходном коде.

Шаблон проектирования одиночки

Шаблон проектирования одиночки применяется многими разработчиками для того, чтобы в программе присутствовал только один экземпляр конкретного класса. Для многих одиночек общими являются следующие свойства.

1. Конструкторы классов одиночек обычно делаются частными.
2. Статический член класса содержит единственный экземпляр класса, который вообще создается в программе.
3. Статический метод используется для доступа к данному экземпляру. Обычно такой метод называется `instance`, т.е. экземпляр.

Несмотря на то, что одиночки препятствуют созданию более чем одного экземпляра класса в выходном коде, они не позволяют также создавать дополнительные экземпляры этого же класса в средствах тестирования.

Для замены одиночек придется немного потрудиться: сначала ввести статический метод установки, или так называемый установщик, для замены экземпляра, а затем сделать конструктор защищенным. После этого можно выполнить подклассификацию одиночки, создать совершенно новый объект и передать ему установщик.

От самой только мысли снять защиту от доступа, чтобы использовать статический установщик, вам может стать не по себе, но не забывайте, что цель защиты от доступа — препятствовать появлению ошибок. Но мы подвергаем код тестированию с той же целью. Просто в данном случае нам требуется более сильное средство.

Ниже приведен пример ввода статического установщика в коде, написанном на C++.

```
void MessageRouter::route(Message *message) {
    ...
    Dispatcher *dispatcher
            = ExternalRouter::instance()->getDispatcher();
    if (dispatcher != NULL)
        dispatcher->sendMessage(message);
}
```

Для получения диспетчеров в двух местах класса `MessageRouter` (маршрутизатор сообщений) используются одиночки. Одним из этих одиночек является класс `ExternalRouter` (внешний маршрутизатор). В нем используется статический метод под названием `instance` для доступа к одному и только одному экземпляру класса `ExternalRouter`. А для диспетчера в классе `ExternalRouter` имеется соответствующий получатель. Мы можем подменить один диспетчер другим, заменив внешний маршрутизатор, который его обслуживает.

Вот как выглядит класс `ExternalRouter` перед вводом статического установщика.

```
class ExternalRouter
{
private:
    static ExternalRouter *_instance;
public:
    static ExternalRouter *instance();
    ...
};

ExternalRouter *ExternalRouter::_instance = 0;

ExternalRouter *ExternalRouter::instance()
{
    if (_instance == 0) {
        _instance = new ExternalRouter;
    }
    return _instance;
}
```

Обратите внимание на то, что маршрутизатор создается при первом же вызове метода `instance`. Для подмены одного маршрутизатора другим нам придется изменить результат, возвращаемый методом `instance`. Прежде всего, введем новый метод для замены экземпляра.

```
void ExternalRouter::setTestingInstance(ExternalRouter *newInstance)
{
    delete _instance;
    _instance = newInstance;
}
```

При этом, конечно, предполагается, что нам удастся создать новый экземпляр. Когда разработчики пользуются шаблоном одиночки, они зачастую делают конструктор класса частным, чтобы не допустить создание более чем одного экземпляра. Если же мы сделаем конструктор защищенным, то сможем выполнить подклассификацию одиночки для распознавания или разделения и передать новый экземпляр методу `setTestingInstance` (установить тестирующий экземпляр). В приведенном выше примере мы создали подкласс `TestingExternalRouter` (тестирующий внешний маршрутизатор) класса `ExternalRouter` и переопределили метод `getDispatcher` (получить диспетчер) таким образом, чтобы он возвращал требующийся нам диспетчер, т.е. фиктивный диспетчер.

```
class TestingExternalRouter : public ExternalRouter
{
public:
    virtual void Dispatcher *getDispatcher() const {
        return new FakeDispatcher;
    }
};
```

На первый взгляд, это может показаться очень похожим на окольный путь для подстановки нового диспетчера. Ведь в конечном счете мы создаем новый объект класса `ExternalRouter` только ради подстановки диспетчеров. Для этого мы могли бы выбрать и более прямой путь, но тогда нам пришлось бы пойти на другие компромиссы. Кроме того, мы могли бы ввести логически устанавливаемый признак в класс `ExternalRouter` и предоставить ему возможность возвращать другой диспетчер при установке этого признака. А в C++ ли C# для выбора диспетчеров можно воспользоваться логической компиляцией. Все эти приемы вполне работоспособны, но они слишком радикальны и громоздки, чтобы применять их по всему приложению. Обычно я стараюсь сохранять отчетливое разделение между выходным и тестовым кодом.

Применение метода установки и защищенного конструктора в одиночке является менее радикальным подходом, но в то же время оно помогает разместить тесты по местам. Существует ли вероятность того, что кто-нибудь сможет злоупотребить общедоступным конструктором и создать несколько одиночек в выходном коде системы? Да, существует. Но, на мой взгляд, если так важно иметь только один экземпляр объекта в системе, то для этой цели лучше всего добиться того, чтобы все ее разработчики отчетливо понимали данное ограничение.

В качестве другого варианта ослабления защиты конструктора и подклассификации может служить *извлечение интерфейса* из класса одиночки и предоставление установщика, воспринимающего объект с этим интерфейсом. Недостаток такого подхода заключается в необходимости изменять тип ссылки, используемой для сохранения одиночки в классе, а также тип значения, возвращаемого методом `instance`. Такие изменения могут оказаться довольно радикальными и не дать желаемого результата. Существенно лучшего результат можно добиться, сократив количество глобальных ссылок на одиночку до такой степени, чтобы он мог просто стать обычным классом.

В приведенном выше примере мы заменили одиночку, используя статический установщик. Одиночкой в данном случае служил объект, который обслуживал другой объект, т.е. диспетчер. Иногда в системах можно обнаружить другой вид глобального объекта — глобальную фабрику. Вместо того чтобы хранить экземпляр, такая фабрика обслуживает совершенно новые объекты всякий раз, когда вызывается один из ее статических методов. Подстановка другого объекта для возврата оказывается непростым делом, но зачастую ее

можно осуществить, если делегировать полномочия одной фабрики другой. Рассмотрим следующий пример кода, написанного на Java.

```
public class RouterFactory
{
    static Router makeRouter() {
        return new EWNRouter();
    }
}
```

Класс `RouterFactory` представляет собой простую глобальную фабрику. В таком виде эта фабрика не позволяет нам заменить при тестировании маршрутизаторы, которые она обслуживает, но мы можем изменить ее так, чтобы она позволяла это делать.

```
interface RouterServer
{
    Router makeRouter();
}

public class RouterFactory implements RouterServer
{
    static Router makeRouter() {
        return server.makeRouter();
    }

    static setServer(RouterServer server) {
        this.server = server;
    }

    static RouterServer server = new RouterServer() {
        public RouterServer makeRouter() {
            return new EWNRouter();
        }
    };
}
```

А в тесте мы можем сделать следующее:

```
protected void setUp() {
    RouterServer.setServer(new RouterServer() {
        public RouterServer makeRouter() {
            return new FakeRouter();
        }
    });
}
```

Не следует забывать, что в любом из этих шаблонов статических установщиков видоизменяется состояние, доступное для всех тестов. Для возврата в некоторое известное состояние перед выполнением остальных тестов можно воспользоваться методом `tearDown` в среде тестирования xUnit. Обычно я делаю это только в том случае, если установка в неверное состояние в последующем тесте приводит к недоразумению. Если же я подставляю фиктивный объект `MailSender` (отправитель почты) во все свои тесты, то его установка в другое состояние не имеет особого смысла. С другой стороны, если у меня глобальный объект, в котором хранится состояние, оказывающее влияние на результаты работы сис-

темы, то я зачастую делаю одно и то же в методах `setUp` и `tearDown`, чтобы непременно оставить все в одном конкретном состоянии.

```
protected void setUp() {
    Node.count = 0;
    ...
}

protected void tearDown() {
    Node.count = 0;
}
```

Могу себе представить, какое отвращение вызовет у вас такое хирургическое вмешательство в систему только ради того, чтобы разместить тесты по местам. В самом деле, эти шаблоны могут сильно изуродовать отдельные части системы. Но ведь хирургическое вмешательство никогда не бывает приятным, особенно поначалу. Что же можно сделать, чтобы вернуть систему в подходящее состояние?

Для этого можно, например, рассмотреть возможность передачи параметров. Проанализируйте классы, которым требуется доступ к глобальному объекту, и выясните, можно ли создать для них общий суперкласс. Если да, то вы можете передать им глобальный объект после его создания и постепенно избавиться полностью от глобальных объектов. Нередко разработчиков пугает один только факт, что каждому классу в системе может потребоваться какой-нибудь глобальный объект. Вам это покажется удивительным. Однажды мне пришлось работать со встроенной системой, в которой управление памятью и формирование сообщений об ошибках было инкапсулировано в классы, передававшие объект памяти или формирователь сообщений всем, кому это требовалось. Со временем в этой системе было сделано отчетливое разделение на те классы, которым такие услуги требовались, и те классы, которым они не были нужны. У тех классов, которым подобные услуги требовались, имелся общий суперкласс. Объекты, передававшиеся в системе, создавались при запуске программы, и это было явно заметно.

Процедура

Для ввода статического установщика выполните следующую процедуру.

1. Ослабьте защиту конструктора до такой степени, чтобы можно было создать подделку с помощью подклассификации одиночки.
2. Введите статический установщик в класс одиночки. Установщик должен воспринимать ссылку на класс одиночки. Убедитесь в том, что установщик правильно уничтожает экземпляр одиночки перед установкой нового объекта.
3. Если вам потребуется доступ к частным или защищенным методам в одиночке, чтобы установить его в надлежащее состояние для тестирования, то рассмотрите возможность подклассификации одиночки или извлечения интерфейса и сохранения одиночкой своего экземпляра в виде ссылки, тип которой должен соответствовать типу интерфейса.

Подстановка связи

ООП предоставляет нам замечательные возможности для подстановки одного объекта вместо другого. Если в двух классах реализуется один и тот же интерфейс или же у них имеется общий суперкласс, то один из них можно без особого труда подставить вместо другого. К сожалению, у программирующих на процедурных языках, в том числе и на C, такая возможность отсутствует. Так, если имеется функция, аналогичная приведенной ниже, то подставить вместо нее другую функцию во время компиляции можно только с помощью препроцессора.

```
void account_deposit(int amount);
```

Имеется ли этому альтернатива? Да, имеется. Для замены одной функции другой вы можете воспользоваться *подстановкой связи*. С этой целью создайте фиктивную библиотеку функций с такими же сигнатурами, как и у функций, которые вам нужно подделать. Если вы осуществляете распознавание, то вам придется организовать определенный механизм сохранения и запрашивания уведомлений. Для этой цели можно воспользоваться файлами, глобальными переменными и всем, что окажется удобным для тестирования.

Рассмотрим следующий пример.

```
void account_deposit(int amount)
{
    struct Call *call =
        (struct Call *)calloc(1, sizeof (struct Call));
    call->type = ACC_DEPOSIT;
    call->arg0 = amount;
    append(g_calls, call);
}
```

В этом примере нас интересует распознавание, и поэтому мы создаем глобальный список вызовов, регистрируемых при каждом обращении к данной функции (или любой другой имитирующей ее функции). Мы можем проверить такой список в тесте после испытания ряда объектов и выяснить, вызывались ли имитируемые функции в надлежащем порядке.

Мне еще не приходилось пользоваться подстановкой связи в классах C++, но я вполне допускаю, что такая подстановка возможна. Я уверен, что скорректированные имена, формируемые компиляторами C++, могут существенно затруднить подстановку связи, но для вызовов функций C это вполне практически осуществимо, особенно при имитации библиотек внешних функций. Лучше всего имитируются библиотеки, которые представляют собой чистые приемники данных, т.е. функции из таких библиотек можно вызывать, не особенно рассчитывая получить обратно возвращаемые значения. Например, для имитации с помощью подстановки связи особенно подходят библиотеки графических функций.

Подстановкой связи можно воспользоваться и в Java. Для этого достаточно создать классы с аналогичными именами и методами и изменить путь к этим классам, чтобы направить все вызовы именно к ним, а не к классам со скверными зависимостями.

Процедура

Для того чтобы воспользоваться подстановкой связи, выполните следующую процедуру.

1. Выявите функции или классы, которые требуется подделать.
2. Предоставьте для них альтернативные определения.
3. Настройте компоновку таким образом, чтобы в нее были включены альтернативные определения, а не их аналоги из выходного кода.

Параметризация конструктора

Если в конструкторе создается объект, то самый простой способ заменить этот объект нередко состоит в том, чтобы создать его вне класса и передать конструктору в качестве параметра. Обратимся к конкретному примеру.

Итак, начнем со следующего кода.

```
public class MailChecker
{
    public MailChecker (int checkPeriodSeconds) {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
}
```

Затем введем новый параметр следующим образом:

```
public class MailChecker
{
    public MailChecker (MailReceiver receiver,
                        int checkPeriodSeconds) {
        this.receiver = receiver;
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
}
```

Такой прием разработчики нередко упускают из виду, в частности, потому что, по их мнению, это вынуждает клиентов передавать дополнительный аргумент. Но мы можем написать конструктор, сохраняющий исходную сигнатуру.

```
public class MailChecker
{
    public MailChecker (int checkPeriodSeconds) {
        this(new MailReceiver(), checkPeriodSeconds);
    }

    public MailChecker (MailReceiver receiver,
                        int checkPeriodSeconds) {
        this.receiver = receiver;
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
}
```

Сделав это, мы можем предоставить другие объекты для тестирования, а клиенты данного класса даже не заметят разницу.

Но сделаем все порядку. Первоначально наш код выглядит следующим образом.

```
public class MailChecker
{
    public MailChecker (int checkPeriodSeconds) {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
}
```

Сначала создадим копию конструктора.

```
public class MailChecker
{
    public MailChecker (int checkPeriodSeconds) {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
    }

    public MailChecker (int checkPeriodSeconds) {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
}
```

Затем введем в конструктор параметр для объекта `MailReceiver` (получатель почты).

```
public class MailChecker
{
    public MailChecker (int checkPeriodSeconds) {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
}

    public MailChecker (MailReceiver receiver,
                        int checkPeriodSeconds) {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
}
```

Далее присвоим данный параметр переменной экземпляра, избавившись от оператора `new`.

```
public class MailChecker
{
    public MailChecker (int checkPeriodSeconds) {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
```

```
    public MailChecker (MailReceiver receiver,
                        int checkPeriodSeconds) {
        this.receiver = receiver;
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
}
```

А теперь вернемся к исходному конструктору и удалим его тело, заменив его вызовом нового конструктора. В исходном конструкторе используется оператор `new` для создания параметра, который требуется ему передать.

```
public class MailChecker
{
    public MailChecker (int checkPeriodSeconds) {
        this(new MailReceiver(), checkPeriodSeconds);
    }

    public MailChecker (MailReceiver receiver,
                        int checkPeriodSeconds) {
        this.receiver = receiver;
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
}
```

Имеются ли у рассматриваемого здесь способа какие-нибудь недостатки? На самом деле у него имеется один недостаток. Когда мы вводим новый параметр в конструктор, открываем путь для дополнительных зависимостей от класса этого параметра. Клиенты данного класса могут использовать новый конструктор в выходном коде и увеличить количество зависимостей во всей системе. Но, как правило, это не вызывает особых осложнений. Параметризация конструктора представляет собой очень простой вид реорганизации кода, и поэтому я пользуюсь ею очень часто.

В тех языках программирования, где допускаются устанавливаемые по умолчанию аргументы, имеется более простой способ параметризации конструктора. Для этого достаточно ввести устанавливаемый по умолчанию аргумент в существующий конструктор. Ниже приведен конструктор, параметризированный подобным образом в C++.

```
class AssemblyPoint
{
public:
    AssemblyPoint(EquipmentDispatcher *dispatcher
                = new EquipmentDispatcher);
    ...
};
```

Такому применению параметризации конструктора в C++ присущ лишь один недостаток. Заголовочный файл, содержащий подобное объявление класса, должен быть включен в заголовок для класса `EquipmentDispatcher` (диспетчер оборудования). Если бы он отсутствовал для вызова конструктора, то мы могли бы воспользоваться упреждающим объявлением для класса `EquipmentDispatcher`. Именно по этой причине я нечасто пользуюсь устанавливаемыми по умолчанию аргументами в C++.

Процедура

Для параметризации конструктора выполните следующую процедуру.

1. Выявите конструктор, который требуется параметризировать, и сделайте его копию.
2. Введите в конструктор параметр для того объекта, создание которого вы собираетесь заменить. Удалите создание объекта и добавьте присваивание этого параметра переменной экземпляра данного объекта.
3. Если один конструктор вызывается из другого конструктора в используемом вами языке программирования, то удалите тело старого конструктора и замените его вызовом нового конструктора. Введите новое выражение в вызов нового конструктора в старом конструкторе. Если же используемый вами язык программирования не допускает вызов одного конструктора из другого, то вам, возможно, придется извлечь любое дублирование среди конструкторов в новый метод.

Параметризация метода

Допустим, что имеется метод, создающий объект внутренним образом, и этот объект требуется заменить для распознавания или разделения. Зачастую для этого проще всего передать объект извне. Ниже приведен соответствующий пример в коде, написанном на C++.

```
void TestCase::run() {
    delete m_result;
    m_result = new TestResult;
    try {
        setUp();
        runTest(m_result);
    }
    catch (exception& e) {
        result->addFailure(e, this);
    }
    tearDown();
}
```

В этом примере мы имеем дело с методом, создающим объект `TestResult` (результат тестирования) всякий раз, когда он вызывается. Если нас интересует распознавание или разделение, то мы можем передать данный объект в качестве параметра.

```
void TestCase::run(TestResult *result) {
    delete m_result;
    m_result = result;
    try {
        setUp();
        runTest(m_result);
    }
    catch (exception& e) {
        result->addFailure(e, this);
    }
    tearDown();
}
```

Для сохранения исходной сигнатуры в неизменном виде мы можем воспользоваться следующим небольшим методом переадресации:

```
void TestCase::run() {
    run(new TestResult);
}
```

В C++, Java, C# и многих других языках программирования допускается наличие в одном классе двух методов с одинаковыми именами, при условии что у них разные сигнатуры. В рассматриваемом здесь примере мы пользуемся такой возможностью, присваивая одно и то же имя новому параметризированному методу и исходному методу. Иногда такой прием лишь вносит дополнительную путаницу в код, хотя и экономит время. В качестве альтернативы можно указать тип параметра в имени нового метода. Так, в данном примере мы могли бы сохранить `run()` в качестве имени исходного метода, но вызывать этот новый метод следующим образом: `runWithTestResult(TestResult)`.

Как и при *параметризации конструктора*, при параметризации метода может возникнуть зависимость клиентов от новых типов, которые использовались в классе прежде, но отсутствовали в интерфейсе. Если мне кажется, что такая зависимость доставит лишние хлопоты, то вместо этого я рассматриваю возможность для *извлечения и переопределения фабричного метода*.

Процедура

Для параметризации метода выполните следующую процедуру.

1. Выявите метод, который требуется заменить, и сделайте его копию.
2. Введите в метод параметр для того объекта, создание которого вы собираетесь заменить. Удалите создание объекта и добавьте присваивание этого параметра переменной, содержащей данный объект.
3. Удалите тело скопированного метода и организуйте вызов параметризированного метода, используя выражение для создания исходного объекта.

Примитивизация параметра

Самый лучший способ внести изменения в класс состоит, как правило, в следующем: создать его экземпляр в средствах тестирования, написать тест для вносимых изменений, а затем сделать изменения, удовлетворяющие тесту. Но иногда подготовка класса к тестированию требует непомерных усилий. Однажды мне пришлось консультировать группу разработчиков, унаследовавших старую систему с классами предметной области, транзитивно зависевшими практически от каждого второго класса в этой системе. Более того, все они были связаны еще и с базовой структурой сохраняемости. Нельзя сказать, что перенос одного из классов предметной области в среду тестирования оказался невыполнимой задачей, но тогда разработчикам пришлось бы тратить большую часть своего рабочего времени на решение именно этой задачи, совершенно не продвинувшись вперед во внедрении новых свойств в систему. Для разделения была выбрана стратегия, описываемая в этом разделе (рассматриваемый здесь пример был изменен из цензурных соображений).

В инструментальном средстве для составления музыкальных композиций звуковая дорожка обычно представлена в виде нескольких последовательностей музыкальных событий. Нам нужно найти "паузу" в каждой последовательности, чтобы заполнить ее мелкими периодически повторяющимися музыкальными образцами. Для этого нам требуется написать метод под названием `bool Sequence::hasGapFor(Sequence& pattern) const`, который должен возвращать значение, указывающее, вписывается ли музыкальный образец в последовательность.

В идеальном случае этот метод следовало бы поместить в класс `Sequence` (последовательность), но последний относится к числу тех ужасных классов, которые тянут за собой едва ли не всю систему, когда мы пытаемся перенести их в средства тестирования и получить их экземпляры. Поэтому прежде чем разрабатывать этот метод, нам придется решить, как написать для него тест. И это оказывается возможным благодаря внутреннему представлению последовательностей в упрощенном виде. Каждая последовательность состоит из векторов событий. К сожалению, событиям присущ тот же недостаток, что и последовательностям: скверные зависимости, затрудняющие компоновку. Правда, для расчета нам требуется только длительность каждого события. Поэтому мы можем написать еще один метод, рассчитывающий целочисленные значения. Имея в своем распоряжении такой метод, мы сможем написать метод `hasGapFor` (содержит промежуток) и предоставить ему возможность делегировать свои полномочия другому методу.

Итак, начнем с написания первого метода. Ниже приведен тест для него.

```
TEST(hasGapFor, Sequence)
{
    vector<unsigned int> baseSequence;
    baseSequence.push_back(1);
    baseSequence.push_back(0);
    baseSequence.push_back(0);

    vector<unsigned int> pattern;
    pattern.push_back(1);
    pattern.push_back(2);

    CHECK(SequenceHasGapFor(baseSequence, pattern));
}
```

Функция `SequenceHasGapFor` (последовательность содержит промежуток) выполняет роль свободной функции и не принадлежит ни одному из классов, но оперирует представлением, построенным из примитивов (в данном случае — целочисленных без знака). Если мы определим функциональные возможности для функции `SequenceHasGapFor` в средствах тестирования, то сможем написать довольно простую функцию в классе `Sequence`, делегирующую свои полномочия этим новым функциональным возможностям.

```
bool Sequence::hasGapFor(Sequence& pattern) const
{
    vector<unsigned int> baseRepresentation
            = getDurationsCopy();

    vector<unsigned int> patternRepresentation
            = pattern.getDurationsCopy();

    return SequenceHasGapFor(baseRepresentation,
                             patternRepresentation);
}
```

Этой функции требуется другая функция для получения массива длительностей, поэтому напишем и ее.

```
vector<unsigned int> Sequence::getDurationsCopy() const
{
    vector<unsigned int> result;
    for (vector<Event>::iterator it = events.begin();
            it != events.end(); ++it) {
        result.push_back(it->duration);
    }
    return result;
}
```

Итак, нам удалось ввести новое свойство, но далеко не самым лучшим образом. Перечислим те ужасные действия, которые мы здесь совершили.

1. Раскрыли внутреннее представление класса `Sequence`.
2. Усложнили понимание реализации класса `Sequence`, вытеснив ее частично в свободную функцию.
3. Написали нетестированный код (поскольку мы никак не могли написать тест для функции `getDurationsCopy()` — копировать длительности).
4. Сдублировали данные в системе.
5. Отсрочили решение проблемы. Мы так и не приступили к тяжелой работе по разрыву зависимостей между классами предметной области и инфраструктурой. (Это нам еще предстоит сделать, но именно от этого зависит, насколько быстро мы сможет двигаться дальше.)

Несмотря на все эти недочеты, мы все же сумели ввести тестированное свойство. Мне лично не очень нравится такая реорганизация кода, но я вынужден прибегать к ней, когда у меня нет иного выхода. Зачастую ее неплохо выполнять перед *почкованием класса*. Для того чтобы лучше понять это, представьте себе, что функция `SequenceHasGapFor` заключена в оболочку класса под названием `GapFinder` (определитель промежутков).

После примитивизации параметра код остается в довольно скверном состоянии. Поэтому, вообще говоря, лучше ввести новый код в исходный класс или же воспользоваться почкованием класса для построения новых абстракций, которые послужат основанием для дальнейшей работы с кодом. Я прибегаю к примитивизации параметра только в том случае, если уверен, что у меня еще будет время на то, чтобы подвергнуть класс тестированию. На данном этапе функцию можно вложить в класс как настоящий метод.

Процедура

Для примитивизации параметра выполните следующую процедуру.

1. Разработайте свободную функцию, которая выполняет всю работу, которую требуется сделать в классе. По ходу дела разработайте промежуточное представление, которое можно использовать для выполнения данной работы.
2. Введите функцию в класс, который формирует представление и делегирует его новой функции.

Вытягивание свойства

Иногда приходится работать с совокупностью методов класса и зависимостями, которые препятствуют получению экземпляра класса, но не связаны непосредственно с этой совокупностью. Это означает, что в методах, с которыми приходится работать, отсутствует прямая или косвенная ссылка на какие-либо труднопреодолимые зависимости. В подобных случаях можно было бы осуществить неоднократное *раскрытие статического метода* или *вынос объекта метода*, но такой путь к устранению зависимостей необязательно оказался бы самым простым.

В данной ситуации можно попытаться вытянуть совокупность методов, т.е. свойство, в абстрактный суперкласс. Имея в своем распоряжении такой суперкласс, можно выполнить его подклассификацию и создать в тестах экземпляры полученного в итоге подкласса. Рассмотрим следующий пример.

```
public class Scheduler
{
    private List items;

    public void updateScheduleItem(ScheduleItem item)
           throws SchedulingException {
        try {
           validate(item);
        }
        catch (ConflictException e) {
            throw new SchedulingException(e);
        }
        ...
    }

    private void validate(ScheduleItem item)
            throws ConflictException {
        // обратиться к базе данных
        ...
    }

    public int getDeadtime() {
        int result = 0;
        for (Iterator it = items.iterator(); it.hasNext(); ) {
            ScheduleItem item = (ScheduleItem)it.next();
            if (item.getType() != ScheduleItem.TRANSIENT
                    && notShared(item)) {
                result += item.getSetupTime() + clockTime();
            }
            if (item.getType() != ScheduleItem.TRANSIENT) {
                result += item.finishingTime();
            }
            else {
                result += getStandardFinish(item);
            }
        }
```

```
        return result;
    }
}
```

Допустим, что нам требуется внести изменения в метод `getDeadTime` (получить время простоя), а метод `updateScheduleItem` (обновить элемент планирования) нас не интересует. При этом было бы неплохо вообще не связываться с зависимостью от базы данных. С этой целью мы могли бы попытаться сделать метод `getDeadTime` статическим и *раскрыть статический метод*, но мы пользуемся многими нестатическими свойствами класса `Scheduler` (планировщик). Другая возможность заключается в *выносе объекта метода*, но метод `getDeadTime` довольно мал, а зависимости от других методов и полей класса сильно затрудняют тестирование данного метода.

Еще одна возможность состоит в вытягивании интересующего нас метода в суперкласс. Сделав это, мы можем оставить труднопреодолимые зависимости в исходном классе, где они не помешают нашим тестам. Вот как после этого выглядит класс `Scheduler`.

```
public class Scheduler extends SchedulingServices
{
    public void updateScheduleItem(ScheduleItem item)
            throws SchedulingException {
        ...
    }

    private void validate(ScheduleItem item)
            throws ConflictException {
        // обратиться к базе данных
        ...
    }
    ...
}
```

Мы вытянули метод `getDeadTime` (т.е. свойство, которое нам требуется протестировать) и все свойства, которые он использует, в абстрактный класс.

```
public abstract class SchedulingServices
{
    protected List items;

    protected boolean notShared(ScheduleItem item) {
        ...
    }

    protected int getClockTime() {
        ...
    }

    protected int getStandardFinish(ScheduleItem item) {
        ...
    }

    public int getDeadtime() {
        nt result = 0;
        for (Iterator it = items.iterator(); it.hasNext(); ) {
```

```
            ScheduleItem item = (ScheduleItem)it.next();
            if (item.getType() != ScheduleItem.TRANSIENT
                    && notShared(item)) {
                result += item.getSetupTime() + clockTime();
            }
            if (item.getType() != ScheduleItem.TRANSIENT) {
                result += item.finishingTime();
            }
            else {
                result += getStandardFinish(item);
            }
        }
        return result;
    }
    ...
}
```

Теперь мы можем создать тестирующий подкласс, который позволит нам получить доступ к приведенным выше методам в средствах тестирования.

```
public class TestingSchedulingServices extends SchedulingServices
{
    public TestingSchedulingServices() {
    }

    public void addItem(ScheduleItem item) {
        items.add(item);
    }
}

import junit.framework.*;

class SchedulingServicesTest extends TestCase
{
    public void testGetDeadTime() {
        TestingSchedulingServices services
            = new TestingSchedulingServices();
        services.addItem(new ScheduleItem("a",
                         10, 20, ScheduleItem.BASIC));
        assertEquals(2, services.getDeadtime());
    }
    ...
}
```

Итак, мы вытянули методы, которые нам требуется протестировать, т.е. перенесли их вверх по иерархии — в абстрактный суперкласс, и создали конкретный суперкласс, который мы можем теперь использовать для их тестирования.

Насколько такой прием оказывается полезным? С точки зрения структуры кода он далеко не идеален. Ведь мы рассредоточили ряд свойств среди двух классов только ради того, чтобы упростить тестирование. Такое рассредоточение может внести путаницу в код, если взаимосвязь среди свойств в каждом из классов оказывается не очень сильной, как в рассматриваемом здесь примере. В частности, у нас имеется класс `Scheduler`, отвечающий

за обновление элементов планирования, а также класс `SchedulingServices` (службы планирования), отвечающий за самые разные функции, включая получение стандартных сроков для элементов планирования и расчет времени простоя. Возможно, было бы лучше делегировать полномочия класса `Scheduler` некоторому объекту проверки достоверности, которому известно, как обращаться к базе данных, но если делать такой шаг сразу слишком рискованно или же если имеются другие труднопреодолимые зависимости, то вытягивание свойств может оказаться неплохим первоначальным шагом. Если же *сохранить сигнатуры* и сделать *упор на компилятор*, то такой шаг станет еще менее рискованным. А к делегированию мы можем перейти в дальнейшем, когда у нас окажется больше тестов на местах.

Процедура

Для вытягивания свойства выполните следующую процедуру.

1. Выявите методы, которые требуется вытянуть вверх по иерархии.
2. Создайте абстрактный суперкласс для класса, содержащего эти методы.
3. Скопируйте методы в суперкласс и выполните компиляцию.
4. Скопируйте в новый суперкласс каждую недостающую ссылку, о которой предупреждает компилятор. Не забудьте при этом сохранить сигнатуры, чтобы свести к минимуму вероятность появления ошибок.
5. Если оба класса скомпилируются успешно, создайте подкласс для абстрактного класса и введите в него все методы, которые вам требуется подготовить к тестированию.

У вас может возникнуть резонный вопрос: зачем делать суперкласс абстрактным? Я лично предпочитаю делать его абстрактным для того, чтобы мне легче было разбираться в коде. Ведь, анализируя код в приложении, приятно осознавать, что в нем используется каждый конкретный класс. Если же в коде обнаруживаются конкретные классы, экземпляры которых нигде не получаются, то они могут оказаться "мертвым", т.е. невыполняемым кодом.

Вытеснение зависимости

У некоторых классов имеется совсем немного проблематичных зависимостей. Если зависимости содержатся в только в нескольких вызовах методов, то для избавления от них при написании тестов можно воспользоваться *подклассификацией и переопределением метода*. Но если зависимости оказываются глубоко проникающими, то подклассификация и переопределение метода может и не дать желаемого результата. Для устранения зависимостей от конкретных типов данных, возможно, потребуется *извлечь интерфейс* несколько раз. Другая возможность состоит в *вытеснении зависимости*. Такой прием позволяет отделить проблематичные зависимости от остального класса, чтобы упростить работу с ним в средствах тестирования.

Применяя вытеснение зависимости, вы делаете текущий класс абстрактным. Затем вы создаете подкласс, который будет служить вам новым выходным классом, после чего вы вытесняете все проблематичные зависимости в этот класс. Все это позволит вам выпол-

нить подклассификацию исходного класса и сделать его методы доступными для тестирования. Обратимся к следующему примеру кода, написанного на C++.

```
class OffMarketTradeValidator : public TradeValidator
{
private:
    Trade& trade;
    bool flag;

    void showMessage() {
        int status = AfxMessageBox(makeMessage(),
                                   MB_ABORTRETRYIGNORE);
        if (status == IDRETRY) {
            SubmitDialog dlg(this,
               "Нажмите кнопку "OK", если это достоверная торговая сделка");
            dlg.DoModal();
            if (dlg.wasSubmitted()) {
                g_dispatcher.undoLastSubmission();
                flag = true;
            }
        }
        else
        if (status == IDABORT) {
            flag = false;
        }
    }

    public:
        OffMarketTradeValidator(Trade& trade)
        : trade(trade), flag(false)
        {}

        bool isValid() const {
        if (inRange(trade.getDate())
                && validDestination(trade.destination)
                && inHours(trade) {
            flag = true;
        }
        showMessage();
        return flag;
    }
    ...
};
```

Когда нам требуется внести изменения в логику проверки достоверности, можем столкнуться с серьезными трудностями, если откажемся от связывания специфических для пользовательского интерфейса функций и классов со средствами тестирования. В таком случае удобным выходом из положения оказывается вытеснение зависимости.

Ниже показано, как будет выглядеть код после вытеснения зависимости.

```
class OffMarketTradeValidator : public TradeValidator
{
protected:
    Trade& trade;
    bool flag;
    virtual void showMessage() = 0;

public:
    OffMarketTradeValidator(Trade& trade)
    : trade(trade), flag(false) {}

    bool isValid() const {
        if (inRange(trade.getDate())
                && validDestination(trade.destination)
                && inHours(trade) {
           flag = true;
        }
        showMessage();
        return flag;
    }
    ...
};

class WindowsOffMarketTradeValidator
    : public OffMarketTradeValidator
{
protected:
    virtual void showMessage() {
        int status = AfxMessageBox(makeMessage(),
                        MB_ABORTRETRYIGNORE);
        if (status == IDRETRY) {
            SubmitDialog dlg(this,
              "Нажмите кнопку "OK", если это достоверная торговая сделка");
            dlg.DoModal();
            if (dlg.wasSubmitted()) {
                g_dispatcher.undoLastSubmission();
                flag = true;
            }
        }
        else
        if (status == IDABORT) {
            flag = false;
        }
    }
    ...
};
```

Когда у нас имеются специфические для пользовательского интерфейса функции, вытесненные в новый подкласс (`WindowsOffMarketValidator`), мы можем создать еще один подкласс для тестирования. Нам остается лишь аннулировать поведение метода `showMessage` (показать сообщение).

```
class TestingOffMarketTradeValidator
        : public OffMarketTradeValidator
{
protected:
    virtual void showMessage() {}
};
```

Теперь у нас имеется класс, который мы можем протестировать, поскольку он освобожден от зависимостей от пользовательского интерфейса. Насколько идеальным оказывается такое применение наследования? Оно, конечно, не идеально, но помогает подготовить часть логики класса к тестированию. Когда у нас появятся тесты для класса `OffMarketTradeValidator` (средство проверки достоверности внерыночной торговой сделки), мы сможем приступить к окончательному редактированию логики повторения и ее вытягиванию из класса `WindowsOffMarketValidator`. Как только у нас останутся лишь вызовы пользовательского интерфейса, можно перейти к их делегированию новому классу. В этом новом классе в итоге окажутся только зависимости от пользовательского интерфейса.

Процедура

Для вытеснения зависимости выполните следующую процедуру.

1. Попробуйте создать класс с проблематичными зависимостями в избранных вами средствах тестирования.
2. Выявите зависимости, затрудняющие компоновку.
3. Создайте новый подкласс с именем, отражающим особенности окружения данных зависимостей.
4. Скопируйте переменные и методы экземпляров, содержащие труднопреодолимые зависимости, в новый подкласс, не забыв сохранить сигнатуры. Сделайте методы защищенными в исходном классе, а сам исходный класс — абстрактным.
5. Создайте тестирующий подкласс и видоизмените тест, чтобы попробовать получить экземпляр этого подкласса.
6. Создайте тесты, чтобы проверить, можно ли получить экземпляр нового класса.

Замена функции указателем функции

Для разрыва зависимостей в процедурных языках программирования отсутствует столько же возможностей, сколько их имеется в объектно-ориентированных языках, в том числе *инкапсуляция глобальных ссылок* и *подклассификация и переопределение метода*. Все эти возможности оказываются недоступными. Вместо этого можно было бы воспользоваться *подстановкой связи* или же *расширением определения*, но зачастую эти меры оказываются излишними для борьбы с относительно легко разрываемыми зависимостями. В качестве альтернативы в процедурных языках программирования, поддерживающих указатели функций, может служить *замена функции указателем функции*. Наиболее известным языком, в котором поддерживаются указатели функций, является C.

В среде программистов сложились разные представления об указателях функций. Одни считают указатели функций ужасно ненадежными, поскольку существует вероят-

ность испортить их содержимое и в конечном счете обратиться к произвольной области памяти. Другие же считают указатели функций полезным средством, которым, впрочем, следует пользоваться очень аккуратно. Если вы принадлежите к числу последних, то у вас появляется возможность отделить такие зависимости, которые очень трудно или вообще нельзя разорвать иным способом.

Прежде всего рассмотрим указатель функции в его естественном окружении. В приведенном ниже примере показано объявление нескольких указателей функций и ряд вызовов, осуществляемых с их помощью в C.

```
struct base_operations
{
    double (*project)(double,double);
    double (*maximize)(double,double);
};

double default_projection(double first, double second) {
    return second;
}

double maximize(double first, double second) {
    return first + second;
}

void init_ops(struct base_operations *operations) {
    operations->project = default_projection;
    operations->maximize = default_maximize;
}

void run_tesselation(struct node *base,
                  struct base_operations *operations) {
    double value = operations->project(base.first, base.second);
    ...
}
```

Указатели функций позволяют выполнять ряд очень простых операций объектно-ориентированного программирования, но насколько они полезны для разрыва зависимостей? Рассмотрим следующий пример.

Допустим, что у нас сетевое приложение, сохраняющее информацию о пакетах в оперативно доступной базе данных. Обращение к базе данных осуществляется следующим образом:

```
void db_store(
    struct receive_record *record,
    struct time_stamp receive_time);
struct receive_record * db_retrieve(time_stamp search_time);
```

Для связывания с новыми телами приведенных выше функций мы могли бы воспользоваться *подстановкой связи*, но иногда подстановка связи приводит к нетривиальным изменениям в компоновке. Нам бы пришлось тогда выделять из библиотек те функции, которые нам требуется сымитировать. Но самое главное, что швы, которые мы получаем при подстановке связи, не совсем подходят для изменения поведения в выходном коде. Если код требуется протестировать и в то же время обеспечить удобство изменения, например,

типа базы данных, к которой код обращается, то для этой цели окажется полезной замена функции указателем функции. Итак, рассмотрим этот способ по порядку.

Прежде всего выявим объявление функции, которую нам требуется заменить.

```
// db.h
void db_store(struct receive_record *record,
              struct time_stamp receive_time);
```

Затем объявим указатель функции с таким же именем.

```
// db.h
void db_store(struct receive_record *record,
              struct time_stamp receive_time);

void (*db_store)(struct receive_record *record,
                 struct time_stamp receive_time);
```

Далее переименуем исходное определение.

```
// db.h
void db_store_production(struct receive_record *record,
                         struct time_stamp receive_time);

void (*db_store)(struct receive_record *record,
                 struct time_stamp receive_time);
```

После этого инициализируем указатель в исходном файле C.

```
// main.c
extern void db_store_production(
               struct receive_record *record,
               struct time_stamp receive_time);
void initializeEnvironment() {
    db_store = db_store_production;
    ...
}

int main(int ac, char **av) {
    initializeEnvironment();
    ...
}
```

И наконец, выявим определение функции `db_store` и переименуем ее на `db_store_production` (сохранить в базе данных при эксплуатации).

```
// db.c
void db_store_production(
        struct receive_record *record,
        struct time_stamp receive_time) {
    ...
}
```

Теперь мы можем выполнить компиляцию и тестирование.

Когда указатели функций находятся на местах, тесты способны предоставить альтернативные определения для распознавания или разделения.

Замена функции указателем функции является удобным способом разрыва зависимостей. Одна из замечательных особенностей такой замены состоит в том, что она происходит только во время компиляции и поэтому оказывает минимальное влияние на систему компоновки. Но если вы применяете этот способ в C, то рассмотрите возможность для перехода к C++, чтобы воспользоваться всеми остальными швами, которые предоставляются в C++. На момент написания этой книги во многих компиляторах C были доступны ключи для выполнения смешанной компиляции кода C и C++. Используя это свойство, можно постепенно перенести свой проект из C в C++, выбрав только те файлы, которые потребуются вам для разрывания зависимостей.

Процедура

Для замены функции указателем функции выполните следующую процедуру.

1. Выявите объявления функций, которые требуется заменить.
2. Создайте указатели функций с такими же точно именами перед каждым объявлением функции.
3. Переименуйте исходные объявления функций, чтобы они отличались от указателей функций, которые вы только что объявили.
4. Инициализируйте указатели адресами старых функций в исходном файле C.
5. Просмотрите всю структуру кода и найдите в ней тела старых функций. Присвойте им имена новых функций.

Замена глобальной ссылки получателем

Глобальные переменные способны доставить немало хлопот, когда приходится работать с фрагментами кода по отдельности. Но об этом уже шла речь при рассмотрении способа *ввода статического установщика*, и поэтому не будем здесь повторяться.

Для того чтобы устранить зависимости от глобальных переменных, можно, в частности, ввести методы получения, или так называемые получатели, в класс для каждой такой переменной. Имея в своем распоряжении получатели, можно далее осуществить *подклассификацию и переопределение метода*, чтобы получатели возвращали нечто подходящее. В некоторых случаях можно даже пойти еще дальше, воспользовавшись *извлечением интерфейса* для разрыва зависимостей от класса глобального объекта. Рассмотрим следующий пример кода, написанного на Java.

```
public class RegisterSale
{
    public void addItem(Barcode code) {
        Item newItem =
          Inventory.getInventory().itemForBarcode(code);
        items.add(newItem);
    }
    ...
}
```

В приведенном выше коде осуществляется доступ к глобальному объекту из класса `Inventory` (товарные запасы). Вы можете спросить: "Какой же это глобальный объект? Ведь это просто вызов статического метода из класса". Дело в том, что в Java класс сам является глобальным объектом, а в данном коде он, по-видимому, должен обращаться к некоторому состоянию, чтобы выполнить свою функцию (возвратить объекты товаров по заданным штриховым кодам). Можно ли как-то избавиться от такой зависимости с помощью *замены глобальной ссылки получателем*? Попробуем сделать это.

Прежде всего напишем получатель. Мы сделаем его защищенным, чтобы переопределить этот метод при тестировании.

```
public class RegisterSale
{
    public void addItem(Barcode code) {
       Item newItem = Inventory.getInventory().itemForBarcode(code);
       items.add(newItem);
    }

    protected Inventory getInventory() {
        return Inventory.getInventory();
    }
    ...
}
```

Затем заменим каждый пример доступа к глобальному объекту получателем.

```
public class RegisterSale
{
    public void addItem(Barcode code) {
        Item newItem = getInventory().itemForBarcode(code);
        items.add(newItem);
    }

    protected Inventory getInventory() {
        return Inventory.getInventory();
    }
    ...
}
```

Теперь мы можем создать подкласс класса `Inventory`, чтобы использовать его в тесте. А поскольку класс `Inventory` является одиночкой, нам придется сделать его конструктор защищенным, а не частным. После этого, мы можем создать аналогичным образом еще один подкласс и поместить его в ту логику, которая потребуется нам в тесте для преобразования штриховых кодов в конкретные товары.

```
public class FakeInventory extends Inventory
{
    public Item itemForBarcode(Barcode code) {
    ...
    }
    ...
}
```

Создадим теперь класс, который мы используем в тесте.

```
class TestingRegisterSale extends RegisterSale
{
    Inventory inventory = new FakeInventory();

    protected Inventory getInventory() {
        return inventory;
    }
}
```

Процедура

Для замены глобальной ссылки получателем выполните следующую процедуру.

1. Выявите глобальную ссылку, которую требуется заменить.
2. Напишите получатель для глобальной ссылки. Защита этого метода получения должна быть достаточно слабой, чтобы его можно было переопределить в подклассе.
3. Замените все ссылки на глобальный объект вызовами получателя.
4. Создайте тестирующий подкласс и переопределите получатель.

Подклассификация и переопределение метода

Подклассификация и переопределение метода является основным способом разрыва зависимостей в объектно-ориентированных программах. В действительности, многие из тех способов разрыва зависимостей, которые представлены в этой главе, являются разновидностями данного способа.

В основу подклассификации и переопределения метода положен следующий принцип: наследование может быть использовано в контексте теста для аннулирования ненужного поведения или же для получения доступа к поведению, которое представляет интерес.

Рассмотрим для примера метод из небольшого приложения.

```
class MessageForwarder
{
    private Message createForwardMessage(Session session,
                                Message message)
            throws MessagingException, IOException {
        MimeMessage forward = new MimeMessage (session);
        forward.setFrom (getFromAddress (message));
        forward.setReplyTo (
                new Address [] {
                    new InternetAddress (listAddress) });
        forward.addRecipients (Message.RecipientType.TO,
                               listAddress);
        forward.addRecipients (Message.RecipientType.BCC,
                               getMailListAddresses ());
        forward.setSubject (
                transformedSubject (message.getSubject ()));
        forward.setSentDate (message.getSentDate ());
        forward.addHeader (LOOP_HEADER, listAddress);
```

```
        buildForwardContent(message, forward);

        return forward;
    }
    ...
}
```

В классе `MessageForwarder` имеется довольно много методов, которые здесь не показаны. В частности, один из общедоступных методов вызывает частный метод `createForwardMessage` для формирования нового пересылаемого сообщения. Допустим, что для тестирования нам не нужна зависимость от класса `MimeMessage` (сообщение типа MIME). В нем используется переменная `session`, а мы не собираемся организовывать настоящий сеанс связи при тестировании. Если же нам требуется отделить зависимость от класса `MimeMessage`, то мы можем сделать метод `createForwardMessage` защищенным и переопределить его в новом подклассе, который мы создаем только для тестирования.

```
class TestingMessageForwarder extends MessageForwarder
{
    protected Message createForwardMessage(Session session,
                                Message message) {
        Message forward = new FakeMessage(message);
        return forward;
    }
    ...
}
```

В этом новом подклассе мы можем сделать все, что нам потребуется для разделения или распознавания. В данном случае мы, по существу, аннулируем бо́льшую часть поведения в методе `createForwardMessage`, но если оно не требуется нам для конкретного тестирования, то нас это обстоятельство вполне устроит.

В выходном коде мы получаем экземпляры объектов класса `MessageForwarder`, а в тестах — экземпляры объектов класса `TestingMessageForwarder`. Таким образом, мы сумели добиться разделения, минимально видоизменив производственный код. Для этого нам было достаточно изменить область действия метода, сделав его защищенным вместо частного.

Как правило, разделение, появляющееся в классе после вынесения за скобки, определяет, насколько хорошо наследование может быть использовано для обособления зависимостей. В одних случаях зависимость, от которой требуется избавиться, уже обособлена в небольшом методе, а в других — для отделения зависимости приходится переопределять более крупный метод.

Подклассификация и переопределение метода — довольно эффективный способ, но пользоваться им следует очень аккуратно. В приведенном выше примере возвращается пустое сообщение без темы, адреса отправителя и прочих атрибутов, но это имело бы смысл, если бы нам пришлось проверять, например, факт пересылки сообщения из одной части программы в другую, не обращая особого внимания на его содержимое и адресацию.

Программирование представляется мне очень наглядным процессом. Когда я программирую, в моем уме возникают всевозможные картины, помогающие мне сделать правильный выбор среди разных проектных решений. К сожалению, блок-схемы UML не могут и близко сравниться с подобными картинами, но они оказывают мне помощь в работе.

Одну из таких картин, часто возникающих в моем воображении в процессе программирования, я называю *бумажным представлением*. Анализируя метод, я начинаю представ-

лять себе различные способы группирования операторов и выражений. При этом я хорошо осознаю, что практически любой мелкий фрагмент кода, который мне удается выявить в методе, я могу заменить чем-то другим во время тестирования, если я сумею извлечь его в другой метод. Это можно сравнить с наложением листа полупрозрачной бумаги на другой лист с распечатанным кодом. На новом листе бумаги может оказаться другой код для фрагмента, который мне требуется заменить. Стопка бумаги в таком бумажном представлении символизирует то, что мне требуется протестировать, а методы, которые просматриваются сквозь верхний лист бумаги, обозначают те методы, которые я могу выполнить при тестировании. На рис. 25.6 предпринята попытка продемонстрировать описанное здесь бумажное представление класса.

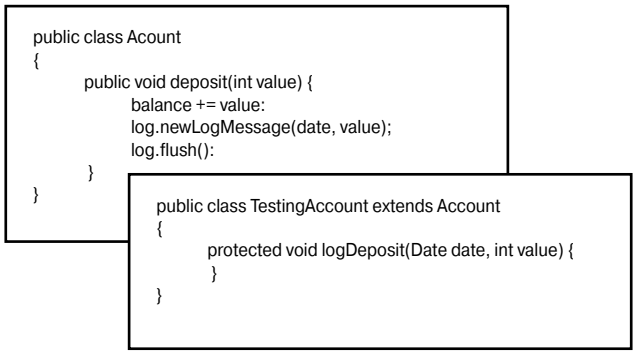


Рис. 25.6. Наложение класса `TestingAccount` *на класс* `Account` *в бумажном представлении*

Бумажное представление помогает мне лучше видеть, что можно сделать с кодом, но, когда я начинаю пользоваться подклассификацией и переопределением метода, стараюсь переопределить те методы, которые уже существуют. Ведь моя цель — разместить тесты по местам, а извлечение методов без тестов на местах иногда оказывается рискованным.

Процедура

Для подклассификации и переопределении метода выполните следующую процедуру.

1. Выявите зависимости, которые вам требуется обособить, или же место, где требуется выполнить распознавание. Постарайтесь найти наименьшую совокупность методов, которые можно переопределить, чтобы добиться поставленной цели.
2. Сделайте каждый метод переопределяемым. Конкретный способ сделать это зависит от используемого вами языка программирования. Так, в C++ методы должны быть сделаны виртуальными, если они таковыми еще не являются. В Java методы должны быть сделаны неконечными. А во многих языках на платформе .NET каждый метод следует сделать явно переопределяемым.
3. Откорректируйте область действия методов, которые вы переопределяете, если этого требует используемый вами язык программирования, чтобы их можно было переопределить и в подклассе. В Java и C# методы должны быть, по крайней мере, защищенными, чтобы их можно было переопределить в подклассе. А в C++ методы должны оставаться частными и, тем не менее, переопределяемыми в подклассе.

4. Создайте подкласс, переопределяющий методы. Убедитесь в том, что вы можете встроить его в свои средства тестирования.

Замена переменной экземпляра

Создание объектов в конструкторах может оказаться затруднительным, особенно если этому препятствует зависимость от этих объектов в тесте. Как правило, подобное затруднение можно обойти, используя *извлечение и переопределение фабричного метода*. Но в тех языках программирования, где не допускается переопределение вызовов виртуальных функций в конструкторах, приходится искать другие возможности. Одна из них состоит в *замене переменной экземпляра*.

Рассмотрим следующий пример, демонстрирующий затруднение, возникающее в связи с виртуальными функциями в C++.

```
class Pager
{
public:
    Pager() {
        reset();
        formConnection();
    }

    virtual void formConnection() {
        assert(state == READY);
       // здесь следует скверный код обращения
       // к аппаратным средствам
       ...
    }

    void sendMessage(const std::string& address,
                     const std::string& message) {
        formConnection();
        ...
    }
    ...
};
```

В данном примере метод `formConnection` (соединение с формой) вызывается в конструкторе. Конечно, нет ничего дурного в том, что конструкторы делегируют свои полномочия другим функциям, но это приводит к не совсем верному представлению о данном коде. Метод `formConnection` объявляется как виртуальный, и поэтому, на первый взгляд, кажется, что нам достаточно осуществить *подклассификацию и переопределение метода*. Но не будем спешить с выводами. Попробуем сначала воспользоваться этим способом.

```
class TestingPager : public Pager
{
public:
    virtual void formConnection() {
    }
};
```

```
TEST(messaging,Pager)
{
    TestingPager pager;
    pager.sendMessage("5551212",
                      "Привет, пойдем на вечеринку? XXXOOO");
    LONGS_EQUAL(OKAY, pager.getStatus());
}
```

Когда мы переопределяем виртуальную функцию в C++, заменяем, как и предполагалось, поведение этой функции в производном классе, но за одним исключением: если виртуальная функция вызывается в конструкторе, то данный язык программирования не допускает ее переопределение. В рассматриваемом здесь примере это означает, что при вызове метода `sendMessage` используется вызов метода `TestingPager::formConnection`. Ну и ладно, ведь мы же не хотим, чтобы приведенное выше флиртующее сообщение было отправлено оператору информации. Но к сожалению, оно уже отправлено. Ведь когда мы построили объект класса `TestingPager`, во время инициализации произошел вызов метода `Page::formConnection`, поскольку в C++ не допускается его переопределение в конструкторе.

Это правило соблюдается в C++, поскольку вызовы переопределенных виртуальных функций из конструктора могут оказаться ненадежными.

```
class A
{
public:
    A() {
        someMethod();
    }

    virtual void someMethod() {
    }
};

class B : public A
{
    C *c;
public:

    B() {
        c = new C;
    }

    virtual void someMethod() {
        c.doSomething();
    }
};
```

В приведенном выше коде метод `someMethod` из класса `B` переопределяет аналогичный метод из класса `A`. Но не следует забывать о порядке вызовов конструкторов. Когда мы создаем объект класса `B`, конструктор класса `A` вызывается прежде конструктора класса `B`. Следовательно, конструктор класса `A` вызывает метод `someMethod`, и этот метод переопределяется, а значит, используется метод `someMethod` из класса `B`. Данный метод, в

свою очередь, пытается вызвать метод doSomething по ссылке типа C, но, как нетрудно догадаться, он вообще не инициализируется, поскольку конструктор класса B не был еще выполнен.

В C++ ничего подобного не допускается, а других языках программирования допускается отчасти. Например, переопределенные методы могут вызываться из конструкторов в Java, но делать это в выходном коде все же не рекомендуется.

В C++ такой незначительный механизм защиты препятствует замене поведения в конструкторах. Правда, у нас имеются и другие возможности. Так, если заменяемый объект не используется в конструкторе, то для разрывания зависимостей можно прибегнуть к *извлечению и переопределению получателя*. Если же объект используется, но нужно убедиться в возможности его замены перед вызовом другого метода, то целесообразно прибегнуть к замене переменной экземпляра. Рассмотрим следующий пример.

```
BlendingPen::BlendingPen()
{
    setName("BlendingPen");
    m_param = ParameterFactory::createParameter(
                      "cm", "Fade", "Aspect Alter");
    m_param->addChoice("blend");
    m_param->addChoice("add");
    m_param->addChoice("filter");

    setParamByName("cm", "blend");
}
```

В этом примере конструктор создает параметр за счет фабрики. Мы могли бы осуществить *ввод статического установщика*, чтобы получить управление очередным объектом, возвращаемым фабрикой, но такое изменение кода было бы слишком радикальным. Если мы не собираемся вводить в класс дополнительный метод, то можем заменить параметр, созданный в конструкторе, следующим образом:

```
void BlendingPen::supersedeParameter(Parameter *newParameter)
{
    delete m_param;
    m_param = newParameter;
}
```

В тестах мы можем создать объекты перьев по мере потребности в них и вызвать метод supersedeParameter (заменить параметр), когда нам нужно будет ввести объект распознавания.

Как правило, предоставление получателей, изменяющих базовые объекты, используемые конкретным объектом, считается порочной практикой. Ведь такие получатели дают клиентам возможность существенно изменить поведение объекта в течение срока его действия. Если кому-нибудь удастся сделать подобные изменения, то нужно знать предысторию данного объекта, чтобы понять, что именно происходит при вызове одного из его методов. В отсутствие получателей код легче понять.

На первый взгляд, замена переменной экземпляра кажется не очень удачным способом получить объект распознавания на месте, но если в C++ *параметризация конструктора* оказывается затруднительной из-за слишком запутанной логики конструктора, то замена переменной экземпляра может стать едва ли не самым лучшим выходом из положения. А

в тех языках программирования, где допускаются вызовы виртуальных функций из конструкторов, лучше всего прибегнуть к *извлечению и переопределению фабричного метода*.

У префикса `supersede` в имени заменяющего метода имеется одна замечательная особенность: он необычен и привлекает к себе внимание. Поэтому если вас интересует, используются ли в коде заменяющие методы, вы можете быстро обнаружить их по этому префиксу.

Процедура

Для замены переменной экземпляра выполните следующую процедуру.

1. Выявите переменную экземпляра, которую требуется заменить.
2. Создайте метод с именем `supersedeXXX`, где `XXX` — имя переменной, которую требуется заменить.
3. Напишите в этом методе любой код, который требуется для уничтожения предыдущего экземпляра переменной и присвоения ей нового значения. Если переменная оказывается ссылкой, проверьте наличие в классе любых других ссылок на тот объект, на который она ссылается. Если же такие ссылки есть, то в заменяющем методе, возможно, придется организовать дополнительный механизм, гарантирующий безопасную замену объекта и правильный результат.

Переопределение шаблона

Многие способы разрыва зависимостей, представленные в этой главе, опираются на такие базовые объектно-ориентированные механизмы, как наследование интерфейса и реализации. Ряд новых свойств языков программирования предоставляет дополнительные возможности. Так, если в языке допускаются обобщения и совмещение типов, то зависимости можно разорвать способом, который называется *переопределением шаблона*. Рассмотрим следующий пример кода, написанного на C++.

```
// AsyncReceptionPort.h

class AsyncReceptionPort
{
private:
    CSocket m_socket;
    Packet m_packet;
    int m_segmentSize;
    ...

public:
        AsyncReceptionPort();
    void Run();
    ...
};

// AsynchReceptionPort.cpp

void AsyncReceptionPort::Run() {
```

```
    for(int n = 0; n < m_segmentSize; ++n) {
        int bufferSize = m_bufferMax;
        if (n = m_segmentSize - 1)
            bufferSize = m_remainingSize;
        m_socket.receive(m_receiveBuffer, bufferSize);
        m_packet.mark();
        m_packet.append(m_receiveBuffer,bufferSize);
        m_packet.pack();
    }
    m_packet.finalize();
}
```

Если у нас имеется подобный код и нам требуется внести изменения в логику конкретного метода, то мы сталкиваемся с тем фактом, что нельзя выполнить данный метод в средствах тестирования, не пересылая что-нибудь через сокет. В C++ мы можем избежать этого полностью, сделав `AsyncReceptionPort` (порт асинхронного приема) шаблоном, а не обычным классом. Ниже показано, как будет выглядеть код после подобного изменения.

```
// AsynchReceptionPort.h
template<typename SOCKET> class AsyncReceptionPortImpl
{
private:
    SOCKET m_socket;
    Packet m_packet;
    int m_segmentSize;
    ...

public:
    AsyncReceptionPortImpl();
    void Run();
    ...
};

template<typename SOCKET>
void AsyncReceptionPortImpl<SOCKET>::Run() {
    for(int n = 0; n < m_segmentSize; ++n) {
        int bufferSize = m_bufferMax;
        if (n = m_segmentSize - 1)
            bufferSize = m_remainingSize;
        m_socket.receive(m_receiveBuffer, bufferSize);
        m_packet.mark();
        m_packet.append(m_receiveBuffer,bufferSize);
        m_packet.pack();
    }
    m_packet.finalize();
}

typedef AsyncReceptionPortImpl<CSocket> AsyncReceptionPort;
```

После этого изменения мы можем получить экземпляр шаблона с другим типом в тестовом файле.

```
// TestAsynchReceptionPort.cpp

#include "AsyncReceptionPort.h"

class FakeSocket
{
public:
    void receive(char *, int size) { ... }
};

TEST(Run,AsyncReceptionPort)
{
    AsyncReceptionPortImpl<FakeSocket> port;
    ...
}
```

Самое замечательное в этом способе — это возможность использовать директиву `typedef`, чтобы исключить изменение ссылок во всей базе кода. Без этого нам пришлось бы заменять каждую ссылку на класс `AsyncReceptionPort` ссылкой на сокет `AsyncReceptionPort<CSocket>`. Для проверки правильности замены всех ссылок мы можем сделать *упор на компилятор*. В тех языках программирования, где поддерживаются обобщения, но отсутствует такой механизм совмещения типов, как `typedef`, приходится делать упор на компилятор.

Данный способ можно использовать в C++ для предоставления альтернативных определений методов, а не данных, хотя это и непросто сделать. Правила языка C++ обязывают предоставлять параметр шаблона. Следовательно, выбрав переменную, можно произвольно задать параметр шаблона или же ввести новую переменную только ради параметризации класса по определенному типу, но это рекомендуется делать лишь в крайнем случае. А прежде следует очень внимательно изучить возможность использования способов, основанных на наследовании.

Переопределению шаблона в C++ присущ главный недостаток: без компилятора C++ с очень хорошей поддержкой шаблонов код из файлов реализации переносится в заголовочные файлы при его формировании по шаблону, что может привести к усилению зависимостей в системе. В этом случае пользователям шаблона приходится перекомпилировать его код всякий раз, когда шаблон изменяется.

Я обычно склоняюсь к использованию способов, основанных на наследовании, для разрыва зависимостей в C++. Тем не менее переопределение шаблона может оказаться полезным, когда зависимости, которые требуется разорвать, уже находятся в шаблонном коде. Рассмотрим следующий пример.

```
template<typename ArcContact> class CollaborationManager
{
    ...
    ContactManager<ArcContact> m_contactManager;
    ...
};
```

Если нам требуется разорвать зависимость от класса `m_contactManager`, то мы не можем так просто *извлечь* из него *интерфейс*, поскольку мы используем в данном коде шаблон именно так, а не иначе. Но мы можем параметризировать шаблон по-другому.

```
template<typename ArcContactManager> class CollaborationManager
{
    ...
    ArcContactManager m_contactManager;
    ...
};
```

Процедура

В этой процедуре описывается переопределение шаблона в C++. В языках программирования, поддерживающих обобщения, эта процедура может оказаться иной, но она отражает саму суть данного способа.

1. Выявите свойства, которые требуется заменить в тестируемом классе.
2. Преобразуйте класс в шаблон, параметризировав его переменными, которые требуется заменить, и скопировав тела методов в заголовок.
3. Присвойте шаблону другое имя. Для этого можно, в частности, присоединить к исходному имени суффикс `Impl` (реализация).
4. Введите после определения шаблона директиву `typedef`, задав исходные аргументы шаблона с помощью имени исходного класса.
5. Включите в тестовый файл определение шаблона и получите экземпляр шаблона по новым типам, чтобы провести замену типов для целей тестирования.

Переопределение исходного текста

В ряде современных интерпретируемых языков программирования предоставляется изящный способ разрывания зависимостей. Методы можно переопределить в процессе самой интерпретации. Ниже приведен пример кода, написанного на языке Ruby.

```
# Account.rb
class Account
    def report_deposit(value)
        ...
    end

    def deposit(value)
        @balance += value
        report_deposit(value)
    end

    def withdraw(value)
        @balance -= value
    end
end
```

Если нам не требуется, чтобы метод `report_deposit` (отчет по вкладам) выполнялся во время тестирования, то мы можем переопределить его в тестовом файле и поместить тесты после переопределения.

```
# AccountTest.rb
require "runit/testcase"
require "Account"

class Account
    def report_deposit(value)
    end
end

# здесь начинаются тесты
class AccountTest < RUNIT::TestCase
    ...
end
```

Следует заметить, что мы переопределяем здесь не весь класс `Account`, а только метод `report_deposit`. Интерпретатор интерпретирует все строки исходного текста из файла Ruby в виде исполняемых операторов. В частности, оператор `class Account` открывает определение класса `Account`, чтобы ввести в него дополнительные определения. А оператор `def report_deposit(value)` начинает процесс ввода определения в открытый класс. Интерпретатор Ruby не интересует, имеется ли определение данного метода. Если оно уже есть, то он просто заменяет его.

Переопределению исходного текста в Ruby присущ следующий недостаток: новый метод заменяет старый метод до конца программы, что может доставить неприятности, если забыть о том, что конкретный метод переопределен в предыдущем тесте.

Переопределение исходного текста можно осуществить также в C и C++, используя препроцессор. Соответствующий пример приведен при рассмотрении *шва предварительной обработки* в главе 4.

Процедура

Для переопределения исходного текста выполните следующую процедуру.

1. Выявите класс с определениями, которые требуется заменить.
2. Введите оператор `require` с именем модуля, содержащего данный класс, в начале тестового исходного файла.
3. Предоставьте альтернативные определения в начале тестового исходного файла для каждого метода, который вам требуется заменить.

Приложение

Реорганизация кода

Реорганизация кода является основным способом его улучшения. Каноническим первоисточником по реорганизации кода служит упоминавшаяся ранее книга Мартина Фаулера, *Рефакторинг. Улучшение существующего кода*. Отсылаю вас к этой книге за дополнительными сведениями о реорганизации кода, которую вы можете выполнить, разместив в нем тесты по местам.

Извлечение метода

Среди всех видов реорганизации кода *извлечение метода* является, вероятно, самым полезным. В основу извлечения метода положен принцип систематического разделения крупных методов на более мелкие. Это позволяет нам сделать код более понятным. Кроме того, мы можем неоднократно использовать фрагменты кода, избегая дублирования логики в других частях системы.

В плохо сопровождаемых базах кода методы обычно проявляют тенденцию к укрупнению. Разработчики вводят логику в существующие методы, а после эти методы постепенно разрастаются. В итоге методы могут выполнять две или три совершенно разные функции при их вызове. А в особо тяжелых случаях они могут выполнять десятки, если не сотни функций. К качестве средства борьбы с подобными патологическими случаями служит извлечение метода.

Для того чтобы извлечь метод, вам нужно, прежде всего, иметь соответствующий ряд тестов. Если у вас имеются тесты, тщательно испытывающие крупный метод, то вы можете извлечь из него методы, придерживаясь следующей процедуры.

1. Выявите код, который требуется извлечь, и закомментируйте его.
2. Придумайте имя для нового метода и создайте его в виде пустого метода.
3. Поместите вызов нового метода в старом методе.
4. Скопируйте код, который требуется извлечь, в новый метод.
5. Сделайте *упор на компилятор*, чтобы выявить те параметры, которые требуется передать, а также те значения, которые требуется возвратить.
6. Откорректируйте определение метода применительно к передаваемым параметрам и возвращаемым значениям, если таковые имеются.
7. Выполните свои тесты.
8. Удалите закомментированный код.

Рассмотрим следующий простой пример кода, написанного на языке Java.

```
public class Reservation
{
    public int calculateHandlingFee(int amount) {
```

```
        int result = 0;

        if (amount < 100) {
            result += getBaseFee(amount);
        }
        else {
            result += (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
       }
       return result;
    }
    ...
}
```

Логика оператора `else` проводит расчет вознаграждения по резервированию премий. Нам же нужно использовать эту логику в другом месте системы. Вместо того чтобы дублировать код, мы можем извлечь его из данного места и затем использовать в другом месте.

Ниже приведен код, полученный в результате выполнения первого этапа.

```
public class Reservation
{
    public int calculateHandlingFee(int amount) {
        int result = 0;

        if (amount < 100) {
            result += getBaseFee(amount);
        }
        else {
            // result += (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
        }
        return result;
    }
    ...
}
```

Нам требуется вызвать новый метод `getPremiumFee` (получить премиальное вознаграждение). Для этого мы введем в код новый метод и его вызов.

```
public class Reservation
{
    public int calculateHandlingFee(int amount) {
        int result = 0;

        if (amount < 100) {
            result += getBaseFee(amount);
        }
        else {
            // result += (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
            result += getPremiumFee();
       }
       return result;
    }

    int getPremiumFee() {
```

```
    }
    ...
}
```

Далее мы копируем старый код в новый метод и проверяем, компилируется он или нет.

```
public class Reservation
{
    public int calculateHandlingFee(int amount) {
        int result = 0;

        if (amount < 100) {
            result += getBaseFee(amount);
        }
        else {
            // result += (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
           result += getPremiumFee();
        }
        return result;
    }

    int getPremiumFee() {
        result += (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
    }
    ...
}
```

Этот код не компилируется. В нем используются переменные `result` (итог) и `amount` (сумма), которые не объявлены. А поскольку мы рассчитываем итог лишь частично, можно возвратить только то, что действительно рассчитываем. Кроме того, мы можем получить нужную сумму, если сделаем ее параметром метода и добавим ее к его вызову.

```
public class Reservation
{
    public int calculateHandlingFee(int amount) {
        int result = 0;

        if (amount < 100) {
            result += getBaseFee(amount);
        }
        else {
            // result += (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
            result += getPremiumFee(amount);
        }

        return result;
    }

    int getPremiumFee(int amount) {
        return (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
    }
```

```
    ...
}
```

Теперь мы можем выполнить наши тесты, чтобы убедиться в их работоспособности. Если они работоспособны, то можно вернуться назад и избавиться от закомментированного кода.

```
public class Reservation
{
    public int calculateHandlingFee(int amount) {
        int result = 0;
        if (amount < 100) {
            result += getBaseFee(amount);
        }
        else {
            result += getPremiumFee(amount);
        }
        return result;
    }

    int getPremiumFee(int amount) {
        return (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
    }
    ...
}
```

Я лично предпочитаю закомментировать код, который я собираюсь извлекать, хотя это и не обязательно. Ведь если я совершу ошибку и тест не пройдет, то смогу легко вернуться к прежнему коду, добиться того, чтобы тест проходил, а затем сделать еще одну попытку.

В рассмотренном здесь примере продемонстрирован лишь один из вариантов извлечения метода. Имея в своем распоряжении тесты, вы можете относительно просто и безопасно выполнять операцию по извлечению метода. А если у вас имеется инструментальное средство реорганизации кода, то такая операция окажется еще более безопасной. В этом случае вам нужно лишь выделить требуемую часть метода и выбрать соответствующую команду из меню. Инструментальное средство реорганизации кода само проверит, может ли выделенный код извлекаться в виде метода, и подскажет вам имя для нового метода.

Извлечение метода является основным способом, применяемым в работе с унаследованным кодом. С его помощью можно извлечь дублированный код, разделить ответственность и разбить длинные методы на мелкие части.

Словарь специальных терминов

Блочный тест. Тест, который выполняется менее чем за 1/10 секунды и достаточно мал, чтобы локализовать ошибки, если он не проходит.

Имитирующий объект. Фиктивный объект, который подтверждает условия внутренним образом.

Компоновочный шов. Место, где можно изменить поведение, прикомпоновав библиотеку. В транслируемых языках программирования можно заменить одни выходные библиотеки, динамически компонуемые библиотеки (DLL), сборки и файлы JAR на другие, чтобы избавиться от зависимостей или распознать определенное условие, которое может возникнуть при тестировании.

Объектный шов. Место, где можно изменить поведение с помощью замены одного объекта другим. В объектно-ориентированных языках программирования такая замена обычно делается за счет подклассификации класса в выходном коде и переопределения различных методов из этого класса.

Программирование по разности. Способ использования наследования для ввода новых свойств в объектно-ориентированные системы. Нередко служит для быстрого внедрения нового свойства в систему. Тесты, которые пишутся для стимулирования ввода нового свойства, можно использовать для последующей реорганизации кода для его улучшения.

Разработка посредством тестирования (TDD). Процесс разработки, заключающийся в последовательном написании контрольных примеров и удовлетворении их условий. В ходе этого процесса код реорганизуется таким образом, что сделать его как можно более простым. Код, разрабатываемый посредством тестирования, покрывается тестами по умолчанию.

Свободная функция. Функция, которая не является составной частью какого-либо класса. В C и других процедурных языках программирования подобные функции так и называются свободными. А в C++ они называются некомпонентными функциями, т.е. функциями, не являющимися членами классов. В Java и C# свободные функции отсутствуют.

Связующее число. Число значений, которые передаются методу и возвращаются методом при его вызове. Если ни одно из значений не возвращается, то связующее число обозначает лишь число передаваемых параметров. Связующее число очень полезно рассчитывать для мелких методов, которые требуется извлечь без тестирования.

Средства тестирования. Программные средства, разрешающие блочное тестирование.

Тестирующий подкласс. Подкласс, созданный для доступа к классу для его тестирования.

Точка изменения. Место в коде, где требуется сделать изменения.

Точка пересечения. Место, где можно написать тест для распознавания определенного условия в отдельной части программного обеспечения.

Точка сужения. Место сужения в эскизе воздействий, указывающее на идеальное место для тестирования совокупности свойств.

Фиктивный объект. Объект, который воплощает другой объект, взаимодействующий с отдельным классом во время тестирования.

Характеристический тест. Тест, написанный для документирования текущего поведения отдельной части программного обеспечения и сохранения этого поведения при изменении кода.

Шов. Место, где можно изменить поведение в системе программного обеспечения, не прибегая к правке кода в этом месте. Например, вызов полиморфной функции из объекта

является швом, поскольку можно выполнить подклассификацию класса этого объекта и тем самым изменить его поведение.

Эскиз воздействий. Небольшой набросок от руки, обозначающий те переменные и возвращаемые методами значения, на которые могут оказать воздействие изменения в программном обеспечении. Эскизы воздействий приносят пользу при выборе подходящего места для написания тестов.

Эскиз свойств. Небольшой набросок от руки, обозначающий использование одними методам класса других методов и переменных экземпляров. Эскизы свойств приносят пользу при выборе подходящего способа разделения крупного класса на части.

Предметный указатель

О

П

Р

ШАБЛОНЫ ТЕСТИРОВАНИЯ XUNIT
РЕФАКТОРИНГ КОДА ТЕСТОВ

Джерард Месарош

www.williamspublishing.com

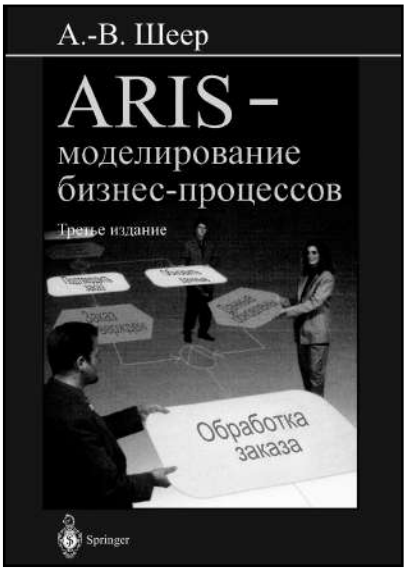
В данной книге показано, как применять принципы разработки программного обеспечения, в частности шаблоны проектирования, инкапсуляцию, исключение повторений и описательные имена, к написанию кода тестов. Книга состоит из трех частей.
В первой части приводятся теоретические основы методов разработки тестов, описываются концепции шаблонов и “запахов” тестов (признаков существующей проблемы). Во второй и третьей частях книги приводится каталог шаблонов проектирования тестов, “запахов” и других средств обеспечения большей прозрачности кода тестов. Кроме этого, в третьей части книги сделана попытка обобщить и привести к единому знаменателю терминологию тестовых двойников и подставных объектов, а также рассмотрены некоторые принципы их применения при проектировании как тестов, так и самого программного обеспечения.
Книга ориентирована на разработчиков программного обеспечения, практикующих гибкие процессы разработки.

ISBN 978-5-8459-1448-4

в продаже

ARIS —
МОДЕЛИРОВАНИЕ БИЗНЕС-ПРОЦЕССОВ
ТРЕТЬЕ ИЗДАНИЕ

А.-В. Шеер



www.williamspublishing.com

Система ARIS занимает лидирующее положение на рынке средств анализа и моделирования бизнес-процессов. В данной книге рассматривается широкий круг методов моделирования в рамках единой концепции. Соответствующие метамодели составляют информационную модель ARIS.
Особое внимание уделяется применению моделей ARIS для создания бизнес-приложений. Обсуждение концепций дополняется прикладными главами, посвященными внедрению стандартных бизнес-приложений с использованием моделей ARIS, применению программной системы ARIS Framework для разработки приложений, а также созданию объектно-ориентированных систем с использованием универсального языка моделирования UML.
Потенциальные читатели этой книги — это менеджеры предприятий, разработчики организационных и информационных систем.

ISBN 978-5-8459-1449-1 **в продаже**

C# 2008 И ПЛАТФОРМА .NET 3.5
ДЛЯ ПРОФЕССИОНАЛОВ

Кристиан Нейгел,
Билл Ивьен,
Джей Глинн,
Карли Уотсон,
Морган Скиннер

www.dialektika.com

В книге подробно рассматриваются такие вопросы, как основы языка программирования C#, организация среды .NET, работа с данными, написание Windows- и Web-приложений, взаимодействие через сеть, создание Web-служб и многое другое. Немалое внимание уделено проблемам безопасности и сопровождения кода. Тщательно подобранный материал позволит без труда разобраться с тонкостями использования Windows Forms и построения Web-страниц. Читатели ознакомятся с работой в Visual Studio 2008, а также с применением таких технологий, как ADO.NET, ASP.NET, GDI+, Windows Presentation Foundation, Windows Communication Foundation, Windows Workflow Foundation, ASP.NET AJAX и LINQ. Прилагаемый к книге компакт-диск содержит исходные коды всех примеров, что существенно упростит освоение материала.
Книга рассчитана на программистов разной квалификации, а также будет полезна для студентов и преподавателей дисциплин, связанных с программированием и разработкой для .NET.

ISBN 978-5-8459-1458-3

в продаже

СОВРЕМЕННОЕ ПРОЕКТИРОВАНИЕ НА C++

Андрей Александреску

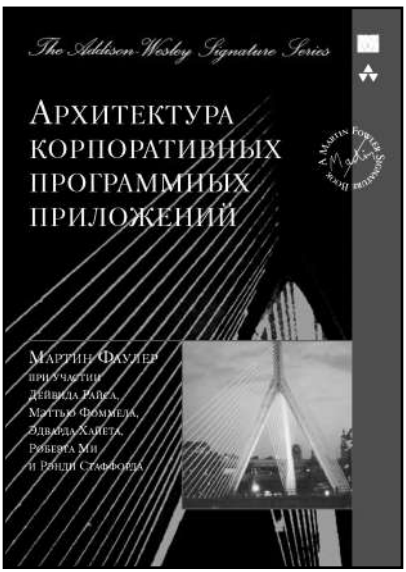
www.williamspublishing.com

В книге изложена новая технология программирования, представляющая собой сплав обобщенного программирования, метапрограммирования шаблонов и объектно-ориентированного программирования на C++. Настраиваемые компоненты, созданные автором, высоко подняли уровень абстракции, наделив язык C++ чертами языка спецификации проектирования, сохранив всю его мощь и выразительность.
В книге изложены способы реализации основных шаблонов проектирования. Разработанные компоненты воплощены в библиотеке Loki, которую можно загрузить с Web-страницы автора.
Книга предназначена для опытных программистов на C++.

ISBN 978-5-8459-0351-8 **в продаже**

АРХИТЕКТУРА КОРПОРАТИВНЫХ ПРОГРАММНЫХ ПРИЛОЖЕНИЙ

Мартин Фаулер



www.williamspublishing.com

Книга дает ответы на трудные вопросы, с которыми приходится сталкиваться всем разработчикам корпоративных систем. Автор, известный специалист в области объектно-ориентированного программирования, заметил, что с развитием технологий базовые принципы проектирования и решения общих проблем остаются неизменными, и выделил более 40 наиболее употребительных подходов, оформив их в виде типовых решений. Результат перед вами — незаменимое руководство по архитектуре программных систем для любой корпоративной платформы. Это своеобразное учебное пособие поможет вам не только усвоить информацию, но и передать полученные знания окружающим значительно быстрее и эффективнее, чем это удавалось автору. Книга предназначена для программистов, проектировщиков и архитекторов, которые занимаются созданием корпоративных приложений и стремятся повысить качество принимаемых стратегических решений.

ISBN 978-5-8459-0579-6

в продаже

АНАЛИЗ И ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ С ПОМОЩЬЮ UML 2.0

ТРЕТЬЕ ИЗДАНИЕ

Лешек А. Мацяшек

www.williamspublishing.com

Книга представляет собой новое издание популярного учебника Лешека Мацяшека по объектно-ориентированной разработке информационных систем. В книге подробно описаны методы анализа и проектирования промышленных информационных систем с использованием языка UML. Отличительной особенностью книги является обилие учебных примеров, упражнений, контрольных вопросов и многовариантных тестов. Уникальный характер книги обусловлен оптимальным сочетанием практического опыта и теоретических представлений. Книга будет полезна системным аналитикам и архитекторам, программистам, преподавателям и студентам высших учебных заведений, а также все специалистам по информационным технологиям.

ISBN 978-5-8459-1430-9 **в продаже**

ШАБЛОНЫ РЕАЛИЗАЦИИ КОРПОРАТИВНЫХ ПРИЛОЖЕНИЙ

Кент Бек

www.dialektika.com

Один из самых креативных и признанных лидеров в индустрии программного обеспечения Кент Бек собрал 77 шаблонов, предназначенных для выполнения ежедневных программистских задач и написания более читаемого кода. Эта новая коллекция шаблонов предназначена для реализации многих аспектов разработки, включая классы, состояние, поведение, методы, коллекции, инфраструктуры и т.д. Автор использует диаграммы, истории, примеры и эссе для того, чтобы увлечь читателя по ходу освещения шаблонов. Вы обнаружите проверенные решения для управления всем, от именования переменных до проверки исключений.

Эта книга предназначена для программистов всех уровней подготовки, особенно для тех, кто применяет в своей практике шаблоны проектирования и методы быстрой разработки. Книга также окажется неоценимым ресурсом для команд разработчиков, ищущих более эффективные методы совместной работы и построения более управляемого ПО.

ISBN 978-5-8459-1406-4

в продаже