БИБЛИОТЕКА ПРОГРАММИСТА

Сэнди Метц

Ruby

Объектно-ориентированное проектирование

ПИТЕР

Practical Object-Oriented Design in Ruby

An Agile Primer

Sandi Metz

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

БИБЛИОТЕКА ПРОГРАММИСТА

ББК 32.988.02-018
УДК 004.738.5
М54

Сэнди Метц
М54 Ruby. Объектно-ориентированное проектирование. — СПб.: Питер, 2017. — 304 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-02437-2

Мировой бестселлер по программированию на языке Ruby. Книга уже стала классической — с ювелирной точностью она описывает огранку профессионального кода на Ruby. Внимательно изучив это незаменимое руководство, вы сможете:

- Понять, как писать на Ruby качественный код в духе ООП
- Решать, что должно входить в состав класса Ruby
- Не допускать тесной связи между объектами в тех случаях, когда требуется разграничить функциональность
- Определять гибкие интерфейсы между объектами
- Освоить утиную типизацию
- Эффективно задействовать наследование, композицию и полиморфизм
- Разрабатывать экономные тесты
- Доводить до совершенства любой legacy-код Ruby

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-0321721334 англ.
ISBN 978-5-496-02437-2

Краткое содержание

Оглавление

Эми, которая все читает первой.

Предисловие

Одна из прописных истин гласит, что в процессе разработки программ увеличивается объем кода и изменяются требования к создаваемой системе. Кроме того, практически во всех случаях возможность сопровождения кода в течение всего срока его существования более важна, чем его оптимизация.

В сравнении с другими технологиями программирования преимущество объектно-ориентированного проектирования заключается в упрощении сопровождения и развития кода. Но как новички-программисты могут узнать все эти секреты? Ведь многие никогда не обучались написанию высококачественного объектно-ориентированного кода, а собирали информацию по крупицам, осваивая опыт коллег и черпая знания из устаревших книг и Интернета. Если даже они изучали основы объектно-ориентированного программирования в учебных заведениях, то явно с использованием таких языков, как Java или C++. (Везунчиков обучали на Smalltalk!)

В книге «Ruby. Объектно-ориентированное проектирование» рассматриваются основы объектно-ориентированного проектирования с использованием языка Ruby, а это значит, что она готова вести новичков в программировании на Ruby и Rails к вершинам мастерства (то есть к получению статуса настоящих программистов).

Ruby, как и Smalltalk, является полноценным объектно-ориентированным языком. В нем все, даже элементарные конструкции данных (такие как строки и числа), представлено в виде объектов с определенным поведением. При написании приложений на Ruby вы программируете свои собственные объекты, в каждом из которых инкапсулируется некое состояние и определяется поведение. Если у вас еще нет опыта объектно-ориентированного программирования,

то сразу не получится приступить к работе. Эта книга проведет вас по всему пути — от основных вопросов о том, что нужно помещать в класс, базовых понятий (таких как принцип единственной обязанности) и выбора возможных компромиссов между наследованием и композицией до вопросов о порядке тестирования объектов в условиях их изоляции.

Но лучшее, что есть в книге, — это мнение самой Сэнди. У нее богатейший опыт, я считаю, что она проделала грандиозную работу, изложив свое понимание вопроса в письменном виде. Я с удовольствием приветствую ее в качестве нового автора серии книг для профессионалов Ruby.

Оби Фернандес (Obie Fernandez),
редактор Professional Ruby Series,
издательство Addison-Wesley

Введение от научных редакторов

Мы старались вложить в работу над книгой все свои силы, и хотелось бы, чтобы она оказалась вам полезной. И, как бы то ни было, от процесса создания книги мы получили большое удовольствие.

Особенно повезло тем из нас, кто работал над написанием фрагментов программ. Эта работа была особенно интересной, поскольку для достижения намеченных целей нам приходилось проявлять недюжинные творческие способности. Нас радовал не только сам процесс написания кода, но и осознание его востребованности. Мы делали нечто значимое. Мы понимали, что создаем программы, соответствующие реалиям сегодняшних дней, и по праву гордимся своими достижениями.

Такие чувства знакомы всем программистам — как восторженным новичкам, так и опытным специалистам — независимо от того, над чем они работают: над легким новым интернет-приложением или над крупным проектом. Мы готовы трудиться не покладая рук. Нам хочется, чтобы наша работа была востребованной. А еще нам нужно, чтобы сам процесс доставлял удовольствие. Поэтому особое беспокойство вызывают неудачи при разработке программ. Низкокачественные программы мешают достижению намеченной цели и не приносят радости от работы. То, что раньше удавалось выполнить быстро, теперь делается медленно. Там, где раньше царило спокойствие, теперь возникает разочарование.

Это разочарование — следствие слишком больших затрат на достижение результатов. Наш «внутренний счетчик» постоянно тикает: мы всегда сравниваем

объем полученных результатов с объемом затраченных сил. Когда трудозатраты превышают ценность работы, кажется, что напрасно приложено столько усилий. Если программирование приносит удовлетворение, значит, у нас есть возможность быть полезными. Когда же оно в тягость, это сигнал, что мы что-то делаем не так.

Книга посвящена проектированию объектно-ориентированных программ. Это не учебник, а рассказ программиста о том, как следует создавать программный код. Книга научит вас, как скомпоновать программное обеспечение, чтобы его высокая продуктивность не снижалась ни через месяц, ни через год. В ней показано, как создаются приложения, которые могут пользоваться успехом сегодня и адаптироваться к требованиям будущего. Это позволит вам более эффективно работать и сократить затраты на сопровождение приложений на протяжении всего их жизненного цикла.

Автор рассчитывает на вашу готовность трудиться и предоставляет вам для этого необходимый инструментарий. Книга имеет чисто практическую направленность и по сути является пособием, в котором рассказывается, как писать код легко и с удовольствием.

Кому эта книга будет полезна

Предполагается, что читатели имеют хотя бы начальный опыт создания объектно-ориентированных программ. При этом неважно, считаете ли вы свой опыт успешным, главное, чтобы у вас была практика работы с каким-либо объектно-ориентированным языком. В главе 1 дается краткий обзор объектно-ориентированного программирования и приводятся общие понятия.

Если вы еще не занимались объектно-ориентированным проектированием, но хотите научиться, то перед тем, как приступить к чтению, нужно пройти хотя бы начальный курс программирования. Примеры, приведенные в книге, предполагают, что вам уже приходилось сталкиваться с чем-то подобным. Опытные программисты могут пропустить этот этап, но большинству читателей мы рекомендуем получить начальный опыт написания объектно-ориентированного кода.

Для обучения объектно-ориентированному проектированию в этой книге используется язык Ruby, но для усвоения изложенных концепций знать Ruby

совсем не обязательно. В издании приводятся примеры кода, и они предельно просты. При наличии опыта программирования на любом объектно-ориентированном языке разобраться с Ruby будет нетрудно.

Если вы привыкли работать с объектно-ориентированными языками со статической типизацией, например с Java или C++, значит, у вас уже есть необходимый багаж знаний, чтобы извлечь пользу из чтения данной книги. Ruby является языком с динамической типизацией, что упрощает синтаксис примеров и позволяет отразить самую суть проектирования. Каждое понятие из книги может быть напрямую переведено в код объектно-ориентированного языка со статической типизацией.

Как следует читать книгу

В главе 1 описываются общие задачи объектно-ориентированного проектирования и обстоятельства, требующие его применения, дается краткий обзор объектно-ориентированного программирования (ООП). Эта глава как бы стоит особняком. Ее можно прочитать первой, последней или пропустить, но если вы в настоящий момент столкнулись с плохо спроектированным приложением, то обязательно прочитайте эту главу.

Если у вас есть опыт написания объектно-ориентированных приложений и вы не хотите терять время на повторение той информации, которая вам уже и так известна, можете смело начинать с главы 2. Если столкнетесь с непонятным термином, вернитесь назад и бегло просмотрите раздел «Краткое введение в объектно-ориентированное программирование» главы 1, где приводятся общие понятия ООП, используемые в данной книге.

В главах 2–9 объясняются премудрости объектно-ориентированного проектирования.

В главе 2 «Проектирование классов с единственной обязанностью» описывается, как решить, что должно принадлежать отдельно взятому классу.

В главе 3 «Управление зависимостями» показано, как объекты сотрудничают друг с другом. В главах 2 и 3 основное внимание уделяется объектам, а не сообщениям.

В главе 4 «Создание гибких интерфейсов» акценты смещаются с проектирования, где все построено на базе объектов, к проектированию, основанному на обмене сообщениями. Эта глава посвящена определению интерфейсов; основное внимание в ней уделяется тому, как объекты общаются друг с другом.

Глава 5 «Снижение затрат за счет неявной типизации» посвящена неявной («утиной») типизации. В главе рассматривается идея о том, что объекты различных классов могут играть общие роли.

Глава 6 «Получение поведения через наследование» учит применять технологии классического наследования.

В главе 7 «Разделение ролевого поведения с помощью модулей» технологии классического наследования применяются для создания ролей с неявной типизацией.

Глава 8 «Объединение объектов путем составления композиции» раскрывает приемы создания объектов посредством композиций и предоставляет руководство по выбору между композицией, наследованием и совместным использованием ролей с применением неявной типизации.

Глава 9 «Проектирование экономически эффективных тестов» посвящена в основном проектированию тестов; при этом в качестве иллюстраций используется код из предыдущих глав книги.

Каждая из глав основана на понятиях, раскрываемых в предыдущих главах, поэтому читать их лучше по порядку.

В чем польза этого издания

Читателям с разными уровнями подготовки книга окажется по-разному полезна. Тем, кто уже знаком с объектно-ориентированным проектированием, будет о чем поразмышлять; возможно, они по-новому посмотрят на уже привычные вещи и, вполне вероятно, с чем-то будут не согласны. Поскольку в объектно-ориентированном проектировании нет истины в последней инстанции, оспаривание принципов (и спор с автором данной книги) только улучшит общее понимание предмета. В конце концов, именно вы должны оценивать собственные конструкции кода, проводить эксперименты и делать правильный выбор.

Хотя издание должно представлять интерес для читателей с различными уровнями подготовки, оно выпускалось с прицелом на новичков. Если вы новичок, то эта часть введения — именно для вас. Усвойте простую истину: объектно-ориентированное проектирование не магия. Это всего лишь еще не освоенная вами сфера. И раз вы читаете эти строки, это говорит о вашей заинтересован-

ности, а тяга к знаниям — единственное условие получения пользы от прочтения данной книги.

В главах 2–9 объясняются принципы объектно-ориентированного проектирования и приводятся четкие правила программирования, значение которых для новичка будет отличаться от значения для опытных программистов. Новичкам нужно следовать этим правилам, не сомневаясь в их важности. Безоговорочное выполнение правил поможет избежать крупных неприятностей, пока не будет накоплен опыт, позволяющий принимать собственные решения. К тому времени, когда правила станут вас раздражать, у вас уже будет достаточно опыта для составления своих правил, и вы начнете двигаться по карьерной лестнице в качестве проектировщика.

Благодарности

Издание этой книги состоялось благодаря усилиям и поддержке многих людей.

На долгом пути ее написания самые первые отзывы о каждой главе приходили от Лори Эванс (Lori Evans) и Ти Джея Станкуса (TJ Stankus). Они живут в Дареме, Северная Каролина, поэтому не могли уклониться от решения моих проблем, но данный факт ничуть не умаляет моей признательности за оказанную помощь.

После того как уже была готова половина книги и стало ясно, что ее написание займет примерно в два раза больше времени, чем первоначально предполагалось, Майк Далессио (Mike Dalessio) и Грегори Браун (Gregory Brown) вычитали черновики и дали бесценные отзывы. Их помощь и энтузиазм помогли поддержать проект в трудные дни.

Ближе к завершению работы первыми читателями книги стали Стив Клабник (Steve Klabnik), Деси Макадам (Desi McAdam) и Сет Вакс (Seth Wax). Их впечатления и предложения позволили внести некоторые полезные изменения.

Чуть позже Катрина Оуэн (Katrina Owen), Авди Гримм (Avdi Grimm) и Ребекка Вирфс-Брок (Rebecca Wirfs-Brock) тщательно проштудировали рукопись — благодаря их содержательным отзывам книга стала намного лучше. С Катриной, Авди и Ребеккой я прежде не была знакома, но благодарна им за участие в проекте и покорена их великодушием. Если книга окажется вам полезной, то поблагодарите их при встрече.

Я также весьма признательна Gotham Ruby Group и всем, кто одобрил дискуссии по вопросам проектирования, затеянные мною на конференции GoRuCo в 2009 и 2011 годах. Организаторы GoRuCo рискнули сделать ставку на неиз-

вестное и позволили мне развернуть дискуссии, в ходе которых я смогла выразить свои идеи; именно с этого и началась работа над книгой. Иэн Макфарлэнд (Ian McFarland) и Брайан Форд (Brian Ford) обратили внимание на эти дискуссии, тут же проявили энтузиазм в отношении данного проекта и убедили меня в возможности его осуществления.

Написанию книги во многом поспособствовал Майкл Тёрстон (Michael Thurston) из Pearson Education, который заряжал меня своим спокойствием и организованностью, когда я тонула в море расходящихся в разных направлениях мыслей. Он терпеливо настаивал на том, что книга должно быть лаконично написана и понятно структурирована. Я считаю, что его усилия не пропали даром (в этом вы сможете убедиться сами в процессе чтения).

Хочу поблагодарить Дебру Уильямс Коли (Debra Williams Cauley) — редактора издательства Addison-Wesley, которая подслушала случайные споры в кулуарах первой конференции по Ruby on Rails в Чикаго в 2006 году и подбила меня на работу, которая в конечном итоге вылилась в написание этой книги. Как я ни отбивалась, Дебра не принимала никаких возражений, умело приводя аргумент за аргументом, пока не нашла самый убедительный. Думаю, она очень предана делу.

Я в долгу перед всем сообществом создателей объектно-ориентированных проектов. Идеи, изложенные в этом издании, не плод моего воображения — я всего лишь их проводник, получивший поддержку истинных гигантов мысли. Хотя заслуги в генерировании этих идей принадлежат другим людям, все недостатки при их передаче читателю целиком на моей совести.

Наконец, эта книга обязана своим выходом в свет моему партнеру Эми Гермут (Amy Germuth). Раньше я не представляла, как это — писать книгу, но Эми вселила в меня веру в осуществимость задуманного. Книга, которую вы держите в руках, стала наградой за ее безграничное терпение и поддержку.

Спасибо всем и каждому по отдельности!

Об авторе

Сэнди Метц 30 лет работает над такими проектами, качество которых позволяет им расширяться и совершенствоваться. Она разработчик структур программного обеспечения в Университете Дьюка, где ее команда решает реальные задачи для клиентов, имеющих крупные объектно-ориентированные приложения, сопровождение и развитие которых ведется на протяжении 15 и более лет. Сэнди стремится создавать полезные, нетривиальные программы исключительно практичными способами. Книга «Ruby. Объектно-ориентированное проектирование» — продукт обобщения множества разработок и итог многолетних дебатов на тему объектно-ориентированного проектирования.

Сэнди живет в Дареме, Северная Каролина, имеет опыт выступления на Ruby Nation, несколько раз участвовала в конференции пользователей Gotham Ruby User's Conference.

Глава 1
Объектно-ориентированное проектирование

Наш мир состоит из процедур. Время течет, что-то постоянно происходит. С утра вы можете пройти процедуру пробуждения, чистки зубов, приготовления кофе, одевания. Эти действия можно смоделировать в процедурных приложениях. Поскольку вам известен порядок событий, можно написать код для каждого действия, а затем связать их вместе — одно за другим.

Мир также имеет объектно-ориентированную природу. К объектам, с которыми вы взаимодействуете, можно отнести, например, жену и кота, старую машину и кучу велосипедных запчастей в гараже, ваше сердцебиение и график тренировок. Каждый из этих объектов обладает характерным только для него поведением. И хотя некоторые варианты взаимодействия объектов могут быть предсказуемы, может случиться так, что жена неожиданно наступит на кота, вызвав у всех учащенное сердцебиение и побудив вас пересмотреть график своих тренировок.

В мире объектов новые варианты поведения возникают вполне естественным образом. Вам не приходится создавать конкретный код для процедуры `spouse_steps_on_cat`, в ходе которой жена наступает на кота, — вам достаточно шагающего объекта-жены и объекта-кота, которому не нравится, когда на него

наступают. Поместите оба этих объекта в комнату — и станут появляться неожиданные комбинации поведения.

Книга посвящена проектированию объектно-ориентированных программ, в ней автор пытается взглянуть на мир как на серию спонтанных взаимодействий объектов. Объектно-ориентированное проектирование требует переосмысления мира: нужно отойти от его представления как коллекции предопределенных процедур и перейти к его моделированию в виде серии сообщений между объектами.

Просчеты в ООП могут казаться просчетами в самих программах, но на самом деле это просчеты в представлении окружающей обстановки. Первое требование при изучении способов ООП — погружение в суть объектов. Как только будет выработан объектно-ориентированный взгляд на окружающий мир, все остальное приложится само собой.

Эта книга проведет вас через процесс погружения. В начале первой главы описаны общие вопросы ООП. От рассмотрения доводов в пользу проектирования мы перейдем к изучению того, когда его следует проводить и оценивать. В конце главы дан краткий обзор ОПП с определением понятий, используемых в книге.

Хвала проектированию

Программное обеспечение не создается по мановению волшебной палочки. Суть в том, что приложение, будь то обычная игра или программа по управлению лучевой терапией, должно иметь определенную цель. Если бы самым экономически эффективным способом создания работоспособных приложений было программирование на пределе всех сил и возможностей разработчиков, то тем бы пришлось постоянно стоически преодолевать моральные трудности или же подыскивать себе другую работу.

К счастью, вам не придется выбирать между удовольствием, получаемым от работы, и ее продуктивностью. Технологии программирования, позволяющие писать код с удовольствием, аналогичны технологиям, позволяющим создавать приложения наиболее эффективным образом. Так что технологии ООП решают как моральные, так и технические дилеммы программирования. Четкое следование этим технологиям позволит создавать экономически

эффективные программы, к тому же над их кодом будет весьма приятно работать.

Проблемы, решаемые с помощью проектирования

Представьте, что создается новое приложение. И в отношении него уже выработан полный и выверенный набор требований. Представьте себе еще один момент: после того как приложение будет создано, в него никогда не придется вносить изменения. В этом случае проектирование не требуется. Подобно жонглеру, который крутит тарелки на гибких прутьях в мире без силы трения и гравитации, вы можете запрограммировать приложение на движение, а затем стоять в сторонке, наблюдая, как оно будет работать. Как бы ни трясло окружающий мир, «тарелки» вашего кода не снизят темпа вращения, балансируя в процессе каждого оборота, но никогда не падая на землю. Пока что-либо не изменится.

К сожалению, что-то *должно* измениться. Изменения происходят всегда. Клиенты не знают, чего хотят, нечетко выражают свои мысли. Вы порой не понимаете, что им нужно, зато знаете, как можно улучшить код. Даже самые совершенные приложения нестабильны. Приложение имело огромный успех, а теперь всем захотелось большего. Таким образом, изменения неизбежны, без них обойтись не удастся.

Изменения в требованиях являются в программировании эквивалентом трения и гравитации. Это силы, совершенно неожиданно влияющие на самые продуманные планы.

Приложения, легко поддающиеся изменениям, приятно создавать и комфортно расширять. Они обладают гибкостью и приспособляемостью. А приложения, сопротивляющиеся изменениям, совершенно иные: любое нововведение дается с большими затратами, и каждое из них приводит к подорожанию следующего изменения. Вряд ли найдется такое слабо поддающееся изменениям приложение, с которым было бы приятно работать. Худшие из них постепенно превращаются в чей-то кошмар с несчастным программистом в главной роли, который должен метаться от одной «тарелки» к другой в попытке не разбить их.

Почему изменения так нелегко даются

Объектно-ориентированные приложения составлены из взаимодействующих частей. Этими частями являются *объекты*, а взаимодействие осуществляется путем обмена *сообщениями*. Чтобы направить правильное сообщение в адрес нужного объекта, объект-отправитель сообщения должен знать, что представляет собой объект-получатель. Эти знания создают зависимости между двумя объектами, и они же мешают изменениям.

Объектно-ориентированное проектирование заключается в *управлении зависимостями*. Это набор методик создания программного кода, выстраивающих зависимости таким образом, чтобы объекты можно было изменять. При отсутствии проектирования неуправляемые зависимости порождают хаос, поскольку объекты слишком осведомлены друг о друге. Изменения, вносимые в один объект, заставляют изменять и те объекты, которые с ним сотрудничают, что, в свою очередь, заставляет вносить изменения в объекты, которые сотрудничают с этими объектами, *и так до бесконечности*. Казалось бы, незначительное усовершенствование может привести к настоящей цепной реакции, и в конечном счете код оставляют нетронутым.

Когда объекты слишком много знают, они многого требуют. Они слишком капризны, и им нужно, чтобы все было «именно так и никак иначе». Эти требования стесняют их работу. Объекты сопротивляются своему повторному использованию в различных контекстах, их очень трудно тестировать — и неизбежно возникает необходимость в их дублировании.

В небольшом приложении еще можно смириться с недостатками проектирования. Даже если все объекты взаимодействуют друг с другом, пока вы способны удерживать это в голове, у нас еще есть возможность усовершенствования приложения. Проблема, связанная со слабым проектированием *небольших* приложений, заключается в том, что в случае успеха они разрастаются до слабо спроектированных *больших* приложений. Они постепенно становятся топью, в которую вы боитесь вступить, поскольку в ней можно запросто утонуть. Простейшие правки могут вызвать каскад изменений по всему приложению, повсеместно нарушая код и требуя большого объема переделок. Тестирование попадает под перекрестный огонь и начинает казаться помехой, а не помощником.

Определение проектирования

Каждое приложение представляет собой набор кода, а систематизация этого кода называется *проектированием*. Два отдельных программиста могут решать одну и ту же задачу разными способами. Проектирование — не сборочная линия, где одинаково обученные работники создают одинаковые виджеты; это студия, где творцы-единомышленники ваяют пользовательские приложения. Проектирование — это искусство, суть которого заключается в систематизации кода.

Некоторые трудности проектирования связаны с тем, что у каждой задачи есть две составляющие. Вы должны не просто написать код для функции, которую планируете поставить сегодня, но сделать его поддающимся дальнейшим изменениям. Для любого периода времени, прошедшего с момента поставки бета-версии, стоимость изменений в конечном итоге превзойдет исходную стоимость приложения. Поскольку принципы проектирования связаны между собой и каждая проблема предполагает изменение сроков готовности проекта, проблемы проектирования могут иметь огромное количество возможных решений.

От вас требуется выработать общий взгляд на вещи (то есть вы должны сопоставить задачу, которую будет решать ваше приложение, со всеми издержками и выгодами проектировочных альтернатив), а затем разработать такую структуру кода, которая была бы экономически выгодна в настоящее время и продолжала быть таковой в будущем.

Может показаться, что прогнозирование не имеет ничего общего с программированием. Но это не так. Конечно, при проектировании не надо стараться предвидеть неизвестные требования и выбирать одно из них для применения в приложении. Программисты — не физики. В практическом проектировании не строятся прогнозы насчет того, что может случиться с вашим приложением, а просто допускается неизбежность перемен, а также их неожиданность. Дело не в выстраивании догадок, а в дополнительных возможностях, сохраняющих способность приспосабливаться к будущему. И дело не в выборе, а в свободе маневра.

Таким образом, основной задачей проектирования является снижение затрат на внесение изменений.

Инструменты проектирования

Проектирование — это не следование определенному набору правил, а путешествие по развилкам, где ранее сделанные выборы закрывают доступ к одним вариантам и открывают его к другим. При создании приложения приходится блуждать по лабиринту требований, а каждый поворот требует принятия решения, которое будет иметь последствия в будущем.

У скульптора под рукой резцы и напильники, а проектировщику объектно-ориентированных программ инструментами служат принципы и шаблоны.

Принципы проектирования

Акроним SOLID, введенный в обиход Майклом Физерсом (Michael Feathers) и популяризированный Робертом Мартином (Robert Martin), обозначает пять самых известных принципов объектно-ориентированного проектирования: единственной обязанности — **S**ingle Responsibility, открытости-закрытости — **O**pen-Closed, подстановки, предложенной Барбарой Лисков (Barabara Liskov), — **L**iskov Substitution, разделения интерфейса — **I**nterface Segregation, инверсии зависимостей — **D**ependency Inversion. К числу других принципов можно отнести DRY (Don't Repeat Yourself — «не повторяйтесь»), введенный Энди Хантом (Andy Hunt) и Дэйвом Томасом (Dave Thomas), и закон Деметры — Law of Demeter (LoD) из проекта «Деметра» Северо-Восточного университета (США).

Сами принципы будут рассматриваться по всей вкниге, а пока зададимся вопросом, откуда они взялись. Существуют ли реальные доказательства их значимости, или это просто чье-то мнение, которым можно пренебречь? И вообще, с какой стати их следует придерживаться?

Все эти принципы возникли в результате выборов, сделанных кем-то при написании кода. На ранних стадиях развития объектно-ориентированной технологии программисты подметили, что одни структуры кода упрощали им жизнь, а другие — усложняли. Исходя из этого опыта было выработано мнение о том, как создается качественный код.

Со временем подключились преподаватели с научными степенями, которые решили дать количественную оценку «совершенству». Такое рвение похвально. Если что-то можно подсчитать, то есть вычислить *количественные показатели* относительно нашего кода, и соотнести эти показатели с приложениями высокого или низкого качества (для которых также нужны объективные показатели), то нам удастся чаще создавать продукты с меньшей себестоимостью. Возмож-

ность оценки качества изменит ООП, переведя его из разряда бесконечно оспариваемых мнений в разряд поддающейся измерению науки.

Именно это и было сделано в 1990-х годах Чидамбером (Chidamber), Кемерером (Kemerer)[1] и Базили (Basili)[2]. Они взяли объектно-ориентированные приложения и попытались дать количественную оценку коду. Они придумали названия и систему измерений, описав общий размер классов, их вложенность друг в друга, глубину и ширину иерархии наследования, а также количество методов, вовлекаемых в результате отправки любого сообщения. Они подобрали структуры кода, которые, по их мнению, могли иметь значение, вывели формулы подсчета, а затем сопоставили получившиеся показатели с качеством имеющихся приложений. Их исследование показывает определенную взаимосвязь между этими технологиями и высококачественным кодом.

Хотя кажется, что эти исследования подтверждают принципы проектирования, любой опытный программист воспринимает их с оговоркой. В ходе этих исследований, проведенных на ранней стадии, были изучены весьма небольшие по объему приложения, созданные аспирантами, и лишь этого одного уже достаточно, чтобы настороженно относиться к сделанным выводам. Код в этих приложениях мог не давать общей картины, свойственной объектно-ориентированным приложениям реального мира.

Но оказалось, что опасения напрасны. В 2001 году Лэйнг (Laing) и Колеман (Coleman) изучили ряд приложений, созданных в NASA Goddard Space Flight Center, в попытке отыскать в них «способ производства более дешевых и высококачественных программных продуктов»[3]. Они изучили три приложения разного качества, в одном из которых было 1617 классов и более 500 000 строк кода. Их исследования подкрепили результаты более ранних исследований и еще раз подтвердили важность принципов проектирования.

Даже если вы никогда не слышали об этих исследованиях, можете быть уверены в достоверности их выводов. Принципы разумного проектирования — это факты, поддающиеся конкретным оценкам. Следуя им, вы повысите качество своего кода.

---

1 *Chidamber S. R., Kemerer C. F.* A metrics suite for object-oriented design // IEEE Trans. Softw. Eng, 1994. — P. 476–493.

2 Basili Technical Report // A Validation of Object-Oriented Design Metrics as Quality Indicators. — Univ. of Maryland, Dep. of Computer Science, College Park, M. D., 20742 USA. — April 1995.

3 *Laing V., Coleman C.* Principal Components of Orthogonal Object-Oriented Metrics. — 2001.

Шаблоны проектирования

Еще одним инструментом объектно-ориентированного проектирования являются *шаблоны*. Так называемая банда четырех (Gang of Four, Gof) — Эрих Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson) и Джон Влиссидис (Jon Vlissides) — написала в 1995 году новаторскую работу по шаблонам. Их книга *Design Patterns* описывает шаблоны как «простые и элегантные решения конкретных проблем в проектировании объектно-ориентированных программ», которые могут использоваться «для придания вашим собственным проектам большей гибкости, модульности, возможности повторного использования кода и доступности для понимания»[1].

Шаблоны проектирования обладают невероятным функционалом. Они имеются в любом наборе инструментов проектировщика. Каждый широко известный шаблон является практически идеальным решением с открытым исходным кодом. Однако популярность шаблонов привела к своеобразному злоупотреблению ими начинающими программистами, которые из лучших побуждений применяли вполне удачные шаблоны для решения совершенно не тех проблем. Неверное использование шаблона приводит к излишне усложненному и запутанному коду, но этот результат нельзя назвать недостатком самого шаблона — инструмент нужно применять по назначению.

Эта книга не о шаблонах, но она научит вас разбираться в них и покажет, как их правильно выбирать и использовать.

Процесс проектирования

Казалось бы, с открытием и распространением общих принципов и шаблонов все проблемы объектно-ориентированного проектирования были решены. Теперь, когда известны основные правила, какие могут быть трудности в разработке объектно-ориентированных программных продуктов?

Оказывается, трудности могут быть, и немалые. Если представлять себе программный продукт в виде мебели на заказ, то принципы и шаблоны похожи на деревообрабатывающие инструменты. А конечный результат, будь то красивый

[1] *Gamma E., Helm R., Johnson R., Vlissides J.* Design Patterns, Elements of Reusable Object-Oriented Software. — New York, N. Y.: Addison-Wesley Publishing Company, Inc, 1995.

комод или расшатанный стул, отражает опыт обращения программиста с инструментами проектирования.

Причины провалов. Программисты изначально имеют слабое представление о проекте. Но это не мешает им, поскольку работоспособные приложения можно создавать, не имея предварительных сведений о проекте. Это утверждение справедливо для любого объектно-ориентированного языка, но некоторые из них более восприимчивы к недостаточному предварительному проектированию, в том числе язык Ruby. Ruby отличается высокой степенью лояльности, разрешая практически каждому создавать сценарии для автоматизации повторяющихся задач, а такая амбициозная среда, как Ruby on Rails, делает доступной разработку веб-приложений для любого программиста. Синтаксис языка Ruby настолько легкий, что создание работоспособных приложений по плечу каждому специалисту с линейным мышлением. При этом в работе с Ruby могут преуспеть даже программисты, ничего не знающие об объектно-ориентированном проектировании.

И все же успешные, но *созданные без соответствующего проектирования* приложения несут в себе элементы саморазрушения: их нетрудно создать, однако постепенно становится невозможно изменить. Опыт программиста не позволяет предсказывать будущее. Как только программисты начинают реагировать на каждый запрос на изменение фразой «Да, я могу добавить эту функцию, *но она все разрушит*», ранее данные обещания относительно безболезненного развития приложения постепенно переходят в разряд невыполнимых, а оптимизм переходит в отчаяние.

Более опытные программисты сталкиваются с другими проблемами проектирования. Они знакомы с технологиями ООП, но еще не разбираются в особенностях их применения. Движимые лучшими побуждениями, эти программисты попадают в ловушку перепроектирования. В энтузиазме они невпопад применяют принципы и видят закономерности там, где их нет. Они возводят из кода сложные красивые замки, а затем чувствуют себя подавленными в окружении этих каменных стен. Таких программистов узнать нетрудно, поскольку они начинают реагировать на запрос на каждое изменение фразой «Нет, я не могу добавить это свойство, поскольку приложение *не было спроектировано для подобных целей*».

И наконец, объектно-ориентированные программы терпят фиаско в том случае, когда процесс проектирования оторван от процесса программирования. Проектирование является процессом постепенных открытий, зависящих от

обратной связи. Эта связь должна быть своевременной и идти по нарастающей, поэтому для создания качественно спроектированных объектно-ориентированных приложений хорошо подходят цикличные методы, присущие гибкой методологии разработки программ Agile Software Movement (http://agilemanifesto.org/). Цикличность гибкой разработки позволяет регулярно корректировать конструкцию и планомерно развивать ее. Если же конструкция навязывается издалека, то сложно проводить нужные корректировки и возникающие из-за недопонимания проблемы отражаются на коде. Программисты, вынужденные создавать приложения без связи с проектировщиком, начинают говорить: «Конечно, я могу создать этот код, *но это совсем не то, что вам действительно нужно, и со временем вы об этом пожалеете*».

Когда нужно приступать к проектированию

Методология гибкой разработки предполагает, что ваши клиенты не могут определить, какое именно приложение им нужно, пока не увидят его, поэтому нужно им его показать как можно скорее. Если это предположение верно, то из него логически вытекает, что программу нужно создавать постепенно, понемногу нащупывая путь к тому, что будет отвечать реальным потребностям клиента. Гибкость предполагает, что самый экономичный способ создания такого продукта заключается в сотрудничестве с этим клиентом (конструировать приложение поэтапно, небольшими объемами, чтобы каждый освоенный объем давал возможность продумать реализацию следующего объема). Опыт применения методологии гибкости подсказывает, что такое сотрудничество приводит к созданию программ, отличающихся от того, что представлялось изначально; получившееся в результате приложение не могло бы быть спрогнозировано с помощью каких-либо иных средств.

Если гибкость ведет нас правильным курсом, значит, справедливы и два других положения. Во-первых, нет абсолютно никакого смысла в объемном предварительном проектировании — Big Up Front Design (BUFD), во-вторых, никто не может точно предсказать, когда приложение будет готово (потому что заранее неизвестно, что оно в конечном итоге будет делать).

Неудивительно, что некоторым гибкая разработка не по душе. Трудно убеждать людей фразами «Мы не знаем, что делаем» и «Мы не знаем, когда это

будет готово». Стремление к BUFD сохраняется, потому что отдельным разработчикам эта технология дает ощущение контроля.

BUFD неизбежно усложняет отношения между клиентами и программистами. Поскольку ни один проект, находящийся еще в стадии разработки, не может быть безупречным, невозможно гарантировать, что приложение будет работать в соответствии с указаниями клиента и отвечать всем его нуждам. Обычно клиенты обнаруживают это при попытке использования приложения. Затем они требуют внесения изменений. Программисты противятся этим изменениям, поскольку у них есть срок, которого нужно придерживаться и который уже, возможно, прошел. В результате такой проект будет обречен на провал.

Причины этих столкновений понятны всем. Когда проект не выдерживает крайних сроков готовности (даже если это случается из-за изменений в спецификации), виноваты программисты. Однако если он готов в срок, но не удовлетворяет фактическим требованиям, значит, что-то упущено в спецификации и виноват клиент. Конструкторская документация BUFD как дорожная карта для разработки приложения, но постепенно она становится причиной разногласий.

Если в ожидании других результатов все повторяется снова и снова, программисты с теплотой вспоминают про манифест о гибком программировании — Agile Manifesto. Гибкая разработка удобна, потому что она *заранее* признает определенную недосягаемость существования идеального приложения. Таким образом, технологии гибкой разработки предлагают решения, когда при создании программы нет представления ни о целях, ни о сроках.

Хотя в концепции гибкой разработки говорится, что «не следует заниматься объемным предварительным проектированием», это еще не означает полного отказа от проектирования. Слово *«проект»* в BUFD имеет иное значение, чем в ООП. BUFD предусматривает полную спецификацию и всестороннее документирование всех ожидаемых свойств предлагаемого приложения. Если к работе привлекается архитектор программного обеспечения, проект может быть расширен после предварительного решения о структурной организации всего кода. Объектно-ориентированное проектирование связано со значительно более узкой областью. Оно систематизирует имеющийся код, чтобы его можно было легко изменить.

Процессы гибкой разработки *гарантируют изменения*, и возможность вносить эти изменения зависит от конструкции вашего приложения. Если вы не способны создавать качественно спроектированный код, то вам придется переписывать приложение на каждом этапе.

Таким образом, гибкая разработка не запрещает проектирование, а *требует* его проведения. Но понадобится *качественное* проектирование. Вам придется выложиться по полной. Успех гибкой разработки зависит от простого и удобного для изменения кода.

Оценка проектирования

В былые времена программистов иногда оценивали по количеству написанных ими строк кода (по так называемому *Source Lines Of Code, SLOC*). Происхождение этой оценочной системы не вызывает сомнений: любой начальник, полагающий, что программирование сродни работе на сборочном конвейере, где рабочие с одинаковой квалификацией собирают одинаковые виджеты, легко поверит в то, что индивидуальную выработку можно оценить простым подсчетом выданной продукции. Для руководителей, которым требуется способ сравнить эффективность разных программистов и оценить программный продукт, показатель SLOC при всех его очевидных проблемах был лучше, чем ничего.

Этот показатель явно разрабатывался не программистами. В то время как SLOC мог оценить индивидуальные трудозатраты и сложность приложения, он ничего не говорил об общем качестве. Если рядом сидящего программиста-новичка сочтут более продуктивным, поскольку на разработку функции у него ушла масса кода, а вы способны создать такую же функцию всего за несколько строк, то какой будет ваша реакция? Данный показатель изменяет систему поощрений таким образом, что это вредит качеству.

В современном мире SLOC уже антиквариат, его практически везде заменили новыми оценочными системами. Существует множество Ruby gem-пакетов (самые последние из них можно найти в поисковике Google по ключевой фразе *ruby metrics*), которые могут оценить, насколько полно ваш код следует принципам ООП. Оценочные программы сканируют исходный код и подсчитывают все, что говорит о его качестве. Запуск оценочного пакета в отношении вашего собственного кода может открыть вам глаза на его качество, обескуражить, а иногда и встревожить. Как оказалось, в толково спроектированных приложениях можно обнаружить внушительное количество нарушений принципов ООП.

Плохие оценочные показатели объектно-ориентированного проектирования, несомненно, являются признаком неудачного проектирования; код с низкими оценками *будет* трудно изменить. К сожалению, высокие показатели не дока-

зывают обратное, то есть не гарантируют, что любое вносимое вами изменение будет легким или незатратным. Проблема в том, что можно создавать превосходные проекты с прицелом на будущее. И, хотя они могут выдавать очень высокие результаты при оценке качества ООП, если они нацелены на *неверно предусмотренное* будущее, то при его наступлении исправления могут оказаться очень затратными. Оценочные показатели объектно-ориентированного проектирования не могут идентифицировать проекты, которые правильно работают, но совершают не те действия, какие требуются.

Следовательно, поучительная история о том, как оценочная система SLOC встала на ложный путь, распространяется и на оценочные системы ООП. К ним следует относиться с долей скепсиса. Оценочные системы полезны, поскольку беспристрастны и выдают числовые показатели, исходя из которых можно сделать выводы относительно программного продукта. Но это не показатель качества, а промежуточные данные, свидетельствующие о необходимости более глубокого исследования. Конечной оценкой программного продукта будут *затраты на каждую функцию в течение значимого интервала времени*, но вычислить ее непросто. Порознь определить, отследить и оценить затраты, функционал и время весьма нелегко.

Даже если вы сможете изолировать отдельно взятую функцию и отследить все связанные с ней затраты, на оценку кода также влияет *значимый интервал времени*. Иногда наличие какой-то функции важно именно сейчас настолько, что оно перевешивает любое будущее увеличение затрат. Если отсутствие функции заставит вас отойти от дел сегодня, то уже неважно, во что обойдется занятие кодом завтра; все возможное нужно сделать в имеющееся у вас время. Подобный компромисс в проектировании известен как *технический долг*. Это кредит, который в конечном счете должен быть погашен, и вполне возможно, что с процентами.

Даже если вы неумышленно оказались в техническом долгу, помните, что проектирование отнимает время и поэтому стоит денег. Поскольку вашей целью является создание программного продукта, то решение, касающееся объема проектирования, зависит от двух обстоятельств: вашего мастерства и отведенного на проектирование времени. Если проектирование в этом месяце займет половину вашего времени и не начнет приносить прибыль в течение года, то, возможно, вряд ли стоит заниматься им в таком объеме. Если объемы проектирования не позволяют сдать продукт в срок, значит, вы проиграли. Поставку половины качественно спроектированного приложения можно сравнить с полным срывом работы. Но если проектирование заняло у вас полдня (и это время

окупится в течение всего дня), а затем станет приносить прибыль на протяжении всего жизненного цикла приложения, значит, вы получаете своеобразную ежедневную капитализацию процентов с затраченного времени. Эти затраты на проектирование окупаются навсегда.

Точка безубыточности для проектирования зависит от программиста. Неопытные программисты, тратящие много времени на опережающее проектирование, могут никогда не достичь того момента, когда окупятся затраченные усилия. Опытные проектировщики, утром создающие грамотный код, уже днем могут сэкономить средства. Ваш уровень опытности находится, наверное, между этими двумя крайностями. Моя книга даст вам уроки мастерства, которыми можно воспользоваться для смещения точки безубыточности в нужном вам направлении.

Краткое введение в объектно-ориентированное программирование

Объектно-ориентированные приложения состоят из объектов и сообщений, передаваемых между ними. Сообщения играют более важную роль, но в данном кратком введении (и в первых главах этой книги) эти понятия тождественны.

Процедурные языки

Можно выделить два стиля программирования: *объектно-ориентированное* и *процедурное* программирование. Полезно рассматривать оба стиля с учетом тех понятий, которые отличают их друг от друга. Представим себе универсальный процедурный язык программирования, один из тех, на котором вы создаете простые сценарии. В этом языке можно определять переменные, то есть придумывать имена и связывать их со значениями. После присваивания к этим значениям можно обращаться, ссылаясь на переменные.

Как и все процедурные языки, наш язык поддерживает такие небольшие фиксированные наборы разнообразных данных, как строки, числа, массивы, файлы и т. д. Эти разновидности данных известны как *типы данных*. Каждый тип данных описывает конкретные виды элементов. *Строковый* тип данных отличается от *файлового*. Синтаксис языка содержит встроенные операции для

совершения действий над различными типами данных. Например, с их помощью можно объединять строки и читать данные из файлов.

Поскольку переменные создаются *вами*, вы знаете, какой тип данных содержит каждая из них. Ваши ожидания насчет видов операций, которыми можно воспользоваться, основываются на ваших знаниях о типе данных переменной. Вы знаете, что строки можно дополнить, с числами провести математические операции, зайти по индексам в массивы, прочитать файлы и т. д.

Все возможные типы данных и операции уже созданы, они встроены в синтаксис языка. Язык позволяет создавать функции (группировать ряд предопределенных операций под новым именем) или определять сложные структуры данных (собирать ряд предопределенных типов данных в имеющую собственное имя упорядоченную структуру), но вы не можете составлять абсолютно новые операции и помечать новые типы данных. Вы располагаете только тем, что видите.

В нашем языке, как и во всех процедурных языках, между данными и поведением целая пропасть. Данные упаковываются в переменные, а затем передаются поведению, которое может влиять на них практически как угодно. Данные похожи на ребенка, которого поведение каждое утро отправляет в школу. Не существует способа узнать, что на самом деле происходит, когда он вне поля зрения. Влияние на данные может быть непредсказуемым, и его сложно отследить.

Объектно-ориентированные языки

Теперь представьте себе другую разновидность языка программирования, основанного на использовании классов объектно-ориентированного языка, наподобие Ruby. Вместо разделения данных и поведения на две отдельные, нигде не сходящиеся сферы Ruby объединяет их в одно целое под названием *«объект»*. У объектов есть поведение, и они могут содержать данные, доступом к которым управляют только они. Объекты предопределяют поведение друг друга, отправляя друг другу *сообщения*.

В Ruby вместо строкового *типа данных* предусмотрен строковый *объект*. Операции, работающие со строками, встроены в сами строковые объекты, а не в синтаксис языка. Строковые объекты отличаются друг от друга тем, что каждый из них содержит свою собственную персональную *строку* данных, но схожи тем, что каждый ведет себя точно так же, как все остальные. Каждая строка *инкапсулируется*, то есть скрывает свои данные от остального мира. Каждый

объект сам для себя решает, какой объем своих данных (больший или меньший) открыть для всех остальных.

Поскольку строковые объекты поддерживают свои собственные операции, Ruby не нужно знать ничего конкретного о строковом типе данных, ему нужно лишь предоставить объектам общий способ отправки сообщений. Например, если строки понимают сообщение `concat`, Ruby не должен содержать синтаксис для объединения строк, он всего лишь должен предоставить способ отправки `concat` от одного объекта другому.

Даже самому простому приложению, скорее всего, понадобится несколько строк, или чисел, или файлов, или массивов. Конечно, иногда может потребоваться уникальный объект с индивидуальными, ни на что не похожими свойствами, но чаще всего приходится создавать множество объектов, имеющих одинаковое поведение, но инкапсулирующих различные данные.

Объектно-ориентированные языки, основанные на применении классов, подобные Ruby, позволяют определить *класс*, представляющий прообраз конструкции похожих объектов. Класс определяет *методы* (манеры поведения) и *атрибуты* (переменные). Методы вызываются в качестве реакции на сообщения. Методы с одинаковыми именами могут быть определены во многих различных объектах, а Ruby для любого отправленного сообщения должен найти и вызвать нужный метод нужного объекта.

Сам факт существования класса `String` позволяет использовать его многократно для *создания новых экземпляров* строкового объекта. В каждом новом созданном экземпляре `String` *реализуются* одни и те же методы и используются одни и те же имена атрибутов, но каждый из них содержит свои собственные персональные данные. Все такие экземпляры совместно применяют одни и те же методы, поэтому ведут себя как строки, `String`-объекты; но они содержат разные данные, поэтому представляют собой разные объекты.

Класс `String` определяет тип, представляющий собой нечто большее, чем просто *данные*. Располагая сведениями о типе объекта, вы можете выстраивать предположения насчет того, как он будет себя вести. В процедурных языках переменные имеют единственный тип данных, позволяющий предполагать, какие операции ему подойдут. В Ruby у объекта может быть много типов, один из которых всегда будет поступать от его класса. Следовательно, сведения о типе (или типах) объекта позволяют строить предположения насчет сообщений, на которые он будет отзываться.

Ruby поставляется с набором предопределенных классов. Наиболее узнаваемы те из них, которые проецируются на типы данных, используемые процедур-

ными языками. Например, класс `String` определяет строки, класс `Fixnum` определяет целые числа. Изначально существует класс для каждого типа данных, наличие которого можно ожидать от языка программирования. Но сами по себе объектно-ориентированные языки построены на использовании объектов, и тут возникают весьма интересные обстоятельства.

Класс `String`, то есть прообраз для новых строковых объектов, *сам по себе является объектом*; он представляет собой экземпляр объекта `Class`. Точно так же, как любой строковый объект отличается лишь данными экземпляра класса `String`, каждый объект класса (`String`, `Fixnum` *и так до бесконечности*) отличается лишь данными экземпляра класса `Class`. Класс `String` производит новые строки, а `Class` — новые классы.

Таким образом, объектно-ориентированные языки являются открытыми. Они не ограничивают вас небольшим набором встроенных типов и предопределенных операций. Вы можете самостоятельно придумать абсолютно новые типы. Каждое объектно-ориентированное приложение постепенно становится уникальным языком программирования, предназначенным конкретно для вашей предметной области.

Принесет ли вам этот язык в конечном счете удовольствие или доставит хлопоты — это вопрос проектирования, которому посвящена данная книга.

Выводы

Если приложению в случае удачи уготована долгая жизнь, самой серьезной проблемой станет внесение изменений. Задача проектирования состоит в создании такой структуры кода, которая позволит эффективно приспосабливаться к изменениям. Наиболее заметными элементами проектирования являются принципы и шаблоны, но, к сожалению, даже правильное применение принципов и надлежащее использование шаблонов не гарантируют создания приложения, которое легко поддается изменениям.

Выявить, насколько строго приложение следует принципам ООП, помогают соответствующие оценочные системы. Плохие показатели являются явным признаком будущих проблем, однако хорошие показатели мало что дают. Недостаточно хороший проект может выдать отличные показатели, но при этом оставаться затратным по части изменений.

Чтобы получить максимальную отдачу от затраченных на проектирование ресурсов, нужно разобраться в теории проектирования и должным образом

применить ее в нужное время и в нужных объемах. Успех в проектировании зависит от вашего умения применить теорию на практике.

А в чем разница между теорией и практикой? Фактически ни в чем. Если бы теория *была* практикой, можно было бы выучить правила ООП, неустанно применять их в работе — и с этого дня создавать идеальный код и на этом завершить свою работу. Однако критерием истины практика все же является. В отличие от теории она требует приложения усилий. Практика — это укладка кирпичей, сооружение мостов и написание кода. Практика живет в реальном мире изменений, путаницы и неопределенности. Она сталкивается с конкурирующими вариантами и выбирает наименьшее из зол, она изворачивается, она подстраховывается. Она зарабатывает на жизнь, делая все возможное с тем, чем располагает.

Теория полезна и необходима, и именно ей была посвящена текущая глава. Но этого уже вполне достаточно, настало время перейти к практике.

Глава 2
Проектирование классов с единственной обязанностью

Основой объектно-ориентированной системы является сообщение, но наиболее заметная организационная структура — класс. Сообщения составляют ядро проекта, но, поскольку классы в приложении заметнее, эта глава начинается с малого и в первую очередь мы рассмотрим, как решить, что должно принадлежать классу. В нескольких следующих главах акцент в проектировании будет постепенно смещаться от классов к сообщениям.

Что представляют собой классы? Сколько их должно быть? Какое поведение в них должно быть реализовано? Каким объемом информации о других классах они располагают? Какой объем информации они сами должны давать?

Подобные вопросы могут привести в замешательство. Каждое кажущееся незыблемым решение таит в себе опасность. Но не стоит бояться: на данном этапе вам нужно глубоко вздохнуть и поверить в то, что вы справитесь. Ваша цель — создать модель приложения с использованием классов, чтобы оно выполняло свою функцию сейчас и позволяло легко внести нужные изменения потом. Это два совершенно разных критерия. Любой программист может заставить код работать именно сейчас. Однако совершенно другое дело — создать приложение, которое можно будет легко изменить. Это качество и является показателем мастерства программирования. Для его достижения требуются знания, а также опыт и чуть-чуть творчества.

К счастью, вы не должны во всем разбираться с нуля. Существуют определенные свойства приложения, гарантирующие легкость внесения изменений. Для того чтобы выявить их, было проведено множество исследований. Методика проста, и вам нужно лишь знать, что они собой представляют и как их использовать.

Что должно принадлежать классу

Есть замысел приложения. Вам известно, что оно должно делать. У вас даже могут быть мысли о том, как реализовать наиболее интересные моменты его поведения. Проблема не в технических знаниях, а в организации. Вы знаете, как написать код, но не знаете, куда его поместить.

Группировка методов в классы

В таких объектно-ориентированных языках, как Ruby, методы определяются в классах. Создаваемые вами классы впредь будут влиять на то, как вы представляете себе свое приложение.

Несмотря на важность правильной группировки методов в классы, на ранней стадии разработки проекта вы не сможете все сделать правильно. Вы уже никогда не будете знать меньше, чем знаете сейчас. Даже если ваше приложение успешно работает, многие решения, принятые сегодня, позже придется изменить. Когда наступит этот день, ваша возможность внесения этих изменений будет определяться конструкцией приложения.

Проектирование в большей степени является искусством сохранения возможности вносить изменения, чем способом достичь совершенства.

Организация кода для легкого внесения изменений

Утверждение, что код должен быть легким для изменения, сродни тому, что дети должны быть вежливыми. С ним нельзя не согласиться, но оно ни в коей мере не помогает родителям воспитать послушного ребенка. Идея легкости носит слишком общий характер: вам требуются конкретное определение легкости и четкие критерии, по которым можно будет оценивать код.

Легкость внесения изменений можно определить по следующим критериям:

- изменения не имеют неожиданных побочных эффектов;
- незначительные изменения в требованиях влекут за собой соответствующие незначительные изменения в коде;
- код легко поддается повторному использованию;
- самый простой способ внести изменения — добавить код, который сам по себе нетрудно изменить.

В этом случае создаваемый вами код должен быть:

- **понятным** (последствия внесения изменений должны быть вполне очевидны как в самом изменяемом коде, так и в удаленном, зависящем от него коде);
- **обоснованным** (стоимость внесения любого изменения должна быть пропорциональна преимуществам, достигаемым в результате изменения);
- **практичным** (существующий код должен быть практичен как в новом, так и в неожиданном контексте);
- **образцовым** (код должен побуждать тех, кто с ним работает, закрепить новые навыки).

Понятный (Transparent), обоснованный (Reasonable), практичный (Usable) и образцовый (Exemplary) код (по первым буквам TRUE) не только отвечает современным требованиям, но и может быть изменен для соответствия будущим потребностям. Первый шаг по созданию TRUE-кода заключается в обеспечении каждого класса единственной, четко определенной обязанностью.

Создание классов с единственной обязанностью

Класс должен совершать наименьшее возможное полезное действие, то есть у него должна быть единственная обязанность.

Для того чтобы проиллюстрировать создание класса с единственной обязанностью и объяснить, почему это важно, приведу пример, который требует небольшого отступления про велосипеды.

Пример приложения: велосипеды и передаточный механизм. Велосипеды — удивительно эффективные средства передвижения, отчасти потому, что в них используются передаточные механизмы. Катаясь на велосипеде, можно

выбрать между небольшой звездочкой (можно легко крутить педали, но не слишком быстро ездить) и большой звездочкой (труднее крутить педали, но можно хорошо разогнаться). Передаточный механизм с разными звездочками очень полезен, поскольку маленькую звездочку можно использовать, карабкаясь в гору, а большую — для того чтобы лететь вниз с горы.

Передаточный механизм позволяет менять протяженность проходимого пути за один оборот педалей. Точнее, этот механизм управляет тем, сколько оборотов делает колесо при каждом обороте педалей. При небольшой звездочке для одного оборота колеса требуется несколько оборотов педалей, а при большой звездочке каждый полный оборот педалей может привести к нескольким оборотам колеса (рис. 2.1).

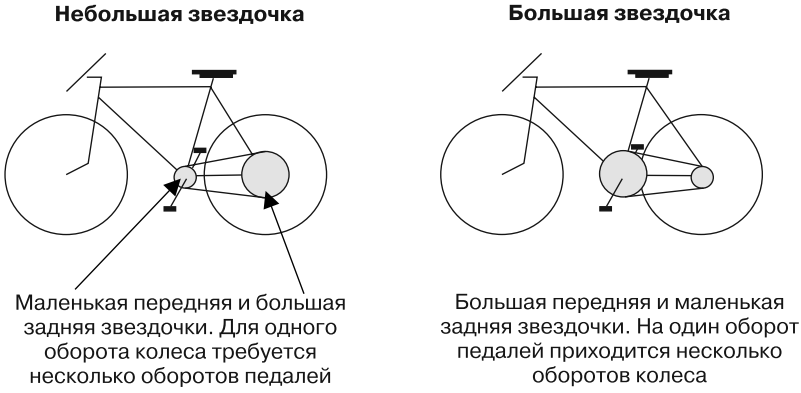

Рис. 2.1. Сравнение больших и малых звездочек велосипедной передачи

Понятия маленького и большого не отличаются точностью. Для сравнения различных звездочек велосипедисты используют соотношение количества их зубцов. Его можно вычислить с помощью следующего простого сценария на языке Ruby:

```
chainring = 52                          # количество зубцов
cog       = 11
ratio     = chainring / cog.to_f
puts ratio                              # -> 4.72727272727273

chainring = 30
cog       = 27
ratio     = chainring / cog.to_f
puts ratio                              # -> 1.11111111111111
```

Передаточный механизм, созданный сочетанием 52-зубцовой ведущей и 11-зубцовой ведомой звездочек (52 × 11), имеет передаточное отношение приблизительно 4,73. На каждый оборот педалей приходится почти пять оборотов колеса. Передаточное отношение 30 × 27 позволяет легче крутить педали; на каждый оборот педалей приходится чуть более одного оборота колеса.

Хотите верьте, хотите нет, но есть люди, которых очень интересует передаточный механизм велосипеда. Вы можете им помочь, написав на Ruby приложение для вычисления передаточных отношений.

Приложение будет состоять из классов Ruby, каждый из которых станет представлять какую-то область. Если в приведенном выше описании отметить имена существительные, представляющие объекты предметной области, это такие слова, как велосипед (bicycle) и передача (gear). Эти существительные представляют собой наипростейших кандидатов на превращение в классы. Интуиция подсказывает, что велосипед (bicycle) должен стать классом, но в приведенном выше описании не перечислено ничего, что имело бы отношение к поведению велосипеда, поэтому пока у него нет никаких оснований для превращения в класс. А вот у передачи (gear) есть ведущая и ведомая звездочки и передаточное отношение, то есть у нее есть как данные, так и поведение. Она заслуживает того, чтобы стать классом. Возьмем поведение из показанного выше сценария и создадим следующий простой класс `Gear`:

```
class Gear
  attr_reader :chainring, :cog
  def initialize(chainring, cog)
    @chainring = chainring
    @cog       = cog
  end

  def ratio
     chainring / cog.to_f
  end
end

puts Gear.new(52, 11).ratio             # -> 4.72727272727273
puts Gear.new(30, 27).ratio             # -> 1.11111111111111
```

Класс `Gear` — сама простота. Вы создаете новый `Gear`-экземпляр, указывая количество зубцов на ведущей и ведомой звездочках (на `chainring` и `cog` соответственно). В каждом экземпляре реализуются три метода: `chainring`, `cog` и `ratio`.

`Gear` является подклассом класса `Object` и наследует множество других методов. Класс `Gear` состоит из всего, что в нем реализовано напрямую, плюс из всего унаследованного, следовательно, полный поведенческий набор, то есть общий набор сообщений, на которые он может реагировать, довольно большой. Наследование имеет значение для проектирования вашего приложения, но этот простой случай, где `Gear` наследует свойства объекта, настолько типовой, что по крайней мере сейчас вы можете действовать, как будто этих унаследованных методов не существует. Более сложные формы наследования будут рассматриваться в главе 6.

Вы показали свой калькулятор передаточных отношений знакомому велосипедисту, он сказал, что вещь полезная, но тут же потребовал усовершенствования. У него два велосипеда с абсолютно одинаковыми передаточными механизмами, но размеры колес у них разные. Ему хотелось бы, чтобы в расчет бралась также разница в размерах колес.

Как показано на рис. 2.2, велосипед с более крупным колесом проходит за один оборот педалей большее расстояние, чем велосипед с колесом меньшего размера.

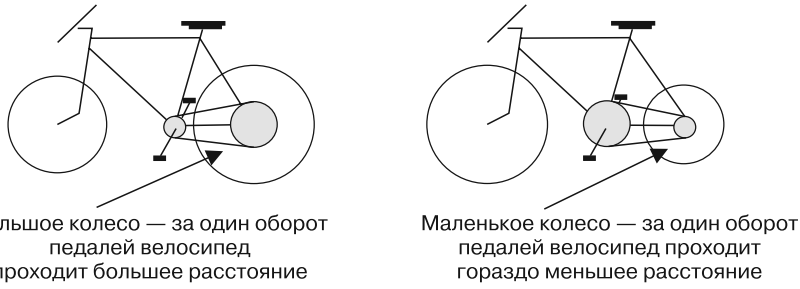

Рис. 2.2. Влияние размера колеса на проходимое расстояние

Велосипедисты (по крайней мере те, что живут в США) для сравнения велосипедов, отличающихся как передачами, так и размерами колес, используют такой показатель, как передаточное отношение в дюймах (gear inches). Для его вычисления используется следующая формула:

Передаточное отношение в дюймах (gear inches) = диаметр колеса (wheel diameter) × передаточное отношение (gear ratio),

где

диаметр колеса = диаметр обода (rim diameter) + двойной диаметр (высота) шины (twice tire diameter).

Чтобы добавить новое поведение, вы внесли изменения в класс `Gear`:

```
class Gear
 attr_reader :chainring, :cog, :rim, :tire
 def initialize(chainring, cog, rim, tire)
    @chainring = chainring
    @cog       = cog
    @rim       = rim
    @tire      = tire
 end

 def ratio
   chainring / cog.to_f
 end

 def gear_inches
   # определяем диаметр с учетом, что шина дважды проходит по ободу
   ratio * (rim + (tire * 2))
 end
end

puts Gear.new(52, 11, 26, 1.5).gear_inches
# -> 137.090909090909

puts Gear.new(52, 11, 24, 1.25).gear_inches
# -> 125.272727272727
```

Новый метод `gear_inches` предполагает, что размеры обода (`rim`) и шины (`tire`) задаются в дюймах, а эти значения могут быть указаны как правильно, так и неправильно. С этой оговоркой класс `Gear` отвечает установленным требованиям (в их текущем состоянии), а код, за исключением следующей ошибки, работает.

```
puts Gear.new(52, 11).ratio # Это раньше работало?
Неверное количество аргументов (два вместо четырех)
# from (irb):20:in 'initialize'
# from (irb):20:in 'new'
# from (irb):20

```

Показанная выше ошибка появилась после добавления метода `gear_inches`. Метод `Gear.initialize` изменился и теперь требует два дополнительных

аргумента, `rim` и `tire`. Изменение количества аргументов, требуемых методу, нарушило работу всех строк кода, вызывающих метод. Обычно это весьма серьезная проблема, требующая немедленного решения, но, поскольку приложение настолько мало, что у `Gear.initialize` в настоящий момент нет другого вызывающего этот метод кода, пока ошибку можно проигнорировать.

Теперь, когда элементарный класс `Gear` уже существует, настало время задать вопрос: является ли рассмотренный способ организации кода наилучшим?

Ответ неизменно будет звучать так: все зависит от конкретных обстоятельств. Если ожидается, что приложение так и останется статическим, то класс `Gear` в его текущей форме вполне подойдет. Хотя уже сейчас можно увидеть перспективу превращения всего приложения в этакий калькулятор для велосипедистов.

И в развивающемся приложении `Gear` станет лишь первым из множества классов. Но, чтобы развитие было эффективным, код должен быть легко изменяемым.

Почему именно единственная обязанность

Легко изменяемые приложения состоят из классов, которые запросто поддаются повторному использованию. Повторно используемые классы представляют собой подключаемые элементы с четко определенным поведением, у которых мало переплетений. Приложение, легко поддающееся изменениям, похоже на вместилище строительных блоков; вы можете выбрать только те из них, которые вам нужны, и собрать их совершенно неожиданным образом.

Класс, имеющий несколько обязанностей, трудно поддается повторному использованию. При наличии сразу нескольких обязанностей высока вероятность их сильных переплетений внутри класса. Если потребуется повторно использовать некоторые из сторон его поведения (но не все), то невозможно получить только нужные части. У вас два варианта, и ни один из них не отличается особой привлекательностью.

Если обязанности настолько связаны друг с другом, что вы не можете воспользоваться только нужным поведением, можно продублировать интересу-

ющий вас код. Но это не самая хорошая идея. Продублированный код предполагает дополнительные затраты на поддержку и увеличивает количество ошибок. Если класс структурирован таким образом, что можно получить доступ только к нужному вам поведению, можно повторно воспользоваться всем классом. Но при этом одна проблема просто заменяется другой.

Поскольку в действиях повторно используемого класса нетрудно запутаться и в нем содержится несколько переплетенных друг с другом обязанностей, существует множество причин для его изменения. В то же время класс может быть изменен по причине, не связанной с тем, что вы им пользуетесь, и при каждом таком изменении есть риск нарушения работы любого зависящего от него класса. Полагаясь на классы с широким кругом обязанностей, вы увеличиваете риск неожиданного отказа вашего приложения.

Определение наличия у класса единственной обязанности

Как можно определить, содержит ли класс `Gear` востребованное где-нибудь еще поведение? Можно заподозрить у него интеллект и устроить допрос. Если представить себе каждый его метод в виде вопросов, то у любого вопроса должен быть определенный смысл. Например, вопрос «Пожалуйста, мистер Gear, скажите, какое у вас передаточное отношение (`ratio`)?» выглядит вполне осмысленным; вопрос «Пожалуйста, мистер Gear, скажите, каков ваш показатель передаточного отношения в дюймах (`gear_inches`)?» сомнителен, а вопрос «Пожалуйста, мистер Gear, скажите, какого размера у вас шина (`tire`)?» задавать просто смешно.

Не будем возражать, что вопрос «Что представляет собой ваша шина?» может быть задан на вполне законном основании. Изнутри класса `Gear` шина (`tire`) может представляться понятием другого вида, нежели обычное или дюймовое передаточное отношение (`ratio` или `gear_inches`), но это не играет никакой роли. С точки зрения любого другого объекта `Gear` может реагировать лишь на сообщение.

Чем же класс фактически занимается, можно узнать, попытавшись описать его обязанности одним предложением. Вспомним, что класс обязан совершать хотя бы одно полезное действие, которое должно поддаваться простому

описанию. Если самое простое описание можно разделить с помощью союза «и», то у класса, скорее всего, несколько обязанностей. Если же используется слово «или», то у класса несколько обязанностей и у них могут быть весьма слабые связи.

Для описания данного понятия проектировщики объектно-ориентированных программных продуктов используют слово *«сцепление»* (cohesion). Если все, что есть в классе, имеет отношение к его главному предназначению, они говорят, что класс имеет сильное сцепление или что у него единственная обязанность. Принцип единственной обязанности — Single Responsibility Principle (SRP) — берет начало из идеи проектирования на основе обязанностей — Responsibility-Driven Design (RDD), выдвинутой Ребеккой Вирфс-Брок (Rebecca Wirfs-Brock) и Брайаном Уилкерсоном (Brian Wilkerson). Они сказали: «У класса имеются обязанности, удовлетворяющие его назначению». SRP не требует, чтобы класс совершал лишь одно узконаправленное действие или изменялся лишь по одной вполне конкретной причине, напротив, SRP требует, чтобы класс имел сильное сцепление, чтобы все, чем он занимается, было сильно связано с его назначением.

Как можно описать обязанность класса `Gear`? Может быть, так: «Вычислить соотношение между двумя зубчатыми звездочками»? Если это так, то класс в его нынешнем виде совершает слишком много действий. А может быть, так: «Вычислить воздействие передаточного механизма на велосипед»? Тогда `gear_inches` получает реальное основание, а для размера шины (`tire`) задача все еще остается весьма неясной.

Похоже, с классом не все в порядке. У `Gear` больше одной обязанности, но что с ним делать, пока непонятно.

Когда следует принимать проектировочные решения

Иногда становится понятно, что с классом что-то не так. Правильно ли называть этот класс `Gear`? Ведь в нем есть такие понятия, как обод (`rim`) и шина (`tire`)! Может, он должен называться не передачей (`Gear`), а велосипедом (`Bicycle`)? Или колесом (`Wheel`)?

Если бы только знать обо всех будущих запросах, то сегодня можно было бы создать идеальное конструкторское решение. К сожалению, неизвестно, какими будут эти запросы. Может случиться что угодно. Можно потратить впустую

уйму времени, выбирая между двумя равнозначными вариантами, бросить монетку — и все равно сделать неверный выбор.

Не нужно принимать преждевременных проектировочных решений. Сталкиваясь с таким несовершенным и запутанным классом, как `Gear`, задайтесь вопросом: «Какую цену придется заплатить в будущем, если сегодня ничего с ним не делать?»

Это очень маленькое приложение. У него один разработчик. Вы хорошо знаете класс `Gear`. Будущее туманно, но меньше, чем сейчас, вы знать уже не будете. Наиболее экономически оправданным действием будет ожидание поступления дополнительной информации.

Код в классе `Gear` вполне понятен и обоснован, но тот факт, что у класса нет зависимостей и в него можно вносить изменения без каких-либо последствий, еще не означает, что он превосходно спроектирован. Если бы он приобрел зависимости, то тут же утратил бы оба этих качества — и пришлось бы безотлагательно заняться его реорганизацией. Кстати, новые зависимости дадут точную информацию, необходимую для принятия правильных проектировочных решений.

Когда будущие потери от бездействия равны стоимости приложенных усилий сегодня, отложите решение. Принимайте его только тогда, когда поступившая к вам информация требует сделать это незамедлительно.

Даже при наличии весомого аргумента в пользу того, чтобы оставить `Gear` в его нынешнем виде, можно также привести не менее весомый аргумент в пользу его изменения. Структура каждого класса является своеобразным посланием тем, кто в будущем станет сопровождать приложение. Она раскрывает ваши проектировочные намерения. Хорошо это или плохо, но шаблоны, закладываемые вами сегодня, будут повторяться всегда.

`Gear` не дает правдивой картины ваших намерений. Он не может считаться ни практичным, ни образцовым. У него несколько обязанностей, поэтому его нельзя применять повторно (он не является шаблоном). Есть вероятность, что классом `Gear` воспользуется кто-нибудь другой или по его шаблону создаст новый код, пока вы будете ждать более полезную информацию.

Противоречие «улучшить сейчас» или «улучшить потом» сохраняется всегда. Идеально спроектированных приложений не бывает. У каждого решения своя цена. Толковый проектировщик понимает суть этого противоречия и сводит к минимуму затраты, идя на сознательный компромисс между настоящими потребностями и будущими возможностями.

Создание кода, легко принимающего изменения

В `Gear` можно создать такую структуру кода, которая позволит этому классу стать легко изменяемым, даже если вы еще ничего не знаете о грядущих изменениях. Поскольку изменений все равно не избежать, программирование, допускающее внесение правок в код, проявит себя в будущем. Дополнительный бонус: такое программирование уже сегодня улучшит ваш код, не требуя затрат.

Для создания кода, легко принимающего изменения, можно воспользоваться некоторыми известными технологиями.

Полагайтесь на поведение, а не на данные

Поведение отражается в методах и вызывается отправкой сообщений. При создании классов с единственной обязанностью любой фрагмент поведения находится в одном и только в одном месте. Фраза «Не повторяйтесь» (Don't Repeat Yourself, DRY) — краткое выражение этой идеи. DRY-код легко принимает изменения, поскольку любое изменение в поведении можно внести только в одном месте кода.

Вдобавок к поведению объекты зачастую содержат данные. Они хранятся в переменных экземпляра и могут быть чем угодно (от простой строки до сложного хеша). Доступ к данным можно получить одним из двух способов: обратиться к переменной экземпляра напрямую или же заключить ее в метод доступа.

Скрытие переменных экземпляра

Вместо непосредственного обращения к переменным экземпляра их всегда следует заключать в метод доступа, как это сделано в методе `ratio`:

```
class Gear
   def initialize(chainring, cog)
     @chainring = chainring
     @cog       = cog
   end

   def ratio
     @chainring / @cog.to_f    # <-- путь к гибели
   end
end
```

Скрывайте переменные даже от того класса, в котором они определены, и заключайте их в методы. В качестве простого способа создания инкапсу-

лированных методов в Ruby предоставляется метод считывания атрибутов `attr_reader`:

```
class Gear
  attr_reader :chainring, :cog   # <-------
  def initialize(chainring, cog)
    @chainring = chainring
    @cog       = cog
  end

  def ratio
    chainring / cog.to_f         # <-------
  end
end
```

Использование метода `attr_reader` заставляет Ruby создавать для переменных простые методы-оболочки. Вот как выглядит виртуальное представление одного из таких методов, созданного для переменной `cog`:

```
# исходная реализация при использовании метода attr_reader
def cog
  @cog
end
```

Теперь этот метод `cog` является единственным местом в коде, понимающим значение идентификатора `cog`. Получение `cog` становится результатом отправки сообщения. Реализация этого метода изменяет `cog`, превращая данные (являющиеся во всех отношениях ссылкой) в поведение (определяемое один раз).

Допустим, на переменную экземпляра `@cog` вы ссылаетесь десять раз. Если внезапно потребуется ее скорректировать, придется внести много изменений в код. Но если переменная `@cog` заключена в метод, изменить то, что означает `cog`, можно, реализовав свою собственную версию метода. Ваш новый метод должен быть таким же простым, как в первой реализации, показанной ниже, или же более сложным, как показано во второй реализации:

```
# простое переопределение cog
def cog
  @cog * unanticipated_adjustment_factor # непредвиденная поправка
end
```

```
# более сложное переопределение
def cog
  @cog * (foo? ? bar_adjustment : baz_adjustment)
end
```

Цели первого примера, наверное, можно было бы добиться путем единственного изменения переменной экземпляра. Но вы никогда не сможете быть уверены, что со временем не потребуется что-нибудь похожее на второй пример. Вторая корректировка при ее осуществлении в методе представляет собой простое изменение поведения, но при применении к большому количеству переменных экземпляра такой код позволяет избавиться от беспорядка.

Работа с данными как с объектом, понимающим сообщения, вызывает две новые проблемы. Первая касается видимости. Заключение переменной экземпляра `@cog` в открытый метод `cog` показывает эту переменную другим объектам приложения; теперь любой другой объект может отправить `cog` классу `Gear`. Ничуть не труднее создать закрытый метод-оболочку, превращающий данные в поведение, не открывая это поведение всему приложению. Выбор между этими двумя альтернативами рассматривается в главе 4.

Вторая проблема носит более абстрактный характер. Из-за возможности заключить каждую переменную экземпляра в метод и рассматривать ее просто как еще один объект разница между данными и обычным объектом стирается. Хотя иногда целесообразно представлять часть приложения в качестве не имеющих поведения данных, о большинстве элементов лучше иметь представление как о простых объектах.

Независимо от того, насколько далеко ваши мысли простираются в этом направлении, данные нужно скрывать. Так код будет защищен от неожиданных изменений. У данных зачастую бывает поведение, о котором вы даже не знаете. Отправляйте для доступа к переменным сообщения, даже если думаете о них как о данных.

Скрытие структур данных

Если привязываться к переменной экземпляра — не лучшее решение, то зависеть от сложной структуры данных — еще хуже. Рассмотрим следующий класс `ObscuringReferences` (скрытие ссылок):

```
1  class ObscuringReferences
2    attr_reader :data
3    def initialize(data)
4      @data = data
5    end
6
7    def diameters
8      # 0  - rim (обод), 1  — tire (шина)
9      data.collect {|cell|
```

```
10        cell[0] + (cell[1] * 2)}
11    end
12    # ... множество других методов, обращающихся по индексам к массиву
```

Ожидается, что этот класс должен быть проинициализирован двумерным массивом, содержащим размеры ободов и шин:

```
1 # размеры ободов и шин (теперь в миллиметрах!) в двумерном массиве
2 @data = [[622, 20], [622, 23], [559, 30], [559, 40]]
```

Класс `ObscuringReferences` сохраняет свои инициализирующие аргументы в переменной `@data` и послушно использует Ruby-метод `attr_reader` для заключения переменной экземпляра `@data` в метод. Метод `diameters` отправляет сообщение `data` для доступа к содержимому переменной. Этот класс делает, конечно, все необходимое, чтобы скрыть переменную экземпляра от себя самого. Но, поскольку в `@data` содержится сложная структура данных, просто скрыть переменную экземпляра еще недостаточно. Метод `data` просто возвращает массив. Чтобы сделать что-нибудь полезное, каждый отправитель данных должен иметь полные сведения о том, какая часть данных под каким индексом находится в массиве.

Метод `diameters` знает не только то, как вычислять диаметры, но и где найти в массиве размеры ободов и шин. Ему точно известно, что при переборе данных в массиве первые находятся в элементе с индексом [0], а вторые — в элементе с индексом [1].

Работа метода *зависит* от структуры массива. Если эта структура изменяется, то должен измениться и код. Когда у вас имеются данные в массиве, вы все равно должны обращаться к структуре массива. Ссылки *склонны к утечкам*. Они выходят за рамки инкапсуляции и проникают во весь код. Они не отвечают принципу DRY. Сведения о том, что размеры ободов находятся в элементе с индексом [0], не должны дублироваться; об этом должно быть сказано только в одном месте.

Этот простой пример не слишком убедителен; представим себе последствия того, что данные возвращают массив хешей, на который имеются ссылки во многих местах. Изменение его структуры вызовет цепную реакцию по всему коду; каждое изменение может привести к скрытой ошибке, а ее поиски могут довести до слез.

Прямые ссылки в сложные структуры дезориентируют, поскольку они скрывают то, что собой в действительности представляют данные, и сопровождение кода с ними может превратиться в ночной кошмар, ведь каждая ссылка будет нуждаться в изменении при изменении структуры массива.

В Ruby отделить структуру от значений совсем несложно. Точно так же, как метод используется для заключения в себя переменной экземпляра, Ruby-класс `Struct` применяется в качестве оболочки структуры. В следующем примере класс `RevealingReferences` имеет такой же интерфейс, как и предыдущий класс `ObscuringReferences`. В качестве инициализирующего аргумента он получает двумерный массив и реализует метод `diameters`. Несмотря на внешнюю схожесть, его внутренняя реализация значительно отличается от реализации предыдущего класса:

```
class RevealingReferences
  attr_reader :wheels
  def initialize(data)
     @wheels = wheelify(data)
  end

  def diameters
     wheels.collect {|wheel|
     wheel.rim + (wheel.tire * 2)}
  end
  # каждый может отправить пару размеров обод-шина структуре wheel

  Wheel = Struct.new(:rim, :tire)
  def wheelify(data)
    data.collect {|cell|
    Wheel.new(cell[0], cell[1])}
  end
end
```

Показанный выше метод `diameters` теперь имеет сведения о внутренней структуре массива. Методу `diameters` известно лишь то, что сообщение `wheels` возвращает перечисление и что каждый перечисляемый элемент отзывается на `rim` и `tire`. То, на что делалась ссылка как на `cell[1]`, превратилось в отправку сообщения `wheel.tire`.

Все сведения о структуре поступающего массива изолированы в методе `wheelify`, который превращает `Array`-массив в `Struct`-массив. В официальной документации по Ruby (http://ruby-doc.org/core/classes/Struct.html) про `Struct` говорится как про «удобный способ объединения в единое целое нескольких атрибутов с помощью методов-оболочек без необходимости создания явно определяемого класса». Именно это и делает метод `wheelify` — создает небольшие легковесные объекты, отзывающиеся на `rim` и `tire`.

В методе `wheelify` содержится только тот код, который понимает структуру поступающего массива. Если входные данные изменятся, код изменится только в одном этом месте. Для создания структуры `Wheel` и определения метода `wheelify` понадобились всего четыре строки кода, но они доставляют минимальные неудобства по сравнению с ценой, которую пришлось бы заплатить за повторяющиеся указания индексов для ссылки на элементы сложного массива.

Стиль кода позволяет вам защититься от изменений тех структур данных, которые относятся к внешним компонентам и делают код более информативным и понятным при чтении. Показанный выше метод `wheelify` изолирует запутанную структурную информацию и приводит код к DRY-стилю. Он существенно повышает терпимость этого класса к изменениям.

Повсеместное внедрение единственной обязанности

Создание классов с единственной обязанностью играет весьма важную роль для проектирования, но идея единственной обязанности может быть успешно использована во многих других частях вашего кода.

Выделение у методов лишних обязанностей

Методы, как и классы, должны иметь единственную обязанность. По тем же причинам. Наличие единственной обязанности облегчает внесение изменений и повторное использование. Здесь работают все приемы, что и раньше. Задавайте вопросы, чем методы занимаются, и старайтесь дать описание их обязанностей в одном предложении.

Посмотрите на метод `diameters` класса `RevealingReferences`:

```
1  def diameters
2    wheels.collect {|wheel|
3    wheel.rim + (wheel.tire * 2)}
4  end
```

У этого метода явно две обязанности: он осуществляет обход всех элементов `wheels` и вычисляет диаметр каждого колеса.

Упростите код, разбив его на два метода, у каждого из которых будет единственная обязанность. Эта реорганизация переместит вычисление диаметра одного колеса в свой собственный метод. Реорганизация вводит дополнительную

отправку сообщения, но здесь конструкции нужно действовать, не замечая этого. При необходимости производительность может быть улучшена позже. А сейчас важно создать код, легко поддающийся изменениям.

```
# во-первых, обход элементов массива
def diameters
  wheels.collect {|wheel| diameter(wheel)}
end

# во-вторых, вычисление диаметра ОДНОГО колеса
def diameter(wheel)
  wheel.rim + (wheel.tire * 2))
end
```

Понадобится ли вам когда-нибудь получить диаметр только одного колеса? Посмотрите на код еще раз — вам уже понадобилось. Эта реорганизация не является избыточным проектированием, в ее рамках просто переделывается уже востребованный код. Тот факт, что имеющий единственную обязанность метод `diameter` может теперь вызываться из других мест, неожиданно приводит к положительному побочному эффекту.

Отделить поэлементный обход массива от действия, совершаемого над каждым элементом, получилось благодаря тому, что это был тот самый общий случай легко распознаваемых нескольких обязанностей. В других случаях решение данной задачи не столь очевидно.

Вернемся к методу `gear_inches` класса `Gear`:

```
def gear_inches
  # определяем диаметр с учетом, что шина дважды проходит по ободу
  ratio * (rim + (tire * 2))
end
```

Является ли метод `gear_inches` обязанностью класса `Gear`? По логике да. Но если это так, почему возникает чувство, что с ним что-то не в порядке? Есть в нем какая-то запутанность и неопределенность, предвещающая возможные проблемы в будущем. Суть проблемы в том, что у этого метода более одной обязанности.

Внутри `gear_inches` скрывается вычисление диаметра колеса. Извлечение этого вычисления в новый метод `diameter` упростит исследование обязанностей класса:

```
def gear_inches
  ratio * diameter
end

```

```
5  def diameter
6    rim + (tire * 2)
7  end
```

Теперь метод `gear_inches`, чтобы получить диаметр колеса, отправляет сообщение. Обратите внимание, что реорганизация не изменяет способа вычисления диаметра, она просто изолирует поведение в отдельном методе.

Эти реорганизации следует проводить, даже не зная конечного вида проекта. Они необходимы не потому, что проект уже сложился, а потому, что он еще не сложился. Чтобы заняться ими, вам не нужно знать, где именно вы собираетесь применить практические подходы качественного проектирования. Проектирование проявляется в использовании правильных методов работы.

Эта простая реорганизация показывает суть проблемы. Класс `Gear` обязан вычислять передаточное отношение в дюймах (`gear_inches`), но не обязан вычислять диаметр колеса.

Конечно, влияние такой одной реорганизации невелико, но накопительный эффект подобного стиля программирования огромен. Методы, имеющие единственную обязанность, дают следующие преимущества:

- **выявляют ранее скрытые качества.** Реорганизация класса в попытке наделения всех методов единственной обязанностью положительно влияет на прояснение сути самого класса. Даже если сегодня вы не намереваетесь заниматься реорганизацией методов внутри других классов, заставляя каждый из них служить единственной цели, это проясняет то, чем в совокупности занимается сам класс;
- **делают ненужными комментарии.** Сколько раз вам попадались устаревшие комментарии? Вообще, комментарии не относятся к исполняемому коду, и если какой-то код внутри метода нуждается в комментарии, лучше выделить этот код в отдельный метод. Имя нового метода послужит той же цели, что и прежний комментарий;
- **способствуют повторному использованию.** Небольшие методы удобны для сопровождения приложения. Вместо того чтобы дублировать весь код, программисты смогут работать с отдельными методами. Они будут следовать созданному вами образцу и создавать, в свою очередь, небольшие повторно используемые методы. Это хорошая практика;
- **обеспечивают легкий переход к другому классу.** Небольшие методы проще перемещать, если получена дополнительная проектировочная информация и принято решение внести изменения. Поведение можно изменить без существенных переделок.

Изоляция лишних обязанностей в классах

Когда у каждого метода останется только единственная обязанность, область применения вашего класса станет более очевидной. В классе `Gear` есть поведение, присущее колесу (wheel). Нужен ли этому приложению класс `Wheel`?

Если обстоятельства позволяют вам создать отдельный класс `Wheel`, то, наверное, это нужно сделать. А теперь представьте, что вы пока решили не создавать новый, постоянный, доступный всем класс. Возможно, есть какие-то конструкторские ограничения, или, может быть, вы пока не хотите создавать новый класс, от которого могут зависеть другие классы.

Может показаться, что просто невозможно оставить в `Gear` единственную обязанность, пока не будет убрано поведение, свойственное колесу. В `Gear` либо есть избыточная обязанность, либо ее нет. Но недальновидно изменять конструкторское решение при двойственной ситуации. Есть другие варианты. Ваша цель — сохранить единственную обязанность в `Gear`, чтобы не перегружать конструкцию. Поскольку вы создаете изменяемый код, решение лучше отложить, пока не настанет момент, когда без него будет просто не обойтись. Любое решение, принимаемое раньше, чем в нем возникнет явная потребность, является просто догадкой. Так что отложите принятие решения на потом.

Ruby позволяет удалить обязанность вычисления диаметра шины из `Gear` без необходимости создания нового класса. Следующий пример является расширением предыдущей структуры `Wheel` с блоком, добавляющим метод для вычисления диаметра.

```
class Gear
  attr_reader :chainring, :cog, :wheel
  def initialize(chainring, cog, rim, tire)
     @chainring = chainring
     @cog       = cog
     @wheel     = Wheel.new(rim, tire)
  end

  def ratio
    chainring / cog.to_f
  end

  def gear_inches
    ratio * wheel.diameter
  end

```

```
  Wheel = Struct.new(:rim, :tire) do
    def diameter
      rim + (tire * 2)
    end
  end
end
```

Теперь у вас есть структура колеса (`Wheel`), способная вычислять свой собственный диаметр. Понятно, что нет смысла встраивать эту структуру `Wheel` в класс `Gear` — это в большей мере эксперимент в организации кода. Такой прием очищает передаточный механизм (`Gear`), но откладывает решение относительно колеса (`Wheel`).

Встраивание `Wheel` в `Gear` предполагает, что `Wheel` будет существовать только в контексте `Gear`. Если на минутку оторваться от книги и

к цепной реакции, распространяющейся на тесты и заставляющей вносить изменения и в них (проектирование тестов рассматривается в главе 9).

Позаботьтесь о том, чтобы ваше приложение не утонуло в ненужных зависимостях. Главное — их распознать, тогда будет нетрудно от них избавиться. Первым шагом должно стать более четкое понимание зависимостей, поэтому настало время посмотреть на примеры кода.

Создание кода со слабой связью

Каждая зависимость, словно капля клея, заставляет класс «прилипать» ко всему, с чем он соприкасается. Некоторые «точки склеивания» необходимы, но стоит только увеличить дозу клея — и ваше приложение склеится в твердый блок. Уменьшение зависимостей означает их распознавание и удаление тех, в которых нет необходимости.

Следующие примеры показывают технологии создания программного кода, уменьшающие зависимости путем расцепления кода.

Внедренные зависимости

Основную точку привязанности создает ссылка на другой класс по его имени. В рассматриваемой нами версии класса `Gear` (еще раз показанной ниже) в методе `gear_inches` содержится явная ссылка на класс `Wheel`:

```
class Gear
  attr_reader :chainring, :cog, :rim, :tire
  def initialize(chainring, cog, rim, tire)
     @chainring = chainring
     @cog       = cog
     @rim       = rim
     @tire      = tire
  end

  def gear_inches
    ratio * Wheel.new(rim, tire).diameter
  end
# ...
end

Gear.new(52, 11, 26, 1.5).gear_inches
```

Вполне очевидным последствием этой ссылки является то, что при изменении имени класса `Wheel` должен измениться и метод `gear_inches` класса `Gear`.

На первый взгляд эта зависимость кажется безобидной. Ведь если классу `Gear` требуется общение с классом `Wheel`, то где-нибудь и когда-нибудь должен быть создан новый экземпляр класса `Wheel`. Если сам класс `Gear` знает имя класса `Wheel`, то код в `Gear` должен измениться, если будет изменено имя класса `Wheel`.

По правде говоря, разобраться с изменением имени — не такая уж большая проблема. В вашем арсенале наверняка уже есть инструмент, позволяющий производить глобальную операцию поиска-замены внутри проекта. Если имя класса `Wheel` поменяется на `Wheely`, то найти и исправить все ссылки будет не так уж и сложно. Но тот факт, что строка 11 в показанном выше коде должна измениться, если изменится имя класса `Wheel`, является наименьшей из проблем данного кода. Есть более серьезная (и менее заметная) проблема, имеющая разрушительный характер.

Когда ссылка на `Wheel` в классе `Gear` жестко прописана глубоко внутри его метода `gear_inches`, она явным образом объявляет, что нужно лишь вычислить передаточное отношение в дюймах (gear inches) для экземпляров `Wheel`. Класс `Gear` отказывается от сотрудничества с любыми другими видами объекта, даже если у объекта есть диаметр и в нем используются передаточные механизмы.

Если ваше приложение расширяется с целью включения таких объектов, как диски или цилиндры, и вам требуется знать передаточное отношение в дюймах при их использовании, то вы этого сделать *не можете*. Независимо от факта наличия у дисков и цилиндров диаметра вы никогда не сможете вычислить их передаточное отношение в дюймах, поскольку класс `Gear` прикреплен к классу `Wheel`.

Приведенный выше код выставляет напоказ ничем не оправданное вложение в статичные типы. Важен не класс объекта, а *сообщение*, которое планируется ему отправить. Классу `Gear` нужен доступ к объекту, который может реагировать на сообщение `diameter`; если хотите, неявная типизация (см. главу 5). Классу `Gear` все равно, к какому классу принадлежит объект, и он не должен этого знать. Классу `Gear` не обязательно знать о существовании класса `Wheel`, чтобы вычислить `gear_inches`. Он не должен знать, что `Wheel` ожидает инициализации сначала с использованием `rim`, а затем с использованием `tire`; ему нужен лишь объект, знающий `diameter`.

Навешивание этих ненужных зависимостей на `Gear` попутно снижает возможность повторного использования `Gear` и увеличивает его восприимчивость к принудительному необязательному изменению. Когда класс `Gear` знает слишком много о *других* объектах, он становится менее полезным, а зная меньше, он способен на большее.

Вместо того чтобы быть приклеенной к `Wheel`, следующая версия `Gear` ожидает инициализации с помощью объекта, способного откликнуться на `diameter`:

```
class Gear
  attr_reader :chainring, :cog, :wheel
  def initialize(chainring, cog, wheel)
     @chainring = chainring
     @cog       = cog
     @wheel     = wheel
  end

  def gear_inches
    ratio * wheel.diameter
  end
# ...
end

# Gear нужна «утка», знающая diameter
Gear.new(52, 11, Wheel.new(26, 1.5)).gear_inches
```

Чтобы хранить этот объект в `Gear`, используется переменная `@wheel`, а для доступа к нему — метод `wheel`, но не заблуждайтесь: `Gear` не знает и не нуждается в понимании того, что этот объект может быть экземпляром класса `Wheel`. Класс `Gear` знает только то, что он хранит в себе объект, откликающийся на `diameter`.

Это изменение настолько незначительно, что его почти не видно, но программирование в таком стиле имеет огромные преимущества. Вывод создания нового экземпляра `Wheel` за пределы класса `Gear` устраняет связь между двумя классами. `Gear` теперь может сотрудничать с любым объектом, в котором реализуется `diameter`. В качестве дополнительного бонуса: это преимущество было получено без каких-либо затрат. Не было написано ни одной дополнительной строки кода, связь была устранена путем перестановки, сделанной в уже существующем коде.

Данная технология известна как *внедрение зависимости*. Несмотря на ее грозную репутацию, во внедрении зависимости нет ничего сложного. В `Gear` до этого были явно обозначенные зависимости от класса `Wheel`, а также от типа и порядка следования аргументов его инициализации, но через внедрение эти зависимости были сведены к единственной зависимости от метода `diameter`. Класс `Gear` стал более рациональным, поскольку теперь он меньше знает.

Использование внедрения зависимости для формирования кода зависит от вашей способности распознать, что обязанность знания имени класса и обязанность знания имени сообщения, отправляемого этому классу, могут располагаться

в разных объектах. Только то, что `Gear` нужно отправить куда-то сообщение `diameter`, еще не значит, что `Gear` должен знать о `Wheel`.

Остается вопрос о том, где именно находится обязанность знать о реальном классе `Wheel`; в показанном выше примере эту проблему удалось удачно обойти, но ее более подробное рассмотрение в этой главе вскоре последует. А пока нам достаточно понять, что это знание классу `Gear` уже не принадлежит.

Изоляция зависимостей

Конечно, лучше всего ликвидировать все ненужные зависимости, но, к сожалению, несмотря на имеющуюся техническую возможность, реальной возможности может и не быть. На вашу работу с уже существующим приложением могут влиять ограничения, накладываемые на объем возможных фактических изменений. Если нельзя достичь совершенства, вашей задачей должно стать оздоровление общей ситуации с целью оставить код в лучшем состоянии, чем он был до вашего вмешательства.

Следовательно, если вы не можете удалить ненужные зависимости, их придется изолировать внутри класса. В главе 2 вы уже занимались изоляцией обязанностей, чтобы их было легче распознать и удалить, как только для этого сложатся благоприятные условия; здесь же вам следует изолировать ненужные зависимости, чтобы их было легче заметить и удалить, как только обстоятельства позволят это сделать.

Воспринимайте любую зависимость как бактерию, пытающуюся инфицировать ваш класс. Предоставьте классу иммунную систему и поместите каждую зависимость в карантин. Зависимости сродни иностранным захватчикам, и их следует выявлять и изолировать.

Изоляция при создании экземпляра

Если накладываются ограничения, не позволяющие изменить код с целью внедрения `Wheel` в `Gear`, вам нужно изолировать создание нового экземпляра `Wheel` внутри класса `Gear`. Ваша задача — явно обозначить зависимость и при этом снизить степень ее вмешательства в ваш класс.

Этот замысел будет проиллюстрирован в следующих двух примерах кода.

В первом примере создание нового экземпляра `Wheel` было перемещено из принадлежащего классу `Gear` метода `gear_inches` в метод инициализации `Gear`. Тем самым очищается метод `gear_inches`, а зависимость открыто обозначается

в методе `initialize`. Обратите внимание на то, что при создании нового экземпляра `Gear` этот метод всякий раз создает и новый экземпляр `Wheel`.

```
 class Gear
   attr_reader :chainring, :cog, :rim, :tire
   def initialize(chainring, cog, rim, tire)
     @chainring = chainring
     @cog       = cog
     @wheel     = Wheel.new(rim, tire)
   end

   def gear_inches
    ratio * wheel.diameter
   end
# ...
```

В следующем варианте изоляции создание нового экземпляра `Wheel` для внутреннего пользования определяется явным образом в методе `wheel`. Этот новый метод создает новый экземпляр `Wheel` в ленивом стиле, то есть только при необходимости, для чего используется Ruby-оператор `||=`. В данном случае создание нового экземпляра `Wheel` откладывается до того момента, когда метод `gear_inches` вызовет новый метод `wheel`.

```
class Gear
  attr_reader :chainring, :cog, :rim, :tire
  def initialize(chainring, cog, rim, tire)
     @chainring = chainring
     @cog = cog
     @rim = rim
     @tire = tire
  end

  def gear_inches
    ratio * wheel.diameter
  end

  def wheel
    @wheel ||= Wheel.new(rim, tire)
  end
# ...
```

И в том и в другом примере `Gear` все еще слишком осведомлен о многом; он по-прежнему принимает в качестве аргументов инициализации `rim` и `tire`, по-прежнему создает свой собственный экземпляр `Wheel`. Класс `Gear`, как и раньше, привязан к классу `Wheel`; он не может вычислить передаточное отношение в дюймах ни для какой другой разновидности объекта.

Но *появилось* и улучшение. Такие стили программирования сокращают количество зависимостей в `gear_inches` и наряду с этим выставляют на всеобщее обозрение зависимость `Gear` от `Wheel`. Они показывают зависимости вместо того, чтобы скрывать их, облегчая повторное использование и упрощая реорганизацию кода при стечении благоприятных для этого обстоятельств. Такое изменение делает код гибче, и его будет проще приспособить под неизвестное пока будущее.

Способ управления зависимостями от имен внешних классов сильно влияет на ваше приложение. Если вы помните о зависимостях и выработали привычку регулярно добавлять их, ваши классы получат вполне естественную слабую связанность. Не стоит игнорировать эту проблему и позволять ссылкам на классы размещаться где попало, иначе ваше приложение будет мало похоже на набор независимых объектов. Приложение, чьи классы пестрят запутанными и непонятными ссылками на имена классов, получаются громоздкими и негибкими, а те приложения, чьи зависимости от имен классов отличаются краткостью, явной выраженностью и изолированностью, подогнать под новые требования намного проще.

Изоляция внешних сообщений, создающих уязвимости

После изолирования ссылок на имена внешних классов пора уделить внимание внешним *сообщениям*, то есть сообщениям, которые «отправлены кому-то другому, но не `self`». Например, показанный ниже метод `gear_inches` отправляет `ratio` и `wheel` в адрес `self`, а вот `diameter` отправляет в адрес `wheel`:

```
1  def gear_inches
2    ratio * wheel.diameter
3  end
```

Этот метод весьма прост и содержит единственную имеющуюся в `Gear` ссылку на `wheel.diameter`. В данном случае с кодом все в порядке, но ситуация может усложниться. Представим себе, что вычисление `gear_inches` требует куда боль-

шего объема математических вычислений и метод имеет примерно следующий вид:

```
def gear_inches
  #... несколько строк сплошной математики
  foo = some_intermediate_result * wheel.diameter
  #... еще несколько строк сплошной математики
end
```

Теперь фрагмент `wheel.diameter` встроен глубоко внутрь сложного метода, который зависит от того, как `Gear` ответит на `wheel`, и от того, как `wheel` ответит на `diameter`. Во встраивании этой внешней зависимости внутрь метода `gear_inches` нет никакой необходимости, к тому же оно повышает уязвимость кода.

При *любом* изменении есть риск что-либо нарушить; теперь `gear_inches` сложный метод, что не только повышает вероятность возникновения необходимости внесения в него изменений, но и делает его более восприимчивым к возможным повреждениям в ходе этих изменений. Как показано в следующем примере, вы можете снизить вероятность возникновения потребности в вынужденном внесении изменений в `gear_inches` путем удаления внешней зависимости и инкапсуляции в ее собственном методе:

```
def gear_inches
  #... несколько строк сплошной математики
  foo = some_intermediate_result * diameter
  #... еще несколько строк сплошной математики
end

def diameter
  wheel.diameter
end
```

Новый метод `diameter` и есть тот самый метод, который нужно написать, если есть множество ссылок на `wheel.diameter`, разбросанных по классу `Gear`, и если вы стремитесь их удалить, следуя DRY-принципу. Вопрос заключается в выборе сроков; как правило, вполне оправданно отложить создание метода `diameter` до той поры, когда появится потребность в пересмотре всего кода на предмет соответствия DRY-принципу, но в данном случае метод создается превентивно для удаления зависимости из `gear_inches`.

В исходном коде в `gear_inches` известно, что у `wheel` есть `diameter`. Это знание является опасной зависимостью, привязывающей `gear_inches` к внешнему объекту и к одному из *его* методов. После этого изменения метод `gear_inches` стал более абстрактным. Теперь `Gear` изолирует `wheel.diameter` в отдельном методе, и `gear_inches` может зависеть от сообщения, отправленного в классе самому себе.

Если в классе `Wheel` изменится имя или сигнатура *его* реализации метода `diameter`, побочные эффекты, влияющие на `Gear`, ограничатся этим одним простым методом, играющим роль оболочки.

Необходимость в этой технологии возникает, когда в классе содержатся встроенные ссылки на *сообщения*, вероятность предстоящих изменений которых довольно высока. Изоляция ссылки дает некоторые гарантии того, что эти изменения не окажут отрицательного влияния. Хотя не каждый внешний метод является кандидатом на подобную упреждающую изоляцию, стоит все же изучить свой код на предмет изоляции зависимостей, сулящих наибольшую уязвимость.

Альтернативный способ исключения таких побочных эффектов — обойти проблемы стороной с самого начала путем изменения направления зависимости. Вскоре мы займемся этим, но сначала рассмотрим еще один прием программирования.

Устранение зависимостей от порядка следования аргументов

Когда отправляется сообщение, требующее аргументов, вы, как отправитель, не можете обойтись без знаний этих аргументов. Такая зависимость неизбежна. Но передача аргументов зачастую сопряжена со второй, менее заметной зависимостью. Многие сигнатуры методов требуют не только аргументов, но и их отправки в фиксированном порядке.

В следующем примере метод `initialize` класса `Gear` получает три аргумента: `chainring`, `cog` и `wheel`. Значения по умолчанию не предоставляются; обязательным является каждый из этих аргументов. В строках 11–14, когда создается новый экземпляр `Gear`, эти три аргумента должны быть переданы и при этом следовать *в правильном порядке*.

```
1  class Gear
2    attr_reader :chainring, :cog, :wheel
3    def initialize(chainring, cog, wheel)
```

```
4       @chainring = chainring
5       @cog = cog
6       @wheel = wheel
7     end
8     ...
9   end
10
11  Gear.new(
12   52,
13   11,
14   Wheel.new(26, 1.5)).gear_inches
```

Отправители `new` зависят от порядка следования аргументов, указанного в методе `initialize` класса `Gear`. Если этот порядок изменится, все отправители будут также вынуждены вносить изменения.

К сожалению, перестановки аргументов инициализации случаются довольно часто. На ранних стадиях разработки, когда конструкция еще не обрела четких форм, вы можете пройти через несколько циклов добавления и удаления аргументов и значений по умолчанию. Если используются аргументы с жестко заданной последовательностью, каждый из таких циклов может потребовать изменений во многих зависимостях. Хуже того, может оказаться, что вы сторонитесь изменений аргументов, даже если того требуют задачи проектирования, потому что вам совершенно не хочется еще раз переделывать все зависимости.

Использование хешей для аргументов инициализации

Есть простой способ, позволяющий обойти зависимость от фиксированного порядка следования аргументов. Если вы в состоянии управлять методом `initialize` класса `Gear`, измените код так, чтобы он получал не фиксированный список параметров, а хеш параметров.

Простая версия этой технологии показана в следующем примере. Теперь метод `initialize` получает только один аргумент — `args`, являющийся хешем, содержащим все вводимые данные. Метод был изменен таким образом, чтобы он мог извлечь свои аргументы из этого хеша. Сам хеш создается кодом в строках 11–14.

```
1   class Gear
2     attr_reader :chainring, :cog, :wheel
3     def initialize(args)
4       @chainring = args[:chainring]
```

```
5       @cog       = args[:cog]
6       @wheel     = args[:wheel]
7     end
8     ...
9   end
10
11  Gear.new(
12    :chainring => 52,
13    :cog => 11,
14    :wheel => Wheel.new(26, 1.5)).gear_inches
```

Данная технология имеет ряд преимуществ. Первым и наиболее очевидным является то, что она удаляет любые зависимости от порядка следования аргументов. Теперь в `Gear` можно добавлять и удалять аргументы и значения по умолчанию, оставаясь уверенными, что никакие изменения не повлекут за собой побочные эффекты, проявляемые в другом коде.

Эта технология более многословна. Во многих ситуациях многословие вредит, но в данном случае идет во благо. Многословие находится на пересечении насущных потребностей с неизвестностью будущего. Использование аргументов, следующих в фиксированном порядке, уменьшает объем кода сегодня, но вы платите за это повышением риска того, что изменения в будущем потребуют целую череду изменений у тех, кто от этого кода зависит.

Когда код в строке 11 изменился с целью использования хеша, он утратил свою зависимость от порядка следования аргументов, но приобрел зависимость от имен ключей в аргументном хеше. Это изменение во здравие. Новая зависимость устойчивее прежней, поэтому код теперь меньше подвержен риску попадания под вынужденные изменения. Возможно, для вас это станет неожиданностью, но хеш предоставляет еще одно новое вторичное преимущество: имена *ключей* в хеше являются явно выраженной документацией аргументов. Это побочный продукт использования хеша, но тот факт, что все получилось непреднамеренно, ничуть не умаляет получаемой от него пользы. Те, кто будет сопровождать код в будущем, станут испытывать благодарность за эту информацию.

Преимущества, получаемые от использования данной технологии, как всегда, варьируются в зависимости от вашей личной ситуации. Если вы работаете над методом, имеющим довольно длинный список параметров и пока что не отличающимся стабильностью, то в той среде, которая предназначена для использования другими разработчиками, указание аргументов в хеше, скорее всего, снизит общие затраты. Но если метод, выполняющий деление двух чисел, соз-

дается для собственного пользования, то куда легче и в конечном итоге дешевле будет просто передать аргументы и смириться с зависимостью от порядка их следования. Между этими двумя крайностями находится ситуация, когда методу требуются несколько весьма стабильных аргументов и, возможно, допускается наличие нескольких менее стабильных аргументов. В таком случае наименее затратная стратегия может заключаться в использовании обеих технологий, то есть принимать несколько аргументов в фиксированном порядке, за которыми последует хеш параметров.

Явное определение значений по умолчанию

Для добавления значений по умолчанию существует множество технологий. Простые, не относящиеся к булевым значения по умолчанию могут быть указаны, как в следующем примере, с помощью Ruby-метода ||:

```
# указание значений по умолчанию с помощью ||
def initialize(args)
  @chainring = args[:chainring] || 40
  @cog       = args[:cog]       || 18
  @wheel     = args[:wheel]
end
```

Эта технология получила широкое распространение, но все же пользоваться ею следует осмотрительно; бывают ситуации, когда ее действия могут не соответствовать желаемым. Метод || работает как условие ИЛИ. Сначала он вычисляет левостороннее выражение, а затем, если выражение возвращает `false` или `nil`, приступает к вычислению и возвращению результата правостороннего выражения. Следовательно, использование метода || в вышеприведенном коде полагается на тот факт, что метод [] класса `Hash` возвратит для отсутствующих ключей значение `nil`.

В том случае, когда `args` содержит ключ `:boolean_thing` со значением по умолчанию `true`, использование метода || делает для вызывающего кода невозможным явно установить для конечной переменной значение `false` или `nil`. Например, следующее выражение устанавливает для `@bool` значение `true`, когда ключ `:boolean_thing` отсутствует или когда он имеется, но установлен в `false` или `nil`:

```
@bool = args[:boolean_thing] || true
```

Эта особенность метода || означает, что если брать в качестве аргументов булевы значения или брать аргументы, для которых нужно отличать `false` от `nil`,

то лучше для установки значений по умолчанию использовать метод `fetch`. Этот метод *ожидает*, что извлекаемый ключ находится в хеше, и предоставляет несколько вариантов для явной обработки отсутствующих ключей. Его преимущество по сравнению с методом `||` заключается в том, что он не возвращает автоматически значение `nil`, когда не может найти ваш ключ.

В следующем примере в строке 3 метод `fetch` используется для присваивания переменной `@chainring` значения по умолчанию `40` только в том случае, если ключ `:chainring` отсутствует в хеше `args`. Установка значений по умолчанию таким способом означает, что вызывающий код действительно может заставить переменную `@chainring` получить установку на `false` или `nil`, что невозможно сделать при использовании технологии, основанной на использовании метода `||`.

```
1  # указание значений по умолчанию с помощью fetch
2  def initialize(args)
3    @chainring = args.fetch(:chainring, 40)
4    @cog       = args.fetch(:cog, 18)
5    @wheel     = args[:wheel]
6  end
```

Можно также полностью удалить значения по умолчанию из `initialize` и изолировать их внутри отдельного метода-оболочки. Показанный ниже метод `defaults` определяет второй хеш, который в ходе инициализации поглощается хешем параметров. В данном случае метод `merge` действует так же, как и метод `fetch`; значения по умолчанию будут поглощены, только если их ключей нет в хеше.

```
1  # указание значений по умолчанию поглощением хэша значений по умолчанию
2  def initialize(args)
3    args       = defaults.merge(args)
4    @chainring = args[:chainring]
5    # ...
6  end
7
8  def defaults
9    {:chainring => 40, :cog => 18}
10 end
```

Эта технология изоляции для показанного выше случая вполне приемлема, но она особенно полезна, когда значения по умолчанию имеют более сложную природу. Если ваши значения по умолчанию не простые числа или строки, реализуйте метод `defaults`.

Изоляция инициализации, при которой используются множественные параметры

До сих пор все примеры избавления от зависимости, связанной с порядком следования аргументов, применялись к ситуациям, когда *вы* контролировали сигнатуру методов, которая нуждалась в изменении. Но подобную роскошь вы можете позволить себе далеко не всегда; порой придется зависеть от метода, требующего аргументов с фиксированным порядком следования, когда этот метод вам не принадлежит (поэтому сам метод вы изменить не в состоянии).

Представим, что `Gear` является частью среды, в силу чего его метод инициализации требует аргументов, следующих в фиксированном порядке. Представим также, что в вашем коде имеется множество мест, где нужно создавать новые экземпляры `Gear`. Принадлежащий `Gear` метод `initialize` — *внешний* по отношению к вашему приложению и является частью неконтролируемого вами внешнего интерфейса.

При всей плачевности ситуации не стоит мириться с зависимостями. Точно так же, как, следуя принципу DRY, вы избавляетесь от повторяющегося кода внутри класса, можно применить DRY-очистку для создания новых экземпляров `Gear` путем определения единого метода, выступающего в роли оболочки внешнего интерфейса. Классы в вашем приложении должны зависеть от вашего собственного кода, а для изоляции внешних зависимостей следует применить метод-оболочку.

В следующем примере класс `SomeFramework::Gear` не принадлежит вашему приложению. Он является частью внешней среды. Его метод инициализации требует фиксированного порядка следования аргументов. Чтобы избежать множественных зависимостей от порядка следования этих аргументов, был создан модуль `GearWrapper`, в котором изолируются все сведения о внешнем интерфейсе и который предоставляет усовершенствованный интерфейс для вашего приложения.

В строке 24 можно увидеть, что `GearWrapper` позволяет вашему приложению создать новый экземпляр `Gear`, используя хеш параметров.

```
1  # Тот самый случай, когда Gear является частью внешнего интерфейса
2  module SomeFramework
3  class Gear
4      attr_reader :chainring, :cog, :wheel
5      def initialize(chainring, cog, wheel)
6        @chainring = chainring
7        @cog       = cog
8        @wheel     = wheel
```

```
 9      end
10    # ...
11    end
12  end
13
14  # заключение интерфейса в оболочку для вашей защиты от изменений
15  module GearWrapper
16    def self.gear(args)
17      SomeFramework::Gear.new(args[:chainring],
18      args[:cog],
19      args[:wheel])
20    end
21  end
22
23  # Теперь новый экземпляр Gear можно создавать, используя хеш аргументов
24  GearWrapper.gear(
25    :chainring => 52,
26    :cog       => 11,
27    :wheel     => Wheel.new(26, 1.5)).gear_inches
```

По поводу `GearWrapper` следует отметить две особенности. Во-первых, это не класс, а Ruby-модуль (строка 15). `GearWrapper` отвечает за создание новых экземпляров `SomeFramework::Gear`. Использование модуля позволяет определить отдельный особый объект, которому можно отправить сообщение `gear` (строка 24), одновременно сообщая, что вы не ожидаете наличия экземпляров `GearWrapper`. У вас уже может быть опыт включения модулей в классы; в показанном выше примере `GearWrapper` предназначен не для включения в другой класс, а для непосредственного ответа на сообщение `gear`.

Еще одна интересная особенность `GearWrapper` — то, что его единственной целью является создание экземпляров некоего другого класса. Для подобных объектов у проектировщиков объектно-ориентированных решений есть особое слово, они называют их *фабриками* (factories). В определенных кругах термин «фабрика» приобрел негативный оттенок, но здесь он лишен этой окраски. Объект, чья цель — создание других объектов, является фабрикой, ничего иного это слово не означает и наиболее точно отражает саму идею.

Показанная выше технология подстановки хеша параметров для списка аргументов, следующих в фиксированном порядке, идеально подходит для тех случаев, когда вы вынужденно находитесь в зависимости от внешних интерфейсов, которые не в состоянии изменить. Не позволяйте подобного рода зависимостям пронизывать ваш код; защититесь от них, заключая каждую зависимость в метод-оболочку, принадлежащий вашему собственному приложению.

Управление направлением зависимостей

У зависимостей всегда есть направление. Ранее в этой главе уже было высказано предположение, что одним из способов управления зависимостями является смена направления (на обратное). В этом разделе мы поговорим о том, как принять решение о направлении зависимостей.

Разворот в обратном направлении

В каждом использованном до сих пор примере класс `Gear` зависим от `Wheel` или `diameter`, но без особого труда можно создать код с обратным направлением зависимостей. `Wheel` может вместо этого зависеть от `Gear` или `ratio`. В следующем примере показана возможная форма реверса зависимостей. В нем в класс `Wheel` были внесены изменения, чтобы он зависел от `Gear` и `gear_inches`. Класс `Gear` по-прежнему отвечает за вычисление, но ожидает, что вызывающим кодом ему будет передан аргумент `diameter` (строка 8).

```
 1 class Gear
 2   attr_reader :chainring, :cog
 3   def initialize(chainring, cog)
 4      @chainring = chainring
 5      @cog       = cog
 6   end
 7
 8   def gear_inches(diameter)
 9      ratio * diameter
10   end
11
12   def ratio
13     chainring / cog.to_f
14   end
15 # ...
16 end
17
18 class Wheel
19   attr_reader :rim, :tire, :gear
20   def initialize(rim, tire, chainring, cog)
21     @rim  = rim
22     @tire = tire
23     @gear = Gear.new(chainring, cog)
```

```
  end

 def diameter
   rim + (tire * 2)
 end

 def gear_inches
   gear.gear_inches(diameter)
 end
# ...
end

Wheel.new(26, 1.5, 52, 11).gear_inches
```

Реверс зависимостей не наносит вреда. Вычисление `gear_inches` по-прежнему требует сотрудничества классов `Gear` и `Wheel`, а на результат вычисления реверс не оказывает никакого влияния. Можно прийти к выводу, что направление зависимости не имеет значения, поскольку все равно, что от чего зависит: `Gear` от `Wheel` или наоборот.

Несомненно, для приложения, которое никогда не изменяется, неважно, какой вы сделаете выбор. Однако ваше приложение будет изменяться, и сегодняшнее решение будет иметь последствия в постоянно меняющемся будущем. Ваш выбор относительно направления зависимостей имеет далеко идущие последствия, проявляющиеся в жизненном цикле вашего приложения. Если все будет сделано правильно, с вашим приложением будет приятно работать и его будет легко сопровождать. Если сделать неверный выбор, то со временем вносить изменения в приложение будет все труднее.

Выбор направления

Представьте, что ваши классы — это люди. Если бы вам пришлось посоветовать им, как себя вести, то вы бы сказали: нужно *действовать в зависимости от того, что меняется реже вас*.

Это краткое утверждение противоречит идее, основанной на трех простых истинах, касающихся программного кода:

- некоторые классы более других подвержены изменениям своих требований;
- конкретные классы имеют более высокую вероятность изменений, чем абстрактные классы;

- внесение изменения в класс, имеющий множество зависимостей, повлечет за собой широкое распространение последствий.

Кое-где эти истины пересекаются, но каждая из них является отдельным понятием.

Понимание высокой степени вероятности изменений. Утверждение, что некоторые классы имеют более высокую степень вероятности изменений, чем другие классы, применимо не только к создаваемому вами коду для собственного приложения, но и к тому коду, который вы используете, но не создаете. У базовых классов Ruby и у другого кода, относящегося к среде, на которую вы полагаетесь, есть своя собственная, присущая им степень вероятности изменений.

Вам повезло, что базовые классы Ruby изменяются гораздо реже вашего собственного кода. Поэтому вполне разумно зависеть от метода `*`, как это и происходит с `gear_inches`, или же ожидать, что Ruby-классы `String` и `Array` будут продолжать работать так же, как и раньше.

Иное дело классы среды — только вам решать, насколько устоявшейся является та среды, в которой вы работаете. В целом же любая используемая вами среда будет более стабильной, чем создаваемый вами код, но, разумеется, можно выбрать и такую среду, которая переживает бурное развитие и код которой изменяется чаще вашего.

Независимо от происхождений каждый класс, используемый в вашем приложении, можно оценить по шкале вероятности изменений относительно всех других классов. Эта оценка — одна из основных информационных составляющих, рассматриваемых при выборе направления зависимостей.

Определение конкретности и абстрактности

Понятие абстрактности используется нами в точном соответствии с определением толкового словаря Мерриам-Уэбстер: «Отсутствием чего-либо общего с любым конкретным экземпляром» — и, как и многое в Ruby, дает представление о коде, а не о конкретных технических ограничениях.

Это понятие уже было проиллюстрировано ранее в данной главе (в разделе, посвященном внедренным зависимостям). Когда `Gear` зависит от `Wheel`, и от `Wheel.new`, и от `Wheel.new(rim, tire)`, этот класс зависит от сугубо конкретного кода. После изменения кода, имевшего целью внедрить `Wheel` в `Gear`, класс `Gear`

внезапно стал зависеть от более абстрактного кода, то есть от факта, что у него имеется доступ к объекту, откликающемуся на сообщение `diameter`.

Знакомство с Ruby может создать у вас впечатление, что этот переход можно воспринять как сам собой разумеющийся, но представьте на минутку, что потребовалось бы для выполнения точно такого же приема в языке со статической типизацией. Поскольку у статически типизированных языков имеются компиляторы, работающие в отношении типов наподобие блочных тестов, у вас может не получиться внедрить в `Gear` какой-либо произвольный объект. Вместо этого придется объявлять *интерфейс*, определять `diameter` в качестве части этого интерфейса, включать интерфейс в класс `Wheel` и сообщать `Gear`, что внедряемый класс является *разновидностью* этого интерфейса.

Ruby-программисты не зря благодарны языку за возможность избежать лишней работы, но те языки, которые заставляют явно показывать это преобразование, дают особое преимущество. Определение абстрактного интерфейса происходит в них вполне очевидным, неотвратимым и нелегким образом. Создать абстракцию по недоразумению или случайно совершенно невозможно; в статически типизированных языках определение интерфейса *всегда* происходит намеренно.

В Ruby, когда вы внедряете `Wheel` в `Gear` таким образом, что потом `Gear` зависит от той самой *«утки»*, отвечающей за `diameter`, вы (хотя и непреднамеренно) определяете интерфейс. Этот интерфейс является абстракцией идеи о том, что у определенной категории вещей будет иметься диаметр. Абстракция была получена из конкретного класса, и теперь идея характеризуется «отсутствием чего-либо общего с любым конкретным экземпляром».

Самое замечательное свойство абстракций заключается в том, что они представляют собой общие стабильные качества. Их изменение менее вероятно, чем изменение конкретных классов, из которых они были выведены. Зависимости от абстракции всегда безопаснее, чем зависимость от конкретики, поскольку по своей природе абстракции более стабильны. Ruby не заставляет вас явно объявлять абстракции, чтобы определить интерфейс, но для целей проектирования вы можете вести себя так, будто ваш виртуальный интерфейс так же реален, как и класс. Более того, при дальнейшем рассмотрении данного вопроса термин «класс» будет означать и класс, и данную разновидность интерфейса. Эти интерфейсы могут иметь зависимости, поэтому должны браться в расчет при проектировании.

Избегайте использования классов, обремененных зависимостями

Если у объекта множество зависимостей, это влечет за собой множество последствий. Последствия изменения классов, обремененных зависимостями, вполне очевидны, но не столь очевидны последствия простого наличия таких классов. Класс, который при изменении вызывает целую волну правок в приложении, будет противиться каким-либо изменениям. Ваше приложение будет постоянно испытывать ограничения, связанные с нежеланием расплачиваться за изменение, вносимое в этот класс.

Выявление проблемных зависимостей

Представим, что согласно каждому из вышеперечисленных утверждений создалась сплошная среда, в которой возникают условия для сбоя всего кода приложения. Классы отличаются друг от друга степенью вероятности их изменений, своим уровнем абстрагированности и количеством зависимостей. Каждое из этих качеств имеет весомое значение, но наиболее интересные проектировочные решения возникают, когда вероятность изменений пересекается с количеством зависимостей. Некоторые возможные комбинаций идут во благо, а некоторые могут нанести вред вашему приложению.

Описание возможностей представлено на рис. 3.2.

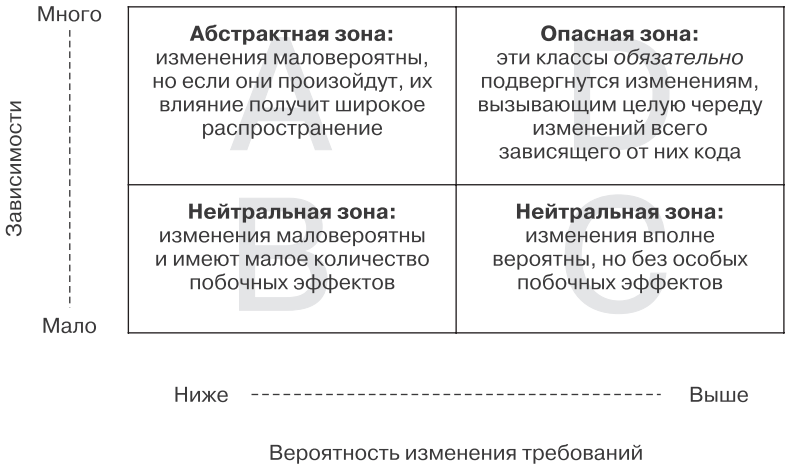

Рис. 3.2. Вероятность изменений в сопоставлении с количеством зависимостей

На горизонтальной оси представлена вероятность изменения требований. А на вертикальной — количество зависимостей. Сетка разбита на четыре зоны, помеченные буквами от A до D. Если дать оценку всем классам в качественно спроектированном приложении и поместить их в ячейки этой сетки, они будут собраны в зонах A, B и C.

Классы, имеющие малую вероятность изменений, но содержащие множество зависимостей, попадут в зону A. В этой зоне обычно находятся абстрактные классы или интерфейсы. В приложении, созданном на основе всесторонне продуманного проекта, подобное размещение классов неизбежно; зависимости группируются вокруг абстракций, потому что они имеют меньшую вероятность изменений.

Учтите, что классы не становятся абстрактными только потому, что они относятся к зоне A; они попадают в эту зону именно потому, что они уже абстрактные. Их абстрактная природа делает их более стабильными и позволяет им безопасно приобретать множество зависимостей. Размещение в зоне A не дает классу гарантию абстрактности, но, конечно же, свидетельствует о том, что ей следовало бы быть.

Пропустим на время зону B, потому что противоположностью зоны A является зона C. В этой зоне содержится код с высокой степенью вероятности изменений, но имеющий при этом мало зависимостей. Классы, попавшие в эту зону, имеют склонность к более высокой степени конкретизации, что придает им более высокую степень вероятности изменений, но это не имеет значения, поскольку от них зависит весьма небольшое количество других классов.

Классы в зоне B вызывают наименьшее беспокойство при проектировании, поскольку они практически нейтральны по отношению к потенциальному влиянию тех ситуаций, которые могут сложиться в будущем. Они крайне редко подвергаются изменениям и имеют мало зависимостей. Зоны A, B и C являются вполне допустимыми участками для кода, а вот зона D названа опасной неспроста. Классы попадают в зону D в том случае, когда у них гарантированно высокая степень вероятности будущих изменений и много зависимостей. Изменения классов из зоны D обходятся слишком дорого; простые запросы превращают программирование в ночной кошмар, поскольку их реализация выливается в целую череду изменений в каждой зависимости. Если у вас имеется весьма специфичный конкретный класс, обладающий множеством зависимостей, и вы полагаете, что место ему в зоне A, то есть верите в низкую вероятность его изменений, подумайте еще раз. Когда у конкретного класса множество зависимостей, вы должны бить тревогу. Фактически этот класс может быть обитателем зоны D.

Классы из зоны D существенно затрудняют изменение приложения. Когда простое изменение вызывает череду эффектов, требующих внесения других изменений, класс из зоны D становится корнем проблемы. Если нарушения, вызванные изменением, распространяются слишком далеко на тот код, который, казалось бы, не имеет к нему никакого отношения, то именно здесь кроется изъян проектирования. Как бы удручающе это ни звучало, так вы лишь усугубите ситуацию. Можно гарантировать, что любое приложение постепенно станет неуправляемым, потому что его классы из зоны D имеют более высокую степень вероятности изменений, чем те классы, которые от них зависят. При этом последствия каждого изменения становятся максимально неблагоприятными.

К счастью, усвоив суть всех этих фундаментальных проблем, можно упредить и обойти ту или иную сложную ситуацию.

Разграничение на зоны будет весьма удобным способом выстраивания ваших размышлений, но в запарке не всегда можно сразу разобраться в том, какие классы к какой зоне относятся. Зачастую вы прокладываете свой собственный путь в проектировании, и в каждый отдельно взятый момент времени будущее представляется весьма туманным. Если следовать этому простому правилу зональности при каждом удобном случае, то ваше приложение постепенно превратится во вполне жизнеспособную конструкцию.

Выводы

Управление зависимостями — основа создания приложений, способных выдержать всевозможные изменения в будущем. Внедрение зависимостей создает объекты со слабой связью, пригодные для повторного использования новыми способами. Изоляция зависимостей позволяет объектам быстро приспосабливаться к неожиданным изменениям. А зависимость от абстракций снижает вероятность подобных изменений. Ключом к управлению зависимостями служит контролирование их направления.

Глава 4
Создание гибких интерфейсов

Об объектно-ориентированных приложениях принято думать как о совокупности классов. Классы можно четко выделить в программе, и проектировочные дискуссии зачастую разворачиваются вокруг обязанностей и зависимостей классов. Именно классы вы видите в своем текстовом редакторе, их наличие проверяется в хранилище исходного кода.

Но объектно-ориентированное приложение — это не только классы, есть и другие не менее важные особенности конструкции, которым следует уделить внимание. Они *состоят из классов*, но *определяются* сообщениями. Классы управляют тем, что находится в хранилище исходного кода, а сообщения отражают живое, динамичное приложение.

Следовательно, при проектировании нужно заниматься сообщениями, передаваемыми между объектами. Здесь речь идет не только о том, что знают объекты (об их обязанностях) и кого они знают (об их зависимостях), но и о том, как они общаются друг с другом. «Разговор» между объектами ведется с помощью их *интерфейсов*, и в этой главе будет рассмотрен вопрос создания гибких интерфейсов, позволяющих приложениям расти и изменяться.

Основные сведения об интерфейсах

Представьте себе два запущенных приложения, показанных на рис. 4.1. Каждое из них состоит из объектов и сообщений, передаваемых между ними.

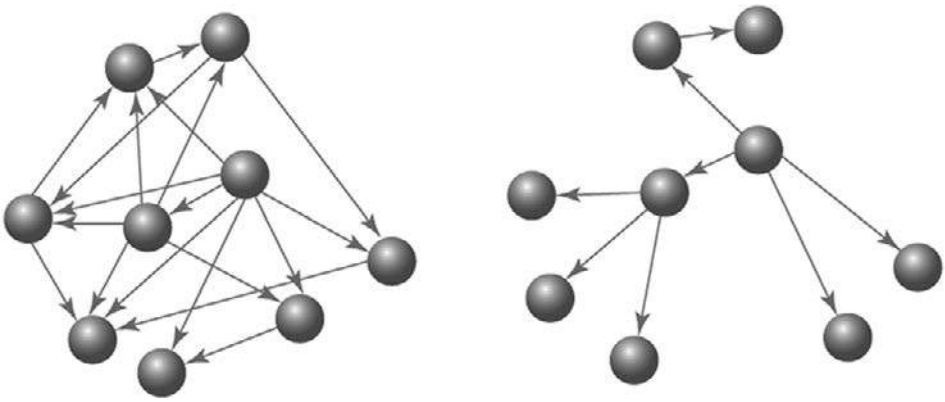

Рис. 4.1. Коммуникационные модели

В первом приложении сообщения не следуют четкой модели. Каждый объект может отправить любое сообщение любому другому объекту. Если бы сообщения оставляли видимые следы, то в конечном итоге из них был бы соткан своеобразный коврик, нити которого показывали бы связь каждого объекта со всеми остальными объектами.

Во втором приложении у сообщений имеется четко определенная модель. Здесь объекты связываются по определенным маршрутам. Если бы эти сообщения оставляли видимые следы, то они бы сливались, создавая архипелаг с редкими мостами между островами.

Оба приложения можно охарактеризовать по моделям их сообщений.

Объекты в первом приложении будет трудно использовать повторно. Каждый из них слишком открыт для других и слишком широко осведомлен о своих соседях. Эта чрезмерная осведомленность приводит к тому, что объекты имеют весьма узкую специализацию, явную предназначенность и губительную настроенность на выполнение только тех действий, которыми они занимаются в данный момент. Ни один из объектов не стоит особняком, и, чтобы повторно

воспользоваться любым из них, вам нужны сразу все объекты, а при внесении одного изменения нужно изменять абсолютно все.

Второе приложение составлено из подключаемых объектов, похожих на компоненты. Каждый из них открывает для других минимум информации и знает о них такой же строго необходимый минимум.

Проблемы проектирования первого приложения могут быть и не связаны с отказом от внедрения зависимостей или от назначения единственной обязанности. Эти методы являются необходимым, но не достаточным условием для предотвращения создания приложения, конструкция которого осложняет вашу жизнь. Суть этой новой проблемы заключается не в том, что делает каждый класс, а в том, что он всем показывает. Каждый метод любого класса на вполне законных основаниях может быть вызван любым другим объектом.

Опыт подсказывает, что не все методы в классе одинаковы, некоторые имеют более общий характер или имеют по сравнению с другими больше шансов претерпеть изменения. По первому приложению этого незаметно. Оно позволяет всем методам любого объекта, независимо от широты их специализации, быть вызванными другими объектами.

Во втором приложении модель сообщений имеет явно выраженный ограниченный характер. В этом приложении действует некое соглашение о том, какие сообщения могут проходить между объектами. У каждого объекта имеется вполне определенный набор методов, на использование которого могут рассчитывать другие объекты.

Эти выставляемые напоказ методы образуют *открытый интерфейс* класса.

Слово *интерфейс* может иметь отношение к нескольким различным понятиям. В данном случае этот термин используется для той разновидности интерфейса, которая является классом. В классах реализуются методы, предназначенные для использования другими классами, именно такие методы и составляют открытый интерфейс класса.

К альтернативной разновидности интерфейса относится тот, который распространяется на несколько классов, в силу чего обладает независимостью от любого отдельно взятого класса. Используемое в этом смысле слово «*интерфейс*» олицетворяет набор сообщений (сами сообщения и определяют интерфейс). Методы, требуемые интерфейсу, могут как часть целого быть реализованы во многих различных классах. Это почти то же самое, как если бы интерфейсом определялся *виртуальный* класс; то есть любой класс, реализующий требуемые методы, может действовать как своеобразный *интерфейс*.

В остальной части этой главы будет рассматриваться интерфейс первой разновидности, то есть методы внутри класса, а также содержимое, выставляемое на показ другим объектам, и применяемые для этого способы. В главе 5 будет рассмотрена вторая разновидность интерфейса, которая представляет собой концепцию, выходящую за рамки отдельно взятого класса и определяется набором сообщений.

Определение интерфейсов

Представьте себе кухню ресторана. Клиенты заказывают блюда из меню. Заказы поступают на кухню через небольшое окошко (рядом с которым находится колокольчик «заказ готов!»), и через него же спустя время подается еда. На первый взгляд может показаться, будто кухня заставлена тарелками с едой в ожидании заказов, но на самом деле кухня заполнена поварами, продуктами и на ней кипит бурная деятельность — каждый заказ запускает новую конструкцию и процесс сборки.

На кухне кипит работа, которую посетители не видят. У кухни имеется *открытый* интерфейс, который вполне ожидаемо используется клиентами, — меню. На кухне происходит множество событий, проходит масса других сообщений, но эти сообщения являются *закрытыми* (они не видны клиентам). И когда клиенты заказывают суп, никто не ждет, что они придут, чтобы его помешивать.

Такая разница между открытым и закрытым существует потому, что это наиболее эффективный способ ведения дел. Если бы клиентам предписывалось готовить еду, то их нужно было бы переучивать каждый раз, когда на кухне не хватало бы продуктов и приходилось менять ингредиенты. Меню устраняет эту проблему, позволяя каждому клиенту заказать то, *что* ему нужно, ничего не зная о том, *как* это готовится на кухне.

Каждый из ваших классов похож на кухню. Класс существует для выполнения одной обязанности, но в нем реализуется множество методов. Эти методы различаются по размеру и степени детализации и варьируются (от основных методов, выставляющих напоказ основную обязанность класса, до небольших служебных методов, предназначенных для использования только внутри класса). Некоторые методы представляют меню для вашего класса и должны быть открытыми, другие же имеют дело с внутренней реализацией особенностей и являются закрытыми.

Открытые интерфейсы

Методы, составляющие открытый интерфейс класса, можно назвать внешними. Они:

- показывают его главную обязанность;
- ожидают вызова из других объектов;
- не изменяются ни с того ни с сего;
- безопасны для других объектов, зависящих от них;
- тщательно задокументированы в тестах.

Закрытые интерфейсы

Все остальные методы в классе являются частью закрытого интерфейса. Они:

- занимаются реализацией особенностей класса;
- не ожидают отправления другим объектам;
- могут изменяться по какой-либо причине;
- в случае зависимости от них не гарантируют никакой безопасности;
- могут даже не упоминаться в тестах.

Обязанности, зависимости и интерфейсы

Глава 2 была посвящена созданию классов, имеющих единственную обязанность — единственное назначение. Когда класс имеет единственное назначение, то род его занятий (его более конкретные обязанности) позволяет ему соответствовать этому назначению. Между заявлениями, которые могли делаться вами насчет этих более конкретных обязанностей и открытых методов класса, существует четкое соответствие. Разумеется, открытые методы должны читаться как описание обязанностей. Открытый интерфейс представляет собой обязательство, ясно выражающее обязанности класса.

В главе 3 говорилось о зависимостях. Ее основным посылом было то, что класс должен зависеть только от тех классов, которые изменяются гораздо реже его самого. Теперь, когда каждый класс разделен на открытую и закрытую части, идея зависимости от менее подвергаемых изменениям объектов применима и к методам *внутри* класса.

Открытые части класса относятся к стабильным частям, а закрытые — к изменяемым. Когда методы помечаются как открытые или закрытые, вы сообщаете пользователям вашего класса, от каких методов они могут безопасно зависеть. Когда ваши классы используют открытые методы других классов, вы верите в то, что эти методы будут стабильными. Когда вы принимаете решение попасть в зависимость от закрытых методов других классов, вы должны отдавать себе отчет, что полагаетесь на заведомо нестабильные вещи, повышая тем самым риск подверженности воздействию отдаленного и не связанного с вашими задачами изменения.

Поиск открытого интерфейса

Поиск и определение открытых интерфейсов является определенного рода искусством. Это задача проектирования, поскольку каких-либо устоявшихся на сей счет правил нет. Существует множество способов создания «достаточно качественных» интерфейсов, а вот издержки от «недостаточно качественных» интерфейсов поначалу могут быть не столь очевидными, затрудняя тем самым работу над ошибками.

Целью проектирования, как и всегда, является поддержка максимальной будущей гибкости в совокупности с написанием кода, соответствующего сегодняшним требованиям. Качественные открытые интерфейсы снижают затраты на непредвиденные изменения, а некачественные эти затраты повышают.

В данном разделе вводятся новое приложение, иллюстрирующее ряд эмпирических правил, касающихся интерфейсов, и новый инструмент, помогающий в их исследовании.

Пример приложения: компания, занимающаяся велотуризмом

Встречайте FastFeet, Inc., компанию, занимающуюся велотуризмом. FastFeet предлагает велопутешествия как по обычным, так и по горным маршрутам. FastFeet ведет дела по бумажной системе. На данный момент какая-либо автоматизация в компании полностью отсутствует.

Каждое путешествие, предлагаемое FastFeet, предполагает следование по определенному маршруту и может осуществляться несколько раз в году. У каждого путешествия существуют ограничения по количеству участников, также

требуется определенное количество сопровождающих, выступающих еще и в роли механиков.

Каждому маршруту присвоен рейтинг в соответствии с нагрузками на дыхательный аппарат человека. У горных велосипедных маршрутов имеется дополнительный рейтинг, отражающий техническую сложность. Чтобы понять, подходит ли путешествие клиентам, у них должна быть определенная группа здоровья и степень мастерства управления горным велосипедом.

Клиенты могут брать велосипеды в аренду или пользоваться своими собственными. В FastFeet есть несколько велосипедов, выдаваемых в аренду, также компания располагает общим фондом (с местными магазинами велосипедов) арендуемых велосипедов. Арендуемые велосипеды бывают разных размеров и подходят либо для обычных, либо для горных путешествий.

Рассмотрим следующее простое требование, которое чуть позже будет названо *пользовательским сценарием*: клиент, чтобы выбрать путешествие, хочет посмотреть список маршрутов соответствующего уровня сложности на конкретную дату и с доступными арендными велосипедами.

Формирование намерения

Начинать создание кода для совершенно нового приложения немного боязно. Когда пополняется уже существующая база исходного кода, расширяется существующий проект. А в нашем случае нужно с нуля принять решение, которое навсегда определит структуру приложения. Сейчас вы закладываете основу проекта, который позже будет расширен.

Вы знаете, что не должны сразу же приступать к написанию кода. Вы можете руководствоваться убеждением, что следует приступать к написанию тестов, но от этого убеждения ничуть не легче. У многих неопытных проектировщиков возникают серьезные трудности с решением, каким должен быть самый первый тест. Написание такого теста требует наличия замысла, касающегося предмета тестирования, то есть того, чего еще может и не быть в вашем распоряжении.

Первоклассным тестировщикам легко начать писать тесты благодаря их богатому опыту. На данной стадии у них уже выстроена мысленная схема возможностей этого приложения по части объектов и их взаимодействия. Они не привязаны к какому-либо конкретному замыслу и планируют использование тестов для исследования альтернатив, но у них настолько обширные познания в области проектирования, что уже сформировалось намерение относительно приложения. Именно это намерение и позволяет им определить характер первого теста.

Осознанно или подсознательно, но вы уже сформировали ряд собственных намерений. Вполне возможно, что описание характера деятельности компании FastFeet помогло определиться относительно потенциальных классов данного приложения. Возможно, вы уже ждете такие классы, как `Customer` (клиент), `Trip` (путешествие), `Route` (маршрут), `Bike` (велосипед) и `Mechanic` (механик).

Идея создания этих классов пришла в голову, потому что они представляют собой существительные в приложении, имеющем как *данные*, так и *поведение*. Назовите их *объектами предметной области*. Их очевидность обусловлена постоянным присутствием, они обозначают крупные, вполне ощутимые в реальном мире вещи, которые в конечном итоге превращаются в представления в вашей базе данных.

Объекты предметной области легко поддаются обнаружению, но они не являются центром проектирования вашего приложения. Это ловушка для опрометчивых людей. Если зацикливаться на объектах предметной области, возникнет желание задать им определенное поведение. Опытные проектировщики *замечают* объекты предметной области, не концентрируясь на них, их внимание сосредотачивается не на них, а на сообщениях, которые проходят между ними. Эти сообщения являются ориентирами, ведущими к открытию других объектов, которые также необходимы, но имеют гораздо меньшую степень очевидности.

Перед тем как приступить к написанию кода, следует сформировать намерения в отношении объектов *и* сообщений, необходимых для удовлетворения данного пользовательского сценария. Было бы неплохо иметь простой и недорогой способ связи для исследования конструкции, не требующий от вас написания кода. К счастью, нашлись умные люди, придумавшие эффективный механизм, именно для этого и предназначенный.

Диаграммы последовательности

Для экспериментов с объектами и сообщениями существует великолепный дешевый способ: *диаграммы последовательности*.

Диаграммы последовательности определяются в унифицированном языке моделирования — Unified Modeling Language (UML), и это только одна из множества диаграмм, поддерживаемых UML. Образцы некоторых диаграмм показаны на рис. 4.2.

Если вам приходилось работать с UML, то вы уже знаете цену диаграммам последовательности. А если UML вам незнаком и графика вас пугает, ничего страшного. Эта книга не превратится в руководство по UML. Упрощенная, гибкая конструкция не требует создания и сопровождения целой кучи вспомогательных

средств. Но создатели UML очень хорошо продумали способы связи с объектно-ориентированным проектированием, и вы можете воспользоваться плодами их труда. Существуют UML-диаграммы, предоставляющие превосходные промежуточные способы изучения и изложения проектировочных возможностей. Воспользуйтесь ими, для этого вам не потребуется заново изобретать колесо.

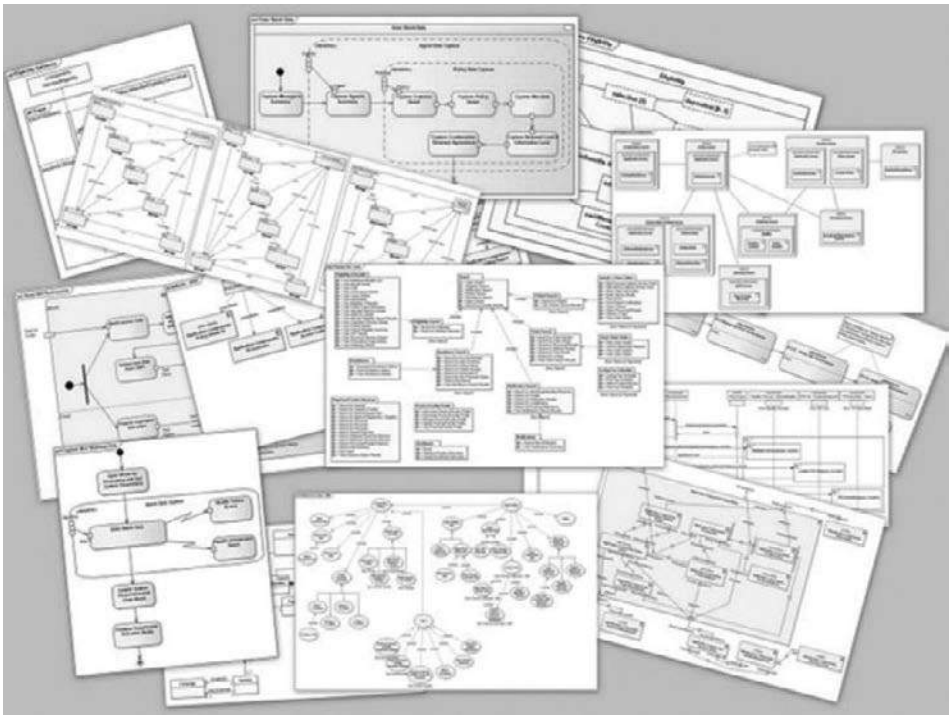

Рис. 4.2. Образцы UML-диаграмм

Работать с диаграммами последовательности очень удобно. Они помогают проводить эксперименты с различными расстановками объектов и схемами передачи сообщений. Они вносят ясность в ваши размышления и предоставляют вспомогательное средство для сотрудничества и связи с другими разработчиками. Их нужно воспринимать как легкий способ формирования намерений об организации взаимодействия. Рисуйте их на классной доске, вносите по мере надобности изменения и стирайте после того, как они сыграют отведенную им роль.

На рис. 4.3 представлена простая диаграмма последовательности. На этой диаграмме отображена попытка реализации рассмотренного выше пользовательского сценария. На ней показаны клиент по имени `Мое`, обозначенный как

`Customer`, и класс `Trip`, где `Moe` отправляет в адрес `Trip` сообщение о подходящих ему путешествиях `suitable_trips` и получает ответ.

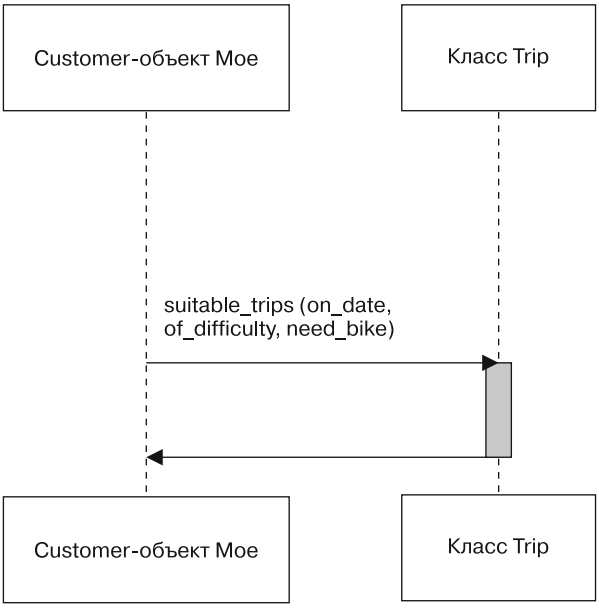

Рис. 4.3. Простая диаграмма последовательности

На рис. 4.3 показаны две основные части диаграммы последовательности. Как видите, на них отображены *объекты* и *сообщения*, которыми они обмениваются. В следующем абзаце описывается часть этой диаграммы, но следует учесть, что UML-полиция за отступления от официального стиля вас не арестует. Делайте то, что считаете полезным.

На приведенной в качестве примера диаграмме каждый объект представлен двумя одинаково названными блоками, расположенными один под другим и соединенными одной вертикальной линией. В них содержатся два объекта `Customer Moe` и класс `Trip`. Сообщения отмечены горизонтальными линиями. Когда отправляется сообщение, линия снабжается надписью с именем сообщения. Линии сообщений начинаются или заканчиваются стрелкой. Эта стрелка направлена в сторону получателя. Когда объект занят обработкой полученного сообщения, он активен, и его вертикальная линия расширяется до вертикального прямоугольника.

В диаграмме также содержится единственное сообщение `suitable_trips`, отправленное `Moe` в адрес класса `Trip`. Исходя из этого данная диаграмма

последовательности должна читаться следующим образом: некий клиент в лице `Customer`-объекта с условным именем `Moe` отправляет сообщение об условиях подходящих ему путешествий `suitable_trips` классу `Trip` (путешествия), который активизируется для обработки сообщения и возвращает ответ.

Диаграмма последовательности является весьма близким и буквальным переводом пользовательского сценария. Имена существительные в пользовательском сценарии становятся объектами в диаграмме последовательности, а действие в пользовательском сценарии превращается в сообщение. Сообщению требуются три параметра: `on_date` (в назначенную дату), `of_difficulty` (с указанной степенью сложности) и `need_bike` (нужен велосипед).

Хотя для иллюстрации частей диаграммы последовательности этот пример можно считать вполне подходящим, предполагаемая им конструкция должна заставить вас подумать. В диаграмме последовательности `Moe` ожидает, что класс `Trip` подыщет для него подходящее путешествие. Казалось бы, понятно, что `Trip` должен нести обязанность по подбору путешествий на указанную дату и с соответствующим уровнем сложности, но `Moe` может также понадобиться велосипед и он, конечно же, ожидает, что `Trip` также справится и с этой задачей.

Рисование этой диаграммы последовательности показывает сообщение, передаваемое между `Customer`-объектом `Moe` и классом `Trip`, и предлагает вам задать вопрос «Должно ли входить в обязанности `Trip` выяснение наличия соответствующего велосипеда для каждого подходящего путешествия?» или в более общей форме: «Должен ли этот получатель нести обязанность давать ответ на это сообщение?»

В этом и заключается ценность диаграмм последовательности. Они в явной форме уточняют сообщения, передаваемые между объектами, а поскольку объекты должны связываться друг с другом, используя открытые интерфейсы, диаграммы последовательности являются средствами проведения экспериментов и в конечном итоге определения таких интерфейсов.

Обратите также внимание, что теперь, когда вы нарисовали диаграмму последовательности, разговор вокруг конструкции приобрел обратное направление. Прежде в конструкции акцент делался на классы и на то, кого и что они знают. И тут внезапно разговор изменил направленность, и теперь он ведется вокруг сообщений. Вместо решений насчет классов с последующим выявлением их обязанностей теперь решения принимаются в отношении сообщений с последующим определением, куда их отправлять.

Этот переход от конструкции на основе классов к конструкции на основе сообщений — поворотная точка в вашей карьере проектировщика. Решение задачи на основе сообщений позволит создавать более гибкие приложения по

сравнению с вариантом на основе классов. Смена основополагающего вопроса проектирования с метода «Я знаю, что мне нужен этот класс, но что он должен делать?» на метод «Мне нужно отправить это сообщение, но кто должен на него ответить?» является первым шагом в этом направлении.

Сообщения отправляются не потому, что есть объекты, а объекты имеются, потому что отправляются сообщения.

С точки зрения отправки сообщения для `Customer` было бы вполне разумно отправить сообщение `suitable_trips`. Проблема не в том, что `Customer` должен его отправлять, а в том, что `Trip` не должен его получать.

Теперь, когда вы настроены на сообщение `suitable_trips`, место, куда его отправлять, отсутствует, поэтому нужно подобрать альтернативные варианты. С применением диаграмм последовательностей исследование возможностей существенно упрощается.

Если класс `Trip` не должен выяснять вопрос наличия велосипедов для путешествия, возможно, этим должен заняться класс `Bicycle` (велосипед). Классу `Trip` может быть вменено в обязанности отвечать на сообщение о подходящий путешествиях (`suitable_trips`), а классу `Bicycle` — о подходящих велосипедах (`suitable_bicycle`). Клиент по имени `Moe` может получить нужный ему ответ, если будет общаться с обоими классами. Соответствующая диаграмма последовательности принимает вид, показанный на рис. 4.4.

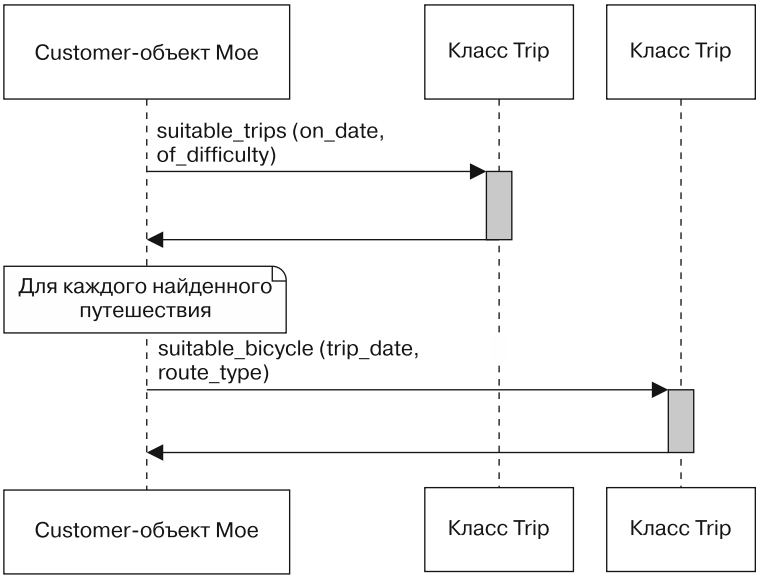

Рис. 4.4. Moe общается с Trip и Bicycle

Рассмотрим для каждой из этих диаграмм, о чем должен знать объект клиента `Moe`.

На рис. 4.3 он знает, что:

- ему нужен список путешествий;
- существует объект, реализующий сообщение `suitable_trips`.

На рис. 4.4 он знает, что:

- ему нужен список путешествий;
- существует объект, реализующий сообщение `suitable_trips`;
- часть подбора подходящего сообщения означает подбор подходящего велосипеда;
- существует еще один объект, реализующий сообщение `suitable_bicycle`.

Рисунок 4.4 более совершенен в одних областях, но не в состоянии справиться с другими областями. В этой конструкции из `Trip` удаляются несвойственные ему обязанности, но, к сожалению, они просто перекладываются на `Customer`.

Проблема, показанная на рис. 4.4, заключается в том, что клиент `Moe` не только знает, что ему нужно, он также знает, как другие объекты должны сотрудничать для предоставления необходимой ему информации. Класс `Customer` становится владельцем соответствующих правил приложения, с помощью которых оценивается пригодность путешествия.

Когда `Moe` знает, как принять решение о пригодности путешествия, он уже не заказывает поведение через меню, а идет на кухню и готовит еду. Класс `Customer` присваивает себе чьи-то обязанности и привязывает себя к реализации, которая может измениться.

Нужно не говорить «как», а спрашивать «что»

Разница между сообщением с вопросом о том, что нужно отправителю, и сообщением, говорящим получателю, как себя вести, может показаться незначительной, но она имеет весьма существенные последствия. Понимание этой разницы является ключевой частью создания повторно используемых классов с хорошо продуманными внешними интерфейсами.

Чтобы проиллюстрировать важность сопоставления вопросов *«что»* и *«как»*, следует воспользоваться более подробным примером. Отложим ненадолго вопросы проектирования по линии *«клиент — путешествие»*, к ним мы вскоре

вернемся. Переключим внимание на новый пример, в котором будут фигурировать путешествия, велосипеды и механики.

На рис. 4.5 путешествие готово к старту, и нужно убедиться, что все велосипеды в хорошем состоянии. Пользовательский сценарий для этой обязанности таков: чтобы начать путешествие, надо убедиться, что велосипеды, которые будут использоваться, механически исправны. Классу `Trip` *нужно* в точности знать, как подготовить велосипед к путешествию, и он должен попросить класс `Mechanic` выполнить каждое из этих действий:

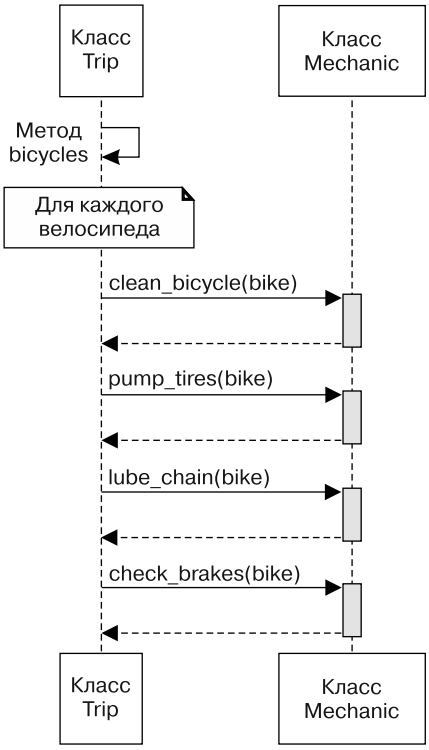

Рис. 4.5. Класс Trip говорит классу Mechanic, как подготовить каждый велосипед

На рис. 4.5:

- открытый интерфейс для `Trip` включает метод `bicycles`;
- открытый интерфейс для `Mechanic` включает методы `clean_bicycle` (помыть велосипед), `pump_tires` (накачать шины), `lube_chain` (смазать цепь) и `check_brakes` (проверить тормоза);

- ожидается, что класс `Trip` будет полагаться на объект, способный отвечать на сообщения `clean_bicycle`, `pump_tires`, `lube_chain` и `check_brakes`.

В данной конструкции класс `Trip` осведомлен о многих подробностях того, что делает класс `Mechanic`. Поскольку `Trip` содержит эти знания и использует их для направления классу `Mechanic`, то `Trip` должен измениться, если `Mechanic` добавляет новые процедуры по подготовке велосипеда. Например, если в классе `Mechanic` в качестве подготовки к работе метода `Trip` реализуется метод для проверки велоаптечки, `Trip` должен измениться, чтобы вызвать этот новый метод.

На рис. 4.6 показан альтернативный вариант, где `Trip` просит `Mechanic` подготовить каждый `Bicycle`, оставляя подробности реализации в ведении `Mechanic`.

На рис. 4.6:

- в открытый интерфейс для `Trip` включается метод `bicycles`;
- в открытый интерфейс для `Mechanic` включается метод `prepare_bicycle`;
- ожидается, что класс `Trip` будет полагаться на объект, способный отвечать на сообщения `prepare_bicycle`.

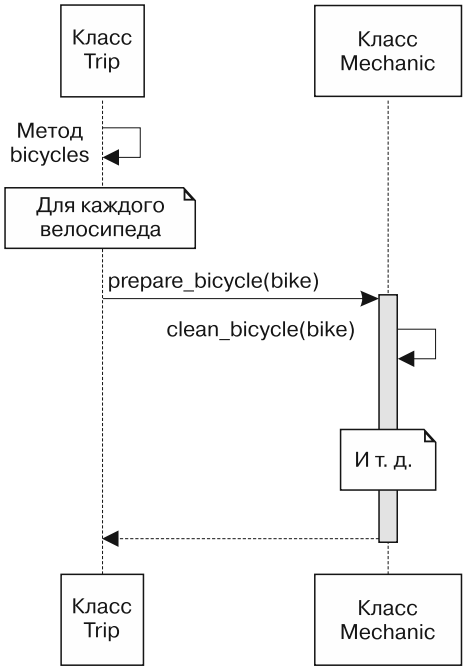

Рис. 4.6. Trip просит Mechanic подготовить каждый Bicycle

Теперь класс `Trip` переложил существенный перечень обязанностей на класс `Mechanic`. `Trip` знает, что ему нужно, чтобы был подготовлен каждый велосипед, и он доверяет выполнение этой задачи классу `Mechanic`. Поскольку ответственность за знание *«как»* была возложена на `Mechanic`, класс `Trip` всегда будет вести себя правильно, какие бы усовершенствования класса `Mechanic` ни случались с ним в будущем.

Когда разговор между `Trip` и `Mechanic` перешел с *«как»* на *«что»*, один из сопутствующих эффектов выразился в том, что размер внешнего интерфейса класса `Mechanic` резко уменьшился. На рис. 4.5 `Mechanic` открывает доступ ко многим методам, а на рис. 4.6 его открытый интерфейс состоит из одного метода по имени `prepare_bicycle`. Поскольку `Mechanic` обещает, что его открытый интерфейс будет стабилен и неизменен, наличие небольшого открытого интерфейса означает, что другие теперь зависят от малого количества методов. Это снижает вероятность того, что однажды открытый интерфейс класса `Mechanic` изменится, нарушив его обещания и заставив внести изменения во многие другие классы.

Это изменение схемы сообщений является существенным улучшением с точки зрения возможности сопровождения кода, но класс `Trip` по-прежнему знает многое о классе `Mechanic`. Код станет гибче и легче в сопровождении, если класс `Trip` сможет выполнить свои задачи при еще меньшем объеме знаний.

Поиск контекста независимости

То, что `Trip` знает о других объектах, составляет его *контекст*. Это нужно представлять таким образом: у `Trip` *есть* единственная обязанность, но этот класс *ожидает* контекст. На рис. 4.6 класс `Trip` возлагает свои надежды на объект типа `Mechanic`, способный ответить на сообщение `prepare_bicycle`.

Контекст является верхней одеждой, которую `Trip` носит повсюду; любое использование `Trip`, будь то тестирование или что-либо иное, требует, чтобы был образован его контекст. Подготовка путешествия, представляемого классом `Trip`, *всегда* требует подготовки велосипедов и происходит таким образом, что `Trip` *всегда* отправляет классу `Mechanic` сообщение `prepare_bicycle`. Вы не можете использовать `Trip` где-либо еще, если не предоставите объект, похожий на объект класса `Mechanic`, который ответит на это сообщение.

Ожидаемый объектом контекст напрямую влияет на степень трудности его повторного использования. Объекты, имеющие простой контекст, проще применять и легче тестировать, и у них весьма скромные ожидания, связанные с окружением. Объекты со сложным контекстом трудно использовать и нелегко тестировать, и для получения от них какой-нибудь пользы требуются сложные настройки. Лучше всего было бы, чтобы объект вообще не зависел от своего контекста. Объект, способный сотрудничать с другими объектами, не зная, кто они такие или что они делают, может повторно использоваться новыми и порой самыми неожиданными способами.

Вам уже известна технология сотрудничества с другими без знания о том, кто они такие, — это внедрение зависимости. Теперь у `Trip` будет новая задача — побудить класс `Mechanic` к правильному поведению, не зная о том, чем он занимается. Классу `Trip` нужно сотрудничать с классом `Mechanic`, сохраняя независимость от контекста.

На первый взгляд это кажется невозможным. Путешествие предполагает наличие велосипедов, которые должны быть подготовлены, а их подготовкой занимаются механики. Поэтому просьба `Trip` в адрес `Mechanic` подготовить `Bicycle` неизбежна.

Но это не так. Решение задачи заключается в различии между *«что»* и *«как»*, нахождение решения требует концентрации на том, что хочет `Trip`.

То, *что* хочет `Trip`, заключается в готовности к путешествию. Знание о том, что нужно подготовить, входит в обязанности `Trip`. Но факт наличия *велосипедов* в готовом состоянии может быть в компетенции класса `Mechanic`. Необходимость подготовки велосипедов больше относится к тому, как готовится `Trip`, а не к тому, что `Trip` нужно.

На рис. 4.7 показан третий вариант диаграммы последовательности для подготовки `Trip`. Здесь `Trip` просто сообщает `Mechanic` о том, что ему нужно, то есть что ему нужно быть в готовности, и передает в качестве аргумента самого себя.

На диаграмме последовательности `Trip` ничего не знает о `Mechanic`, но попрежнему управляет сотрудничеством с ним для получения готовых к эксплуатации велосипедов. `Trip` сообщает `Mechanic` о том, что ему нужно, передавая в качестве аргумента самого себя, и `Mechanic` тут же совершает обратный вызов в адрес `Trip`, чтобы получить список тех велосипедов, которые нужно подготовить.

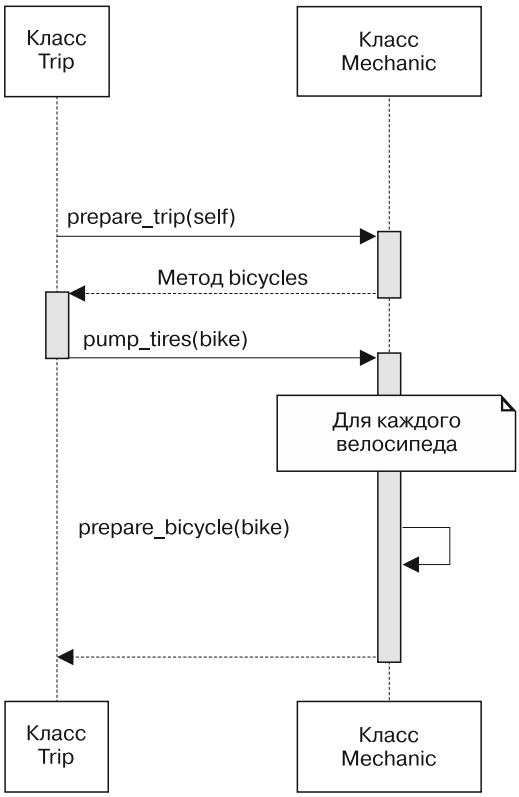

Рис. 4.7. Trip просит Mechanic подготовить Trip

На рис. 4.7:

- открытый интерфейс для `Trip` включает `bicycles`;
- открытый интерфейс для `Mechanic` включает `prepare_trip` и, возможно, `prepare_bicycle`;
- ожидается, что класс `Trip` будет полагаться на объект, который может ответить на сообщение `prepare_trip`;
- класс `Mechanic` ожидает передачи с сообщением `prepare_trip` аргумента, чтобы ответить на `bicycles`.

Все знания о том, как механики подготавливают путешествие, теперь изолированы внутри `Mechanic`, и контекст `Trip` сократился. Теперь оба объекта легче поддаются изменениям, тестированию и повторному использованию.

Доверие, оказываемое другим объектам

Конструкции, показанные на рис. 4.5–4.7, отображают перемещение в направлении все более *объектно*-ориентированного кода и таким образом отражают стадии разработки, которая ведется новичком в деле проектирования.

Конструкция на рис. 4.5 носит сугубо процедурный характер. `Trip` сообщает `Mechanic`, как готовить `Bicycle`, почти так же, как если бы `Trip` был основной программой, а `Mechanic` был пакетом вызываемых функций. В данной конструкции `Trip` является единственным объектом, точно знающим, как подготовить велосипед; получение готового к эксплуатации велосипеда требует использования `Trip` или создания дубликата кода. Контекст класса `Trip` получается большим; таким же большим получается открытый интерфейс класса `Mechanic`. Эти два класса вовсе не являются отдельными островами с наведенными между ними мостами, вместо этого они являются единой конструкцией.

Многие новички в области объектно-ориентированного программирования, создавая процедурный код, действуют именно так. Данный стиль зеркально отображает накопленный ими опыт работы с прежними процедурными языками.

К сожалению, программирование в процедурном стиле лишает ориентацию на объекты всякого смысла. В результате вновь возникают те самые проблемы сопровождения кода, для избавления от которых и создавалось объектно-ориентированное программирование.

На рис. 4.6 представлена более выраженная ориентация на объекты. Здесь показан класс `Trip`, который *просит* `Mechanic` подготовить `Bicycle`. Контекст `Trip` сократился, а открытый интерфейс `Mechanic` уменьшился. В дополнение к этому открытый интерфейс `Mechanic` превратился в то, чем теперь может воспользоваться любой объект. Чтобы подготовить велосипед, `Trip` теперь вам не нужен. Эти объекты общаются более конкретно, они меньше связаны друг с другом, их повторное использование упростилось.

В описанном стиле программирования обязанности возложены на правильные объекты, что является существенным усовершенствованием, но при этом все еще сохраняется требование наличия у `Trip` контекста, большего по объему, чем это необходимо на самом деле. `Trip` по-прежнему в курсе, что он полагается

на объект, способный ответить на сообщение `prepare_bicycle`, и такой объект всегда должен быть в его распоряжении.

На рис. 4.7 показана еще более объектно-ориентированная конструкция. `Trip` не знает или не беспокоится о том, что у него есть `Mechanic`, и он абсолютно не в курсе, что этот `Mechanic` будет делать. `Trip` просто полагается на объект, которому он будет отправлять сообщение `prepare_trip`; он верит в то, что получатель этого сообщения будет вести себя соответствующим образом.

В плане расширения данного замысла `Trip` может пометить несколько таких объектов в массив и послать каждому из них сообщение `prepare_trip`, доверяя каждому, кто занимается подготовительными работами, проведение соответствующих, возлагаемых на него работ. В зависимости от того, как используется `Trip`, у него может быть множество тех, кто занимается подготовительными работами, или же их может быть совсем немного. Эта модель позволяет добавлять к `Trip` новых сотрудников, ничего не меняя в самом коде этого класса, то есть `Trip` можно расширять без его *модификации*.

Если бы объекты были людьми и могли описать их взаимоотношения, то на рис. 4.5 `Trip` сказал бы `Mechanic`: «Я знаю, что мне нужно, и я знаю, как это сделать», на рис. 4.6 было бы сказано следующее: «Я знаю, что мне нужно, и я знаю, чем ты занимаешься», а на рис. 4.7: «Я знаю, что мне нужно, и я *доверяю тебе выполнение твоей части работы*».

Такое слепое доверие является краеугольным камнем объектно-ориентированного проектирования. Оно позволяет объектам сотрудничать, не привязывая их самих к контексту; это необходимо обеспечить в любом приложении, для которого рассматриваются возможности его разрастания и изменения.

Сообщения для обнаружения потребности в новых объектах

Вооружившись сведениями о различиях между *«что»* и *«как»* и о важности контекста и доверия, мы можем вернуться к исходной задаче проектирования, показанной на рис. 4.3 и 4.4.

Вспомним, что у этой задачи был следующий пользовательский сценарий: клиент, чтобы выбрать путешествие, захотел увидеть список доступных маршрутов соответствующего уровня трудности на конкретную дату и там, где предоставляется аренда велосипедов.

На рис. 4.3 был изображен буквальный перевод этого пользовательского сценария, и у `Trip` было слишком много обязанностей. На рис. 4.4 была предпринята попытка переместить обязанность по поиску доступных велосипедов с `Trip` на `Bicycle`, но при этом на `Customer` были возложены обязанности по содержанию слишком большого объема сведений о том, что именно делает путешествие «подходящим».

Ни одна из этих конструкций не может широко использоваться повторно или оставаться невосприимчивой к изменениям (эти проблемы были выявлены в диаграммах последовательности). У обеих конструкций нарушен принцип единственной обязанности. На рис. 4.3 у `Trip` имеется слишком много сведений. На рис. 4.4 слишком многими знаниями обременен `Customer`, который сообщает другим объектам, как им нужно себя вести, и требует слишком большого объема контекста.

Вполне логично, что `Customer` нужно отправлять сообщение `suitable_trips`. Это сообщение повторяется в обеих диаграммах, поскольку оно представляется правильным по определению самой сути этого класса. Это именно то, *что* `Customer` хочет получить. Проблема не с отправителем, а с получателем. Вы еще не определили объект, в чьи обязанности будет входить реализация этого метода.

Это приложение нуждается в объекте, воплощающем в себе правила на пересечении `Customer`, `Trip` и `Bicycle`. Метод `suitable_trips` будет частью именно *его* открытого интерфейса.

Вам следует понять, что нужен еще не определенный объект, на котором заканчивается множество маршрутов. Преимущество, получаемое от обнаружения необходимости в пока еще отсутствующем объекте при составлении диаграммы последовательности, заключается в том, что ошибки обходятся весьма недорого, а препятствий для того, чтобы изменить свое мнение, нет практически никаких. Диаграммы последовательности составляются в экспериментальных целях и со временем утрачивают свое значение, их особенностью является то, что вы к ним никак не привязаны. Они не демонстрируют вашу конструкцию в ее конечном виде, но зато в них отображается замысел, являющийся стартовой позицией для проектирования.

Независимо от того, как именно вы дошли до этой позиции, теперь уже совершенно ясно, что вам нужен новый объект, необходимость в котором была обнаружена потому, что вам нужно отправить ему сообщение.

Возможно, в приложении должен быть класс для поиска путешествий — `TripFinder`. На рис. 4.8 показана диаграмма последовательности, где классу `TripFinder` вменено в обязанность подыскивать подходящие путешествия.

`TripFinder` содержит все сведения о том, что именно делает путешествие подходящим. Его работа заключается в том, чтобы делать все необходимое для ответа на соответствующее сообщение. Он предоставляет соответствующий открытый интерфейс и в то же время скрывает запутанные и склонные к изменениям внутренние детали.

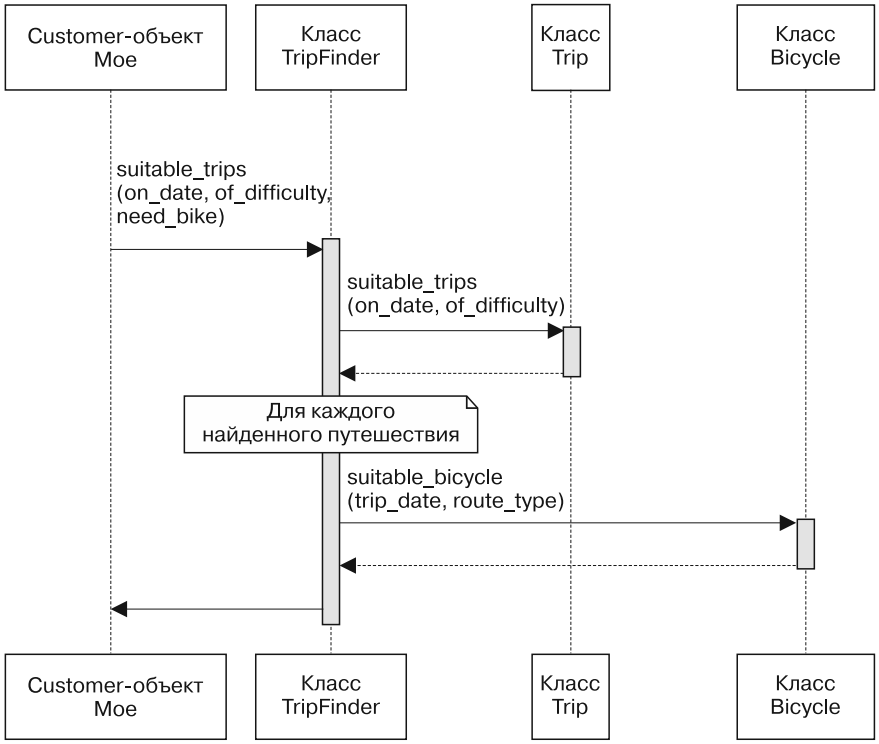

Рис. 4.8. Мое просит TripFinder подобрать подходящее путешествие

Перемещение этого метода в `TripFinder` делает поведение доступным любому другому объекту. Возможно, в будущем классом `TripFinder` воспользуются другие туристические компании, чтобы подобрать подходящие путешествия через веб-сервис. Теперь, когда это поведение было изъято из `Customer`, он может изолированно использоваться любым другим объектом.

Создание приложения, основанного на сообщениях

В этом разделе диаграммы последовательности использовались для исследования конструкции, определения открытых интерфейсов и обнаружения необходимости в новых объектах.

Особую пользу диаграммы последовательности приносят на этапе перехода к проектированию — позволяют разобраться во всей первоначальной путанице. Вернитесь на несколько страниц назад и попытайтесь выстроить рассуждения без этих диаграмм.

Но, как бы полезны они ни были, это всего лишь инструмент. Диаграммы помогают сосредоточиться на сообщениях и позволяют сделать предварительную прикидку, что нужно подтвердить тестами в первую очередь. Переключение внимания с объектов на сообщения позволяет сконцентрироваться на проектировании приложения, выстраиваемого вокруг открытых интерфейсов.

Написание кода с отличным интерфейсом

Ясность созданных вами интерфейсов свидетельствует о вашем мастерстве проектировщика. Поскольку совершенству проектировочного мастерства нет предела и даже сегодняшняя совершенная конструкция в свете завтрашних требований может оказаться уродливой, создание идеальных интерфейсов дается с очень большим трудом. Однако это ни в коем случае не должно вас останавливать.

Интерфейсы развиваются, но для этого им сначала необходимо появиться на свет. Важнее совершенства интерфейса оказывается лишь его четкое определение.

Тщательно продумывайте свои интерфейсы. Создавайте их с вполне конкретными намерениями. Ваше приложение и определение его будущего зависят больше от интерфейсов, чем от всех тестов и любого другого вашего кода.

В следующем разделе содержатся правила, которых нужно придерживаться при создании интерфейсов.

Создавайте четко выраженные интерфейсы

Ваша цель заключается в написании кода, который работает в сегодняшних условиях, может быть легко использован повторно и приспособлен для неожиданных вариантов применения в будущем. Ваши методы будут вызываться другими людьми, и вы обязаны сообщить, каким из них можно доверять.

При каждом создании класса объявляйте его интерфейсы. Методы в *открытом* интерфейсе должны:

- быть явно идентифицированными;
- больше иметь отношение к тому, *«что»*, чем к тому, *«как именно»*;
- обладать именами, которые (насколько это возможно предвидеть) не претерпят изменений;
- получать в качестве дополнительного параметра хеш.

Применяйте такие же намерения и для создания закрытого интерфейса, делайте его как можно более понятным. Эти намерения могут поддерживаться тестами, поскольку они служат в качестве документации. Либо вообще не предлагайте тесты для закрытых методов, либо (если без этого не обойтись) отделяйте эти тесты от тестов открытых методов. Не позволяйте своим тестам вводить других в заблуждение относительно непреднамеренной зависимости от склонного к изменениям закрытого интерфейса.

В Ruby предоставляются три соответствующих ключевых слова: `public` (открытый), `protected` (защищенный) и `private` (закрытый). Использование этих ключевых слов преследует две цели. Во-первых, они показывают, какие из методов стабильны, а какие — нет. Во-вторых, они управляют видимостью метода со стороны других частей приложения. Эти две цели отличаются друг от друга: передача информации о том, что метод является стабильным или нестабильным, — это одно, а попытка управлять тем, как им пользуются другие, — совершенно иное.

КЛЮЧЕВЫЕ СЛОВА PUBLIC, PROTECTED И PRIVATE

Ключевое слово private служит признаком наименее стабильной разновидности метода и предоставляет весьма ограниченную видимость. Объявляемые с его помощью закрытые методы должны вызываться без указания явного получателя, или, иначе говоря, они вообще не могут вызываться с указанием конкретного получателя.

Если в классе Trip содержится закрытый метод fun_factor, вы не можете отправить сообщение self.fun_factor за пределы Trip или отправить сообщение a_trip.fun_factor из другого объекта. Однако вы можете отправить fun_factor с исходной установкой на self (неявный получатель) из экземпляров Trip и из его подклассов.

Ключевое слово protected также служит признаком нестабильности метода, но ограничения видимости задаются несколько иные. Объявляемые с его помощью защищенные методы допускают явно указанных получателей при условии, что получателем выступает self или экземпляр этого же класса либо подкласса от self.

Таким образом, если принадлежащий классу Trip метод fun_factor является защищенным, вы всегда сможете отправить сообщение self.fun_factor. Кроме того, вы можете отправить сообщение a_trip.fun_factor, но только из класса, где self обозначает ту же разновидность объектов (класс или подкласс), что и a_trip.

Ключевое слово public служит признаком стабильности объекта. Объявляемые с его помощью открытые методы видны повсюду.

Чтобы еще больше все усложнить, Ruby предоставляет не только эти ключевые слова, но и различные механизмы обхода ограничений видимости, навязываемых применением ключевых слов `private` и `protected`. Пользователи класса могут переопределить любой метод в открытый независимо от его первоначального объявления. Ключевые слова `private` и `protected` больше похожи на гибкие барьеры, чем на твердые ограничения. Преодолеть их может кто угодно, стоит лишь приложить усилия.

Поэтому любые утверждения, что с помощью этих ключевых слов доступ к методу можно предотвратить, не более чем иллюзия. Ключевые слова не запрещают доступ, они его только слегка усложняют. Их использование сообщает о следующих двух обстоятельствах:

- вы полагаете, что владеете сегодня более полной информацией, чем она будет у программистов *в будущем*;
- вы полагаете, что этих будущих программистов нужно предостеречь от случайного использования метода, который на данный момент вы считаете нестабильным.

Эти представления могут быть верными, но будущее непредсказуемо. Предполагаемые наиболее стабильные методы могут периодически изменяться, а многие изначально нестабильные методы могут пройти проверку временем. Если иллюзия управляемости вас устраивает, можете свободно пользоваться

этими ключевыми словами. Но многие опытные Ruby-программисты применяют вместо них комментарии или специальное соглашение об присваиваемых методам именах (в Ruby on Rails, например, к закрытым методам добавляется символ подчеркивания «_»), чтобы обозначить открытую и закрытую части интерфейсов.

Эти стратегии вполне приемлемы, а иногда им даже следует отдавать предпочтение. Они предоставляют информацию о стабильности методов без введения ограничений видимости. Их использование передает будущим программистам право правильного выбора тех методов, от которых можно зависеть на основе дополняющейся информации, имеющейся на данный момент.

Независимо от того, каким станет ваш выбор (при условии, что вы нашли определенный способ передачи этой информации), будет считаться, что вы выполнили свои обязательства перед будущим.

Уважайте чужие открытые интерфейсы

Чтобы наилучшим образом организовать взаимодействие с другими классами, используйте для этого исключительно их открытые интерфейсы. Нужно предполагать, что у авторов этих классов были такие же намерения, как и у вас сейчас, и что они отчаянно пытаются сквозь время и пространство донести до всех, какие из методов следует считать надежными. Установленные ими различия открытости-закрытости предназначены для того, чтобы помочь *вам*, поэтому к ним лучше прислушаться.

Если ваша конструкция будет вынуждена пользоваться закрытым методом другого класса, то в первую очередь следует пересмотреть ее. Вполне возможно, что в результате целенаправленных усилий вы отыщете альтернативные варианты (нужно постараться найти хотя бы один).

Когда выстраивается зависимость от закрытого интерфейса, повышается риск вынужденных изменений. Когда закрытый интерфейс является частью внешней среды, периодически обновляющейся, эта зависимость становится похожа на бомбу с часовым механизмом, готовую взорваться в самый неподходящий момент. Человек, создавший зависимость, рано или поздно найдет себе более выгодную работу, внешняя среда обновится, закрытый метод будет

зависеть от изменений, и приложение даст сбой, сбивающий с толку специалистов, сопровождающих его код в данный момент. Так что лучше избегайте подобных зависимостей.

Будьте осмотрительны при наличии зависимости от закрытых интерфейсов

Несмотря на все приложенные усилия, может оказаться, что зависимости от закрытого интерфейса избежать не удается. Это весьма опасная зависимость, которая должна быть изолирована с использованием технологий, рассмотренных в главе 3. Даже если избежать использования закрытого метода не удается, вы можете воспрепятствовать появлению ссылок на него во многих местах вашего приложения.

Минимизация контекста

Открытый интерфейс следует создавать с прицелом на минимизацию контекста, требуемого ему от других. Нужно постоянно помнить о разнице понятий *«что»* и *«как»*; создавайте открытые методы, позволяющие отправителям получить желаемое без знаний о том, как именно ваш класс реализует свое поведение.

И наоборот, не попадайте в подчиненность к классу, имеющему слабовыраженный открытый интерфейс или вовсе его не имеющему. При возникновении подобной ситуации, как и в случае с классом `Mechanic` на рис. 4.5, сообщите ему, как себя вести, путем вызова всех его методов. Даже если автор исходного кода не определил открытый интерфейс, то еще не поздно создать его для себя.

В зависимости от того, насколько часто вы планируете использовать новый открытый интерфейс, он может быть представлен новым, определенным вами методом и помещен в класс `Mechanic`, новый класс-оболочку, созданный вами и используемый вместо `Mechanic`, или же представлен одним методом-оболочкой, помещенным в ваш собственный класс. Нужно делать то, что больше отвечает вашим потребностям, то есть создать определенную разновидность открытого интерфейса и использовать его в своих целях. Тем самым вы сократите контекст класса, облегчите возможность повторного использования и упростите проведение тестирования.

Закон Деметры

Начитавшись об обязанностях, зависимостях и интерфейсах, вы теперь подкованы знаниями для исследования закона Деметры.

Закон Деметры (Law of Demeter, LoD) представляет собой свод правил программирования, соблюдение которых призвано обеспечить слабую связь объектов. Слабая связь практически всегда идет во благо, но это всего лишь один из компонентов проектирования, и он должен быть сбалансирован с конкурирующими потребностями. Одни нарушения закона Деметры вреда не наносят, а другие не позволяют правильно идентифицировать и определить открытые интерфейсы.

Определение закона

Закон Деметры ограничивает набор объектов, которым метод может *отправлять* сообщения, и запрещает перенаправление сообщения третьему объекту через второй объект другого типа. Закон Деметры часто трактуют как «ведение разговора только с ближайшими соседями» или «использование в синтаксисе только одного символа точки». Представим, что метод `depart` (отправление) класса `Trip` (путешествие) содержит каждую следующую строку кода:

```
customer.bicycle.wheel.tire
customer.bicycle.wheel.rotate
hash.keys.sort.join(', ')
```

Каждая строка является цепочкой сообщения, содержащей несколько точек. Эти цепочки в просторечии называют *кошмарным сном крушения поезда*; каждое имя метода представляется вагоном, а каждая точка — сцепкой. Такие поезда являются свидетельством нарушения закона Деметры.

Последствия нарушений

Закон Деметры — это скорее правило, а не закон в привычном понимании слова.

В главе 2 утверждалось, что этот код должен быть *понятным*, *обоснованным*, *практичным* и *образцовым*. Некоторые из показанных выше цепочек сообщений дадут сбой, если погрешат против этой истины:

- если `wheel` изменяет `tire` или `rotate`, может потребоваться внесение изменений в `depart`. `Trip` не имеет никакого отношения к `wheel`, при этом изменения

в `wheel` могут предопределить внесение изменений в `Trip`. Совершенно неоправданно повысятся затраты на изменения; код не обладает *обоснованностью*;

- внесение изменения в `tire` или `rotate` может что-нибудь нарушить в `depart`. Поскольку `Trip` от них дистанцирован и, по-видимому, не связан, сбой будет совершенно неожиданным. Этот код *непонятен*;
- `Trip` не может повторно использоваться, пока у него имеется доступ к `customer` (клиенту) с `bicycle` (велосипедом), у которого есть `wheel` (колесо) и `tire` (шина). Ему требуется большой объем контекста; его *практичность* под вопросом;
- при повторении этой схемы сообщений кем-нибудь другим будет создано еще больше кода с теми же проблемами (как говорится, подобное рождает подобное). Такой код не может считаться *образцовым*.

Первые две цепочки сообщений почти что идентичны и различаются только тем, что одна извлекает отдаленный атрибут (`tire`), а другая вызывает поведение (`rotate`). Даже в среде опытных проектировщиков идут споры, насколько обоснованно закон Деметры применим к тем цепочкам сообщений, которые возвращают *атрибуты*. Возможно, *в вашем конкретном случае* самым простым способом извлечения отдаленных атрибутов как раз и будет переход через промежуточные объекты. Нужно находить баланс между вероятностью, стоимостью изменений и стоимостью устранений нарушения закона. Если, к примеру, вы выводите на печать отчет о наборе взаимосвязанных объектов, наиболее рациональной стратегией может быть явное указание промежуточных объектов и изменение отчета по мере надобности. Поскольку для стабильных атрибутов риск, возникающий при нарушении законов Деметры, невелик, эта стратегия может оказаться наиболее экономически эффективной.

Этот компромисс допустим до тех пор, пока вы не изменяете значение извлекаемого атрибута. Если метод `depart` отправляет сообщение `customer.bicycle.wheel.tire`, намереваясь внести изменение в результат, это уже не просто извлечение атрибута, а реализация поведения, принадлежащего классу `Wheel`. В данном случае `customer.bicycle.wheel.tire` становится похожим на `customer.bicycle.wheel.rotate`; эта цепочка проходит через множество объектов, чтобы добраться до отдаленного поведения. Такой стиль программирования обходится слишком дорого, и данное нарушение нужно устранить.

Третья цепочка сообщений `hash.keys.sort.join` вполне разумна, несмотря на обилие точек (это вообще может не иметь никакого отношения к нарушению закона Деметры). Вместо того чтобы оценивать эту фразу по количеству точек, оцените ее путем проверки типов промежуточных объектов:

- `hash.keys` возвращает тип `Enumerable` (перечисление);
- `hash.keys.sort` также возвращает тип `Enumerable` (перечисление);
- `hash.keys.sort.join` возвращает тип `String` (строку).

Исходя из этой логики, незначительное нарушение закона Деметры все же имеет место. Однако если вы в состоянии смириться с тем фактом, что `hash.keys.sort.join` возвращает перечисление строк (`Enumerable` из `Strings`), все промежуточные объекты имеют один и тот же тип и здесь нет нарушения закона Деметры. Если убрать точки из *этой* строки кода, расходы вместо снижения вырастут.

Как видите, подходить к закону Деметры следует намного деликатнее, чем кажется на первый взгляд. Зафиксированные в нем правила не являются самоцелью; как и любые принципы проектирования, они существуют *для обслуживания* ваших общих целей. Некоторые нарушения закона Деметры снижают гибкость вашего приложения и затрудняют его сопровождение, в то время как другие имеют вполне определенный смысл.

Как обойтись без нарушений

Одним из устоявшихся способов избавления от «крушения поезда» в коде программы является использование делегирования с целью отказа от точек. Чтобы в объектно-ориентированном понимании *делегировать* сообщение, его нужно передать в другой объект, зачастую применяя для этого метод-оболочку. Этот метод инкапсулирует или скрывает знания, которые в противном случае были бы воплощены в цепочке сообщений.

Осуществить делегирование можно несколькими способами. В Ruby есть библиотеки `delegate.rb` и `forwardable.rb`, а в среду Ruby on Rails включен метод `delegate`. Каждый из них предназначен для упрощения автоматического перехвата объектом сообщения, посланного *ему самому* вместо того, чтобы отправить его кому-нибудь еще.

Делегирование выглядит отличным решением проблем с законом Деметры, поскольку устраняет видимые свидетельства его нарушений.

Прислушиваясь к закону Деметры

Закон Деметры старается донести до вас полезные установки — и они не ограничиваются призывом «больше делегировать».

Цепочки сообщений вроде `customer.bicycle.wheel.rotate` возникают, когда ваша конструкция находится под чрезмерным влиянием уже известных вам объектов. Ваше знакомство с открытыми интерфейсами известных объектов может привести к объединению в одной строке длинных цепочек сообщений, чтобы добраться до отдаленного поведения.

Добираться через разрозненные объекты, чтобы вызвать отдаленное поведение, — все равно что говорить: «Вон там есть некий способ поведения, который мне нужен именно здесь, и *я знаю, как до него добраться*». Код знает не только то, *что* ему нужно (получить вращение, `rotate`), но и *как* пробраться через целую связку промежуточных объектов, чтобы добраться до желаемого поведения. Точно так же, как раньше класс `Trip` знал, как класс `Mechanic` должен подготовить велосипед, и поэтому был сильно сцеплен с `Mechanic`, здесь метод `depart` знает, как пройти через целый ряд объектов, чтобы заставить колесо вращаться, и поэтому он сильно сцеплен со всей структурой объектов.

Это сцепление — причина всевозможных проблем. Самой очевидной из них является повышение риска, что класс `Trip` будет вынужден претерпеть изменения из-за не имеющего к нему отношения изменения где-нибудь в цепочке сообщений. Но есть и еще одна более серьезная проблема.

Когда метод `depart` знает об этой цепочке объектов, он привязывает самого себя к весьма специфической реализации и не может быть повторно использован в любом другом контексте. Клиенты `Customers` должны всегда обеспечиваться велосипедами `Bicycles`, которые, в свою очередь, должны иметь вращающиеся колеса `Wheels`.

Посмотрим, на что станет похожа эта цепочка сообщений, если вы начнете с решения, *что именно* нужно методу отправления `depart` от клиента `customer`. С точки зрения, основанной на первоочередном рассмотрении сообщений, ответ очевиден:

```
customer.ride
```

Метод `ride` объекта `customer` скрывает подробности реализации от класса `Trip` и сокращает его контекст и количество его зависимостей, существенно повышая качество конструкции. Когда компания FastFeet изменит свой профиль и приступит к организации пеших туров, будет намного проще ввести в употреб-

ление метод `customer.depart` или `customer.go` вместо `customer.ride` (это распутает хитросплетения имеющейся в вашем приложении цепочки сообщений).

Проблемы при нарушении закона Деметры сообщают о наличии объектов с недостаточно развитыми открытыми интерфейсами. Если вы предпочитаете обзор перспектив на основе сообщений, то выявленные сообщения превратятся в открытые интерфейсы в новых объектах, необходимость в которых обнаружится благодаря этим сообщениям. Но если вы «повязаны» уже существующими объектами конкретной предметной области, придется собирать имеющиеся у них открытые интерфейсы в длинные цепочки сообщений, упуская возможность поиска и построения гибких открытых интерфейсов.

Выводы

Объектно-ориентированные приложения определяются с помощью сообщений, передаваемых между объектами. Передача сообщений происходит по открытым интерфейсам. Четко выраженные открытые интерфейсы состоят из стабильных методов, предоставляющих сведения об обязанностях своих исходных классов и дающих максимум преимуществ при минимуме затрат.

Сконцентрировавшись на сообщениях, вы сможете увидеть необходимость в новых объектах, которую иначе можно было бы упустить из виду. Когда сообщения отправляются с *доверием* к получателю и запрашивают у него то, что нужно отправителю (вместо того чтобы сообщать ему о том, как нужно выстроить его поведение), объекты естественным образом формируют открытые интерфейсы, обладающие гибкостью и позволяющие использовать эти объекты повторно новыми и порой самыми неожиданными способами.

Глава 5

Снижение затрат за счет неявной типизации

Цель объектно-ориентированного проектирования — снижение затрат на внесение изменений. Теперь, когда стало понятно, что центром проектирования приложения являются сообщения, и когда вы стали приверженцем построения четко определенных открытых интерфейсов, можно объединить эти два положения в эффективную технологию проектирования, которая поможет снизить ваши затраты.

Эта технология называется *неявной («утиной», или латентной) типизацией* (под ней понимаются открытые интерфейсы, не привязанные ни к какому конкретному классу). Эти проникающие сквозь классы интерфейсы придают вашим приложениям невероятную гибкость путем замены весьма затратных зависимостей от классов зависимостями от сообщений.

Объекты с неявной типизацией — это хамелеоны, определяемые в первую очередь по их поведению, а не по их классу (именно за это технология и получила свое название; если объект крякает как утка и ходит как утка, то его класс не имеет значения, это утка).

В этой главе рассматриваются способы распознавания и использования неявной типизации для придания приложениям большей гибкости и облегчения процесса их изменения.

Основные сведения о неявной типизации

Понятие «тип» используется в языках программирования для описания категории содержимого переменной. Процедурные языки предоставляют небольшое фиксированное количество типов, которые используются в основном для описания разновидностей *данных*. Даже в самых простых языках определяются типы для хранения строк, чисел и массивов.

Именно знание категории содержимого переменной, или ее типа, позволяет приложению выстраивать предположение о том, как это содержимое себя поведет.

Приложения вполне обоснованно предполагают, что числа могут использоваться в математических выражениях, строки могут объединяться, а массивы — индексироваться. В Ruby подобные ожидания, касающиеся поведения объекта, принимают форму представления о его открытом интерфейсе. Если один объект знает тип другого объекта, то он знает, на какие сообщения этот объект в состоянии отвечать.

Экземпляр класса `Mechanic`, по-видимому, содержит полный открытый интерфейс этого класса. Совершенно очевидно, что любой объект, полагающийся на экземпляр класса `Mechanic`, может рассматривать этот экземпляр, как будто он и *есть* `Mechanic`; объект по своей природе реализует открытый интерфейс класса `Mechanic`.

Но вы не ограничены ожиданием того, что объект реагирует только на *один* интерфейс. Объект Ruby подобен участнику бала-маскарада, меняющего маски, подстраиваясь под тему. Каждому, кто на него смотрит, он может показывать другое лицо; он может осуществлять реализацию множества различных интерфейсов. Тип объекта внутри вашего приложения определяется взглядом смотрящего. Пользователям объекта не нужно знать, к какому классу он относится, и они не должны беспокоиться об этом. Класс для объекта является всего лишь одним из способов получения открытого интерфейса, а открытый интерфейс, приобретаемый объектом от своего класса, может быть одним из нескольких, составляющих его содержимое. Приложения могут определять множество открытых интерфейсов, не связанных с одним конкретным классом; эти интерфейсы как бы пронизывают класс. Пользователи любого объекта могут совершенно спокойно ожидать, что он будет действовать как любой или как все реализованные в нем открытые интерфейсы. И неважно, чем именно является этот объект, важно то, что он *делает*.

Если каждый объект верит всем остальным объектам, что они в любой отдельно взятый момент станут тем, что он ожидает, и любой объект может быть чем угодно, то возможности конструкции становятся безграничными. Этими возможностями можно воспользоваться для создания гибких конструкций, являющих собой чудеса изобретательности.

Разумное использование гибкости требует от вас способности распознавать проникающие сквозь классы типы и выстраивать открытые интерфейсы с такими же четко выраженными намерениями и с тем же усердием, которое проявлялось при работе с типами внутри классов в главе 4. Типы, проникающие сквозь классы, то есть неявные типы, обладают открытыми интерфейсами, представляющими собой четко прописанный и хорошо документированный контракт.

Неявные типы лучше всего объяснить путем изучения последствий отказа от их использования. В этом разделе используется пример, претерпевающий множество реорганизаций и решающий весьма запутанную задачу проектирования путем поиска возможности применения и реализации неявного типа.

Упущение из виду возможностей применения неявной типизации

В следующем коде принадлежащий классу `Trip` метод `prepare` отправляет сообщение `prepare_bicycles` объекту, содержащему параметр `mechanic`. Обратите внимание, что ссылка на класс `Mechanic` отсутствует; хотя параметр называется `mechanic`, содержащийся в нем объект может быть любого класса.

```
class Trip
    attr_reader :bicycles, :customers, :vehicle

    # этот аргумент 'mechanic' может быть любого класса
    def prepare(mechanic)
         mechanic.prepare_bicycles(bicycles)
    end

# ...
end

# если случайно передать экземпляр класса *this*,
# это сработает
class Mechanic
```

```
15      def prepare_bicycles(bicycles)
16       bicycles.each {|bicycle| prepare_bicycle(bicycle)}
17      end
18
19      def prepare_bicycle(bicycle)
20          #...
21      end
22  end
```

Соответствующая диаграмма последовательности показана на рис. 5.1. Внешний объект получает все, что ему нужно, начиная с отправки `prepare` в адрес `Trip` и передавая при этом аргумент.

У метода `prepare` нет явно выраженной зависимости от класса `Mechanic`, но он зависит от получения объекта, способного отвечать на сообщения `prepare_bicycles`. Эта зависимость столь основательная, что ее легко проглядеть или не взять в расчет, но тем не менее она существует. Принадлежащий классу `Trip` метод `prepare` твердо верит, что его аргумент содержит сведения о том, кто занимается подготовкой велосипедов.

Усугубление проблемы

Возможно, вы заметили, что этот пример похож на диаграмму последовательности, показанную на рис 4.6 из главы 4. Следующая реорганизация кода улучшит конструкцию за счет отправки сведений о том, как готовится путешествие `Trip` в класс `Mechanic`. Но в показанном примере не предвидится вообще никаких улучшений.

Представим себе, что требования изменились. Теперь, кроме механика, к подготовке путешествия подключаются координатор (coordinator) и водитель (driver). В соответствии со сложившейся моделью кода вы создаете новые классы `TripCoordinator` и `Driver` и снабжаете их поведением в соответствии с обязанностями. Также вносятся изменения в метод `prepare` класса `Trip`, чтобы он вызывал корректное поведение на основе каждого из своих аргументов.

Изменения показаны в следующем коде. Новые классы `TripCoordinator` и `Driver` просты и безобидны, но метод `prepare` класса `Trip` теперь вызывает тревогу. Он ссылается на три различных класса по именам и знает конкретные методы, реализованные в каждом из них. Риски существенно возросли. Метод `prepare` класса `Trip` может принуждаться к изменениям из-за изменений в другом месте и способен неожиданно дать сбой в результате отдаленного, не связанного с ним изменения.

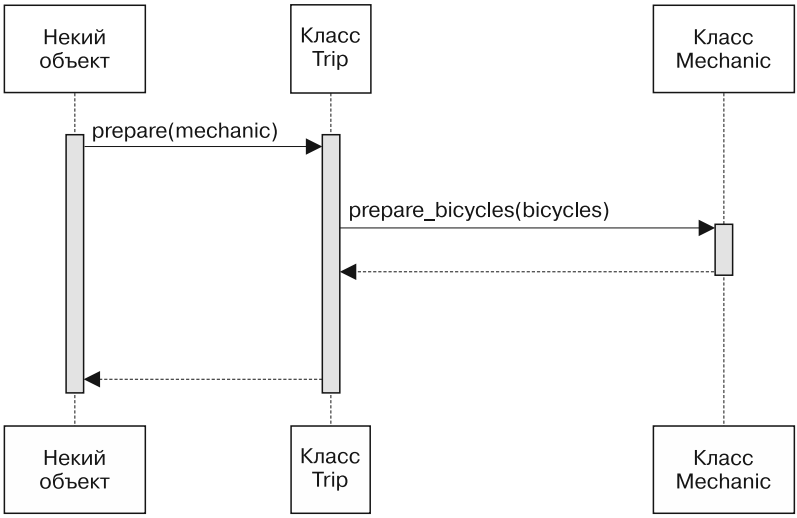

Рис. 5.1. Trip подготавливается, отправляя механику просьбу о подготовке велосипедов

```
# подготовка путешествия, Trip, становится сложнее
class Trip
    attr_reader :bicycles, :customers, :vehicle

    def prepare(preparers)
         preparers.each {|preparer|
            case preparer
            when Mechanic
                 preparer.prepare_bicycles(bicycles)
            when TripCoordinator
                 preparer.buy_food(customers)
            when Driver
                 preparer.gas_up(vehicle)
                 preparer.fill_water_tank(vehicle)
            end
         }
    end
end

# вводятся TripCoordinator и Driver
class TripCoordinator
    def buy_food(customers)
        # ...
```

```
    end
end

class Driver
    def gas_up(vehicle)
        #...
    end

    def fill_water_tank(vehicle)
        #...
    end
end
```

Этот код — первый шаг на пути, который заведет вас в тупик. Код подобного рода создается, когда у программистов перед глазами стоят одни лишь существующие классы и они не хотят замечать, что упустили из виду важные сообщения. Такой перегруженный зависимостями код — закономерный результат выбора принципа оценки ситуации на основе классов.

Основную суть проблемы составляют, казалось бы, совершенно безобидные вещи. Можно легко оказаться в ловушке, думая об исходном методе `prepare`, что он ожидает реальный экземпляр класса `Mechanic`. Вы со своим техническим мышлением, безусловно, согласитесь с тем, что аргумент метода `prepare` может вполне законно принадлежать любому классу, но это вас не спасет; в глубине души вы все равно будете думать, что аргумент принадлежит классу `Mechanic`.

Поскольку вам известно, что `Mechanic` понимает сообщение `prepare_bicycle`, и вы уверены в том, что оно передается классу `Mechanic`, то изначально все вроде бы идет неплохо. Этот взгляд на вещи кажется разумным, пока что-нибудь не изменится и в списке аргументов не станут появляться экземпляры других классов, не относящиеся к классу `Mechanic`. Как только это произойдет, методу `prepare` придется совершенно неожиданно для него работать с объектами, не понимающими сообщения `prepare_bicycle`.

Если ваши представления о проектировании ограничиваются классом и вдруг обнаружится, что вы имеете дело с объектами, не понимающими отправленного вами сообщения, у вас возникнет желание открыть охоту на сообщения, которые *понятны* этим новым объектам. Поскольку новые аргументы являются экземплярами классов `TripCoordinator` и `Driver`, вы вполне естественно начнете изучать открытые интерфейсы этих классов и обнаружите методы закупки продовольствия `buy_food`, заправки горючим `gas_up` и наполнения водой резервной емкости `fill_water_tank`. Метод подготовки `prepare` теперь нуждается именно в этом поведении.

Самый очевидный способ вызова данного поведения — отправка соответствующих ему сообщений, но теперь перед вами возникло препятствие. Все ваши аргументы принадлежат различным классам и реализуют различные методы, и вам требуется определить класс каждого аргумента, чтобы знать, какое сообщение отправлять. Добавление инструкции `case`, переключающей классы, позволяет решить проблему отправки правильного сообщения, но вызывает взрывной рост зависимостей.

Подсчитайте количество новых зависимостей в методе `prepare`. Он полагается на конкретные классы, и ничего другого ему не остается. Он зависит от имен этих классов. Он знает имена сообщений, в которых разбирается каждый класс, а также аргументы, требуемые для этих сообщений. Все эти знания повышают степень риска, и теперь многие отдаленные изменения будут иметь побочные эффекты, отражающиеся на данном коде.

Хуже того, этот стиль программирования склонен к саморазмножению. Как только появится еще одно звено подготовки путешествия, вы или следующий программист, занятый этим приложением, добавите к инструкции `case` новую ветвь `when`. В вашем приложении станет появляться все больше подобных методов, осведомленных об именах многих классов и отправляющих конкретные сообщения на основе используемого класса. Логическим исходом этого стиля программирования станет появление жестко заданного, совершенно негибкого приложения, которое в конечном итоге будет легче полностью переписать, чем что-либо в нем менять.

На рис. 5.2 показана новая диаграмма последовательности. До сих пор каждая диаграмма последовательности была проще соответствующего ей кода, но данная диаграмма пугает более сложной структурой. Эта весьма тревожный знак. Диаграммы последовательности всегда должны быть проще представляемого ими кода; если это не так, значит, что-то неладно в самой конструкции.

Скрытые возможности неявной типизации

Ключ к избавлению от зависимостей — понимание того, что поскольку метод `prepare` класса `Trip` работает на получение одного результата, то его аргументы желают сотрудничать для достижения единой цели. Все аргументы находятся здесь по одной и той же причине, и эта причина не связана с базовым классом аргумента.

Не отвлекайтесь на то, чем уже занимается класс каждого аргумента, подумайте вместо этого о том, что нужно методу `prepare`. Если посмотреть с точки

зрения метода `prepare`, проблема решается очень просто. Метод `prepare` хочет подготовить путешествие. Его аргументы готовы для сотрудничества в подготовке путешествия. Конструкция станет проще, если `prepare` просто доверится им в этом деле.

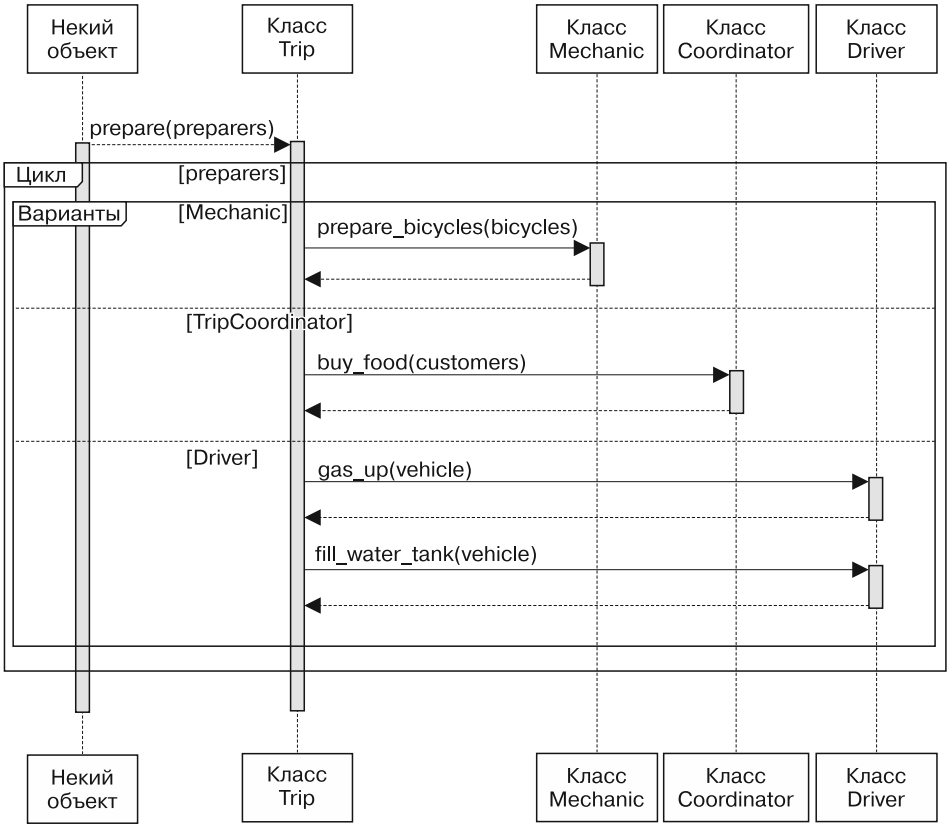

Рис. 5.2. Классу Trip известно слишком многое о конкретных классах и методах

Этот замысел показан на рис. 5.3, где у метода `prepare` отсутствуют предопределенные ожидания насчет класса его аргументов, вместо этого он ожидает, что каждый из них будет тем, кто готовит путешествие (preparer).

Данное предположение выворачивает все наизнанку. Вы освобождаетесь от влияния существующих классов и изобретаете неявную типизацию. Следующим шагом станет решение вопроса о том, какое сообщение метод `prepare` может результативно отправить в адрес каждого `Preparer`. С этой точки зрения ответ очевиден: `prepare_trip` (подготовить путешествие).

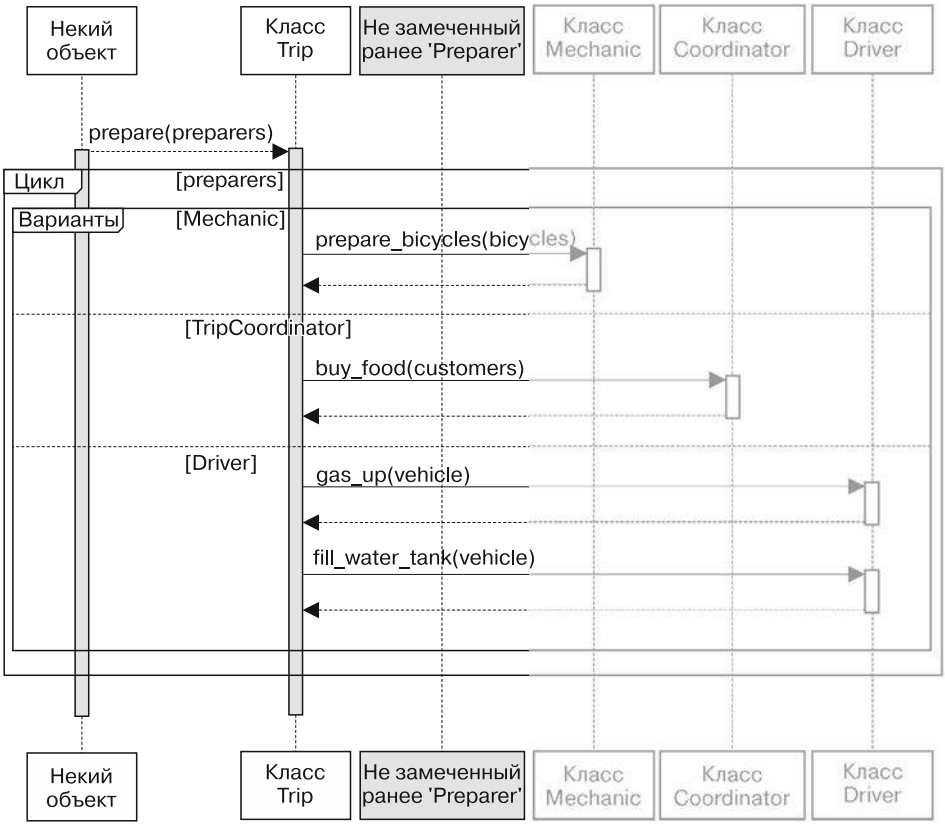

Рис. 5.3. Классу Trip нужно, чтобы каждый аргумент действовал в качестве участника подготовки путешествия

На рис. 5.4 представлено новое сообщение. Метод `prepare` класса `Trip` теперь ожидает, что его аргументами будут некие участники подготовки `Preparers`, способные отвечать на сообщение `prepare_trip`.

Что собой представляет `Preparer`? На данный момент ничего подобного еще не существует, это абстракция, соглашение об открытом интерфейсе на основе замысла. Это плод проектирования.

Объекты, реализующие `prepare_trip`, — это `Preparers`, и наоборот: объекты, взаимодействующие с `Preparers`, нуждаются только в выражении доверия, что в `Preparers` реализуется интерфейс `Preparer`. После того как сложится образ этой базовой абстракции, станет легче вносить поправки в код. `Mechanic`, `TripCoordinator` и `Driver` должны вести себя как `Preparers`; в них должна быть реализована подготовка путешествия `prepare_trip`.

Далее следует код для новой конструкции. Теперь метод `prepare` ожидает, что все его аргументы будут относиться к `Preparers` и что новый интерфейс будет реализован в каждом классе аргументов.

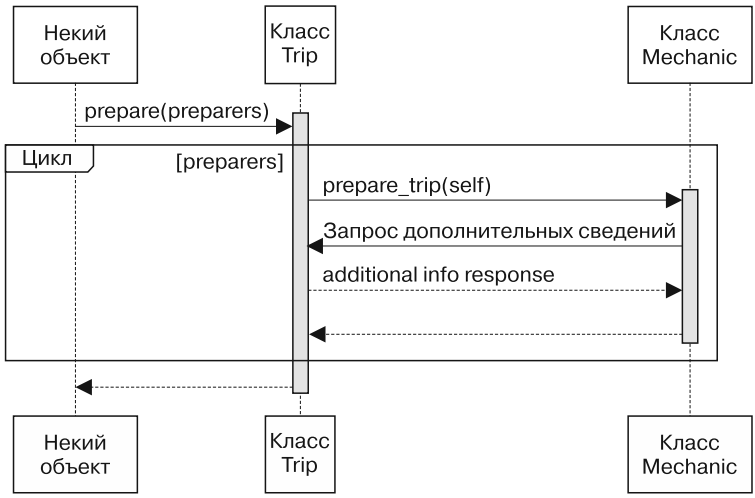

Рис. 5.4. Trip сотрудничает с неявной типизацией участников подготовки путешествия

```
# Подготовка путешествия, Trip, упрощается
class Trip
    attr_reader :bicycles, :customers, :vehicle

    def prepare(preparers)
         preparers.each {|preparer|
         preparer.prepare_trip(self)}
    end
end

# каждый участник подготовки, preparer, рассматривается в виде объекта
# с неявной типизацией, отвечающего на сообщение 'prepare_trip'
class Mechanic
    def prepare_trip(trip)
         trip.bicycles.each {|bicycle|
         prepare_bicycle(bicycle)}
    end

# ...
```

```
end

class TripCoordinator
    def prepare_trip(trip)
        buy_food(trip.customers)
    end

    # ...
end

class Driver
    def prepare_trip(trip)
        vehicle = trip.vehicle
        gas_up(vehicle)
        fill_water_tank(vehicle)
    end
    # ...
end
```

Теперь метод `prepare` может получать новых участников подготовки (`Preparers`) без каких-либо вынужденных изменений; при надобности можно легко создавать дополнительных `Preparers`.

Последствия неявной типизации

Новая реализация обладает привлекательной симметрией, вселяющей уверенность в правильности подхода к проектированию, но последствия неявной типизации простираются гораздо шире.

В исходном примере метод `prepare` зависит от конкретного класса. В самом последнем примере `prepare` зависит от неявной типизации. Путь между этими двумя примерами пролегает сквозь дебри усложненного, перегруженного зависимостями кода.

Конкретность первого примера упрощает его понимание, но вызывает беспокойство при мысли о его расширении. Заключительный альтернативный вариант с неявной типизацией носит более абстрактный характер, он требует приложить немного больше усилий для понимания, но, в свою очередь, предлагает легкость расширения. Теперь после обнаружения возможности применения неявной типизации вы можете вызвать новое поведение из вашего приложения, не меняя существующего кода, а просто превращая в `Preparer` еще один объект и передавая его в метод `prepare` класса `Trip`.

Противоречия между затратами на конкретизацию и затратами на абстракцию носят при объектно-ориентированном проектировании фундаментальный характер. В конкретизированном коде легче разобраться, но его расширение обходится намного дороже. Абстрактный код может сначала показаться менее понятным, но после того, как вы в нем разберетесь, станет намного проще вносить в него изменения. Использование неявной типизации перемещает код по шкале от более конкретного к более абстрактному, облегчая его расширение, но обходя стороной базовый класс «утки».

Способность терпимо относиться к неопределенности класса объекта — отличительная черта уверенного в своих способностях проектировщика. Как только вы начнете относиться к своим объектам учитывая их поведение, а не их принадлежность к тому или иному классу, вы попадете в новый мир выразительных и гибких конструкций.

ПОЛИМОРФИЗМ

Понятие полиморфизма используется в объектно-ориентированном программировании довольно часто, но в качестве определения в повседневной речи оно встречается довольно редко.

Полиморфизм является выражением весьма специфического характера и в зависимости от ваших намерений может использоваться либо для общения, либо для устрашения. В любом случае о нем нужно иметь конкретное представление.

Сперва нужно дать общее определение: «морф» (с греч.) означает форму, «морфизм» — состояние наличия формы, а «полиморфизм» — состояние наличия множества форм. Это слово используется в биологии. Полиморфизм присущ зябликам Дарвина; один вид имеет множество форм.

Полиморфизм в объектно-ориентированном программировании означает способность множества различных объектов откликаться на одно и то же сообщение. Отправителям сообщения не нужно беспокоиться о классе получателя; получатели предоставляют свою собственную версию поведения. Таким образом, одно сообщение имеет множество («поли») форм («морф»).

Существует несколько способов достижения полиморфизма. Один из них, как вы уже, наверное, догадались, — применение неявной типизации. Другими способами являются наследование и разделение ролевого поведения (посредством модулей Ruby), но это темы других глав.

Полиморфные методы соблюдают негласное соглашение; они соглашаются быть взаимозаменяемыми с точки зрения отправителя. Любой объект, реализующий полиморфный метод, может быть заменен любым другим объектом; отправителю сообщения не нужно знать об этой замене или нести за нее ответственность.

Взаимозаменяемость не является результатом магии. Когда используется полиморфизм, вы отвечаете за обеспечение соответствующего поведения всех своих объектов. Этот замысел рассматривается в главе 7.

Написание кода с использованием неявной типизации

Использование неявной типизации зависит от вашей способности распознавать те места, где ваше приложение получит преимущества за счет проникающих сквозь классы интерфейсов. Реализация неявного типа дается сравнительно легко; вам, как проектировщику, следует заметить потребность в такой реализации и абстрагировать интерфейс этой реализации.

В данном разделе содержатся шаблоны, указывающие решения, которыми можно воспользоваться для обнаружения возможности применения неявной типизации.

Обнаружение скрытых возможностей применения неявной типизации

Зачастую нераспознанные неявные типы уже существуют, скрываясь внутри готового кода. На наличие скрытых возможностей применения неявной типизации указывают несколько весьма распространенных шаблонов программирования. Неявными типами можно заменить следующие фрагменты кода:

- инструкции `case`, переключающие код на основе класса;
- `kind_of?` и `is_a?`;
- `responds_to?`.

Инструкции case, переключающие код на основе класса

Наиболее часто встречающийся очевидный шаблон, указывающий на необнаруженную возможность применения неявной типизации, отображен в уже приводимом примере. Инструкция `case` переключает код на основе имен классов объектов из предметной области вашего приложения. Следующий метод `prepare` (такой же, как показанный выше) должен привлекать ваше внимание.

```
1 class Trip
2     attr_reader :bicycles, :customers, :vehicle
3
4     def prepare(preparers)
5          preparers.each {|preparer|
6             case preparer
```

```
            when Mechanic
                 preparer.prepare_bicycles(bicycles)
            when TripCoordinator
                 preparer.buy_food(customers)
            when Driver
                 preparer.gas_up(vehicle)
                 preparer.fill_water_tank(vehicle)
            end
        }
    end
end
```

Когда вам встречается этот шаблон, вы знаете, что все `preparers` должны по большому счету что-либо совместно использовать; они здесь появились благодаря чему-то общему. Исследуйте код и задайтесь вопросом: «Что именно хочет `prepare` от каждого из этих аргументов?»

Ответ на этот вопрос подскажет характер того сообщения, которое вам следует отправить; это сообщение становится отправной точкой определения базового неявного типа.

В данном случае метод `prepare` хочет от своих аргументов добиться подготовки к путешествию. Таким образом, `prepare_trip` (подготовка к путешествию) становится методом в открытом интерфейсе нового неявного типа `Preparer`.

kind_of? и is_a?

Существуют разнообразные способы проверки класса объекта. Один из них — инструкция `case`. Сообщения `kind_of?` и `is_a?` (являющиеся синонимами) также проверяют принадлежность к классу. Перезапись прежнего примера следующим образом не приводит к улучшению кода.

```
if preparer.kind_of?(Mechanic)
     preparer.prepare_bicycles(bicycle)
elsif preparer.kind_of?(TripCoordinator)
     preparer.buy_food(customers)
elsif preparer.kind_of?(Driver)
     preparer.gas_up(vehicle)
     preparer.fill_water_tank(vehicle)
end
```

Использование `kind_of?` ничем не отличается от применения инструкции `case`, переключающей код на основе класса. Это вызывает одинаковые проблемы, и для исправления ситуации нужно использовать одинаковые технологии.

responds_to?

Программисты, понимающие, что не нужно впадать в зависимость от имен классов, но не сделавшие еще шаг навстречу неявной типизации, склонны заменять сообщение `kind_of?` сообщением `responds_to?`.

```
if preparer.responds_to?(:prepare_bicycles)
     preparer.prepare_bicycles(bicycle)
elsif preparer.responds_to?(:buy_food)
     preparer.buy_food(customers)
elsif preparer.responds_to?(:gas_up)
     preparer.gas_up(vehicle)
     preparer.fill_water_tank(vehicle)
end
```

Несмотря на незначительное сокращение количества зависимостей, их в коде остается все же слишком много. От имен классов удалось избавиться, но код по-прежнему сильно привязан к классу. Чем отличается то, что объект будет знать о `prepare_bicycles`, от того, что он будет знать о `Mechanic`? Не обольщайтесь тем, что вам удалось избавиться от явных ссылок на класс. В этом примере по-прежнему ожидаются вполне конкретные классы.

Даже если складывается такая ситуация, при которой `prepare_bicycles` или `buy_food` реализуются более чем одним классом, этот шаблон кода все равно содержит ненужные зависимости; он контролирует другие объекты, вместо того чтобы им доверять.

Внедрение доверия в использование неявной типизации

Использование `kind_of?`, `is_a?`, `responds_to?` и инструкций `case`, переключающих код на основе классов, свидетельствует о присутствии необнаруженных возможностей применения неявной типизации. В каждом из этих случаев код как бы говорит: «Я знаю, кто вы, и поэтому *знаю, что вы делаете*». Это знание свидетельствует о дефиците доверия между взаимодействующими объектами и словно камень на шее ваших объектов. Оно вносит зависимости, существенно затрудняющие внесение в код изменений.

Как и при нарушении закона Деметры, этот стиль кода свидетельствует о том, что вы еще не обнаружили открытый интерфейс объекта. Тот факт, что неза-

меченный объект имеет отношение к неявной типизации, а не к конкретному классу, не имеет абсолютно никакого значения; здесь главное — интерфейс, а не класс объекта, который его реализует.

Гибкие приложения создаются на основе объектов, работающих на доверии; а сделать объекты заслуживающими доверия — ваша задача. Когда вы видите эти шаблоны кода, сосредоточьтесь на ожиданиях вызывающего кода и воспользуйтесь ими для того, чтобы отыскать возможность применения неявной типизации. С прицелом на неявный тип определите его интерфейс, реализуйте этот интерфейс там, где это необходимо, а затем доверьтесь правильному поведению его реализаторов.

Документирование неявных типов

Простейшая разновидность неявного типа существует только в виде соглашения о его открытом интерфейсе. Именно такая разновидность, когда несколько различных классов реализуют метод `prepare_trip` (в силу чего могут рассматриваться в качестве `Preparers`), и создается в примерах кода данной главы.

Неявный тип `Preparer` и его открытый интерфейс считаются четко выраженной частью конструкции, но виртуальной частью кода. `Preparers` являются абстракцией (и становятся инструментом проектирования), но эта же самая абстракция делает неявный тип менее очевидным в коде.

При создании неявных типов их открытые интерфейсы следует документировать и тестировать. К счастью, лучшей документации, чем хорошие тесты, не найти, поэтому полдела уже сделано; осталось только написать тесты.

Дополнительные сведения о тестировании неявных типов даются в главе 9.

Распределение кода между «утками»

В этой главе «утками» `Preparer` предоставляются особые для каждого класса версии поведения, требуемые их интерфейсом. Метод `prepare_trip` реализуется в `Mechanic`, в `Driver` и в `TripCoordinator`. Общей является только сигнатура этого метода. Совместно используется лишь интерфейс, а не реализация.

Приступив к использованию неявных типов, вы поймете, что реализующие их классы зачастую нуждаются в разделении общего поведения. Написание «уток», разделяющих код, — одна из тем, рассматриваемых в главе 7.

Мудрый подход к выбору «уток»

Из каждого показанного ранее примера, несомненно, следует, что не нужно использовать `kind_of?` или `responds_to?` для решения, какое сообщение должен отправить объект. Кроме того, вы не должны выискивать области получающего кода, выполняющего именно эту задачу.

Следующий код является примером из среды Ruby on Rails (active_record/relations/finder_methods.rb). Этот пример заведомо использует классы для принятия решения о том, как распорядиться своими входящими данными, то есть применяет технологию, совершенно противоположную тем указаниям, которые были даны выше. Показанный ниже метод `first` точно определяет, как себя вести на основании класса своего аргумента `args`.

Если отправка сообщения на основании класса получаемого объекта столь пагубно влияет на ваше приложение, то почему же это код признается приемлемым?

```
# Удобная обертка для <tt>find(:first, *args)</tt>.
# You can pass in all the same arguments to this
# method as you can to <tt>find(:first)</tt>.
def first(*args)
    if args.any?
        if args.first.kind_of?(Integer) ||
                (loaded? && !args.first.kind_of?(Hash))
             to_a.first(*args)
        else
             apply_finder_options(args.first).first
        end
    else
         find_first
    end
end
```

Главное отличие этого примера от предыдущих заключается в стабильности проверяемых классов. Когда метод `first` зависит от `Integer` и `Hash`, то он зависит от корневых классов Ruby, которые намного стабильнее его самого. Вероятность того, что `Integer` или `Hash` изменятся таким образом, что это заставит вносить изменения в метод `first`, ничтожно мала. Эта зависимость не вызывает никаких опасений. Возможно, где-то в этом коде и прячется неявный тип, но его поиск и реализация вряд ли снизят общие затраты на приложение.

Из данного примера можно понять, что решение о создании нового неявного типа должно быть взвешенным. Цель проектирования заключается в снижении затрат, именно этим критерием следует руководствоваться в каждой си-

туации. Если создание неявного типа сократит число нестабильных зависимостей, его нужно создавать. Подходите к делу с трезвым расчетом.

Подразумеваемая в предыдущем примере «утка» распространяется на `Integer` и `Hash`, следовательно, ее реализация потребует внесения изменений в базовые классы Ruby. Изменение базовых классов называется *обезьяньим патчем* (monkey patching) и является замечательной особенностью Ruby, которая может стать в неопытных руках весьма опасной.

Одно дело — реализовывать неявную типизацию сквозь свои собственные классы, и совершенно другое — вносить изменения в базовые классы Ruby для введения новых неявных типов (другие побочные эффекты и более высокие риски). Но ни одно из этих соображений не должно в случае необходимости помешать использованию в Ruby обезьяньего патча, однако вам следует четко обосновать такое проектное решение. И доказательства должны быть весомыми.

Преодоление страха применения неявной типизации

До сих пор в этой главе тема противостояния между динамической и статической типизацией деликатно обходилась стороной, но далее избегать этого вопроса нельзя. Если у вас имеется опыт программирования с использованием статической типизации и идея неявной типизации вызывает у вас тревогу, то этот раздел — для вас.

Если же этот спор вам непонятен, если вам посчастливилось использовать Ruby и вы согласны с предыдущими рассуждениями насчет неявной типизации, можете просто пролистать этот раздел без опасения пропустить что-либо важное. Но вы можете посчитать следующий материал полезным, если вам нужно парировать аргументы ваших друзей — приверженцев статической типизации.

Подрыв неявной типизации с помощью статической типизации

Ранее в данной главе *тип* был определен как категория содержимого переменной. Языки программирования обладают либо *статической*, либо *динамической* типизацией. Большинство (хотя и не все) статически типизированных языков требуют явного объявления типа каждой переменной и каждого параметра метода. Динамически типизированные языки опускают это требование; они позволяют вам помещать любое значение в любую переменную и передавать

любой аргумент любому методу без дальнейшего объявления. Вполне очевидно, что Ruby обладает динамической типизацией.

Ставка на динамическую типизацию заставляет некоторых программистов испытывать дискомфорт. Для одних причина в недостатке опыта, другие же верят в то, что статическая типизация более надежна.

Проблема недостатка опыта со временем проходит сама собой, а вот вера в то, что статическая типизация предпочтительнее, зачастую неизменна. Программисты, которых пугает динамическая типизация, стремятся в своем коде к проверке классов объектов; эти чрезмерные проверки подрывают эффективность динамической типизации, делая использование неявной типизации невозможным.

При появлении новых классов методы, не способные к правильному поведению без сведений о принадлежности их аргументов к тем или иным классам, дадут сбой (с ошибками несоответствия типов). Программисты, верящие в статическую типизацию, приводят эти особенности в качестве доказательств в необходимости дополнительной проверки типов. После добавления дополнительных проверок код становится менее гибким и еще более зависимым от класса. Новые зависимости становятся причинами дополнительных сбоев из-за типов, и программисты реагируют на это путем добавления дополнительных проверок типов. Тем, кто попал в этот круговорот, конечно, будет трудно поверить, что выходом станет полное удаление проверки типов.

Применение неявной типизации — спасение из этой ловушки. Она устраняет зависимости от класса и тем самым позволяет избегать последующих сбоев, связанных с типами. Она открывает стабильные абстракции, от которых совершенно безопасно может зависеть ваш код.

Сравнение статической и динамической типизации

В надежде развеять любые опасения, которые мешают вам стать приверженцем динамической типизации, сравним динамическую и статическую типизацию.

И статическая, и динамическая типизация дают определенные обещания, и у каждой из них есть свои плюсы и минусы.

Приверженцы статической типизации ссылаются на следующие ее качества:

- компилятор обнаруживает ошибки, связанные с типами данных в ходе компиляции;
- наглядная информация о типе служит в качестве документации;
- откомпилированный код оптимизирован под быстрое выполнение.

Все эти качества становятся сильными сторонами языка программирования, только если принять следующий набор соответствующих предположений:

- если компилятор не производит проверок типа, возникают ошибки в ходе выполнения программы;
- программисты в противном случае не смогут разобраться в коде; они не в состоянии вывести тип объекта, исходя из его контекста;
- без данных оптимизаций приложение будет выполняться слишком медленно.

Сторонники динамической типизации перечисляют следующие ее качества:

- код интерпретируется и может быть загружен в динамическом режиме; цикл компиляции-сборки отсутствует;
- в исходном коде отсутствует явно выраженная информация о типе;
- легче осуществляется метапрограммирование.

Эти качества обретают силу при принятии следующего набора предположений:

- в целом разработка приложения без цикла компиляции-сборки проводится быстрее;
- программисты считают, что без объявлений типа код проще понять; они могут вывести тип объекта из его контекста;
- метапрограммирование является востребованным свойством языка.

Вступление на путь динамической типизации

Некоторые из этих качеств и предположений основаны на эмпирических фактах и легко поддаются оценке. Несомненно, для отдельных приложений качественно оптимизированный статически типизированный код превзойдет динамически типизированную реализацию. Когда динамически типизированное приложение не может быть настроено на достаточно быстрое выполнение, альтернативой становится приложение со статической типизацией. Тут уж ничего не поделаешь.

А вот аргументы насчет ценности объявления типов в качестве документации носят более субъективный характер. Программисты, имеющие опыт работы с динамической типизацией, считают, что объявления типов носят отвлекающий характер. А те, кто привык к статической типизации, из-за дефицита информации о типах могут потерять ориентацию. Если вы ранее работали со статически

типизированным языком, таким как Java или C++, и чувствуете, что отсутствие в Ruby явного объявления типов выбивает у вас почву из-под ног, то вам следует проявить упорство. Отзывы других людей дают основание предположить, что после того, как у вас выработается привычка работать с этим языком, окажется, что менее подробный синтаксис легче прочитать, написать и усвоить.

Насчет метапрограммирования (то есть написания кода, который пишет код) мнения программистов расходятся, и какой точки зрения они придерживаются (за или против), зависит от их личного опыта. Если вам приходилось решать серьезную проблему с помощью простого, элегантного фрагмента метапрограммирования, вы агитируете в его поддержку. Если же вам приходилось сталкиваться с непростой задачей отладки чересчур запутанного, совершенно неочевидного и, возможно, ненужного фрагмента метапрограммирования, то у вас могло сложиться представление о нем как о новом способе пытки (то есть как об инструменте, от применения которого нужно отказаться раз и навсегда).

Метапрограммирование как скальпель: в неумелых руках нанесет вред, в руках же опытных программистов торит чудеса (рис. 5.5). Метапрограммирование дарует невероятную эффективность и требует высокого уровня ответственности. Облегчение метапрограммирования — сильный аргумент в пользу динамической типизации.

Рис. 5.5. Острые инструменты полезны, но не для всех

Рассмотрим два остальных качества — проверку типов в ходе компиляции, присущую статической типизации, и отсутствие цикла компиляции-сборки при динамической типизации. Приверженцы статической типизации заявляют, что предохранение от неожиданных ошибок, связанных с типами в ходе выполнения программы, настолько необходимое и ценное качество, что выгода от его наличия перевешивает более эффективный процесс программирования, достигаемый за счет удаления компилятора.

Эта аргументация исходит из предположений, что при статической типизации:

- компилятор действительно *может* уберечь вас от случайных ошибок;
- без помощи компилятора эти ошибки типов обязательно *будут* допущены.

Если у вас многолетний стаж программирования на языках со статической типизацией, то эти предположения вы можете считать прописными истинами. Однако динамическая типизация рассматривается здесь, чтобы поколебать основы вашей веры. На эти аргументы динамическая типизация может ответить: «Я не могу» и «Они не будут допущены».

Компилятор *не может* уберечь вас от случайных ошибок, связанных с типами. Таким ошибкам подвержен любой язык, допускающий приведение переменной к новому типу. Как только вы приступаете к приведению типов, ремни безопасности расстегиваются: компилятор самоустраняется и вам приходится полагаться в деле предотвращения ошибок типов на самого себя. Качество кода определяется только там, где он был протестирован, а от сбоев в ходе выполнения вы все равно не застрахованы. Мысль о том, что статическая типизация гарантирует безопасность (насколько бы утешительной она ни была), — это иллюзия.

Более того, способен компилятор вас защитить или нет, не имеет никакого значения; вы не нуждаетесь в защите. В реальном мире компилятор предотвращает ошибки, связанные с типами, которые в ходе выполнения программы *практически никогда не возникают*. Их просто не бывает. Это не значит, что вы никогда не столкнетесь с ошибками, связанными с типами, в ходе выполнения программы. Вряд ли найдутся программисты, не отправлявшие сообщение неинициализированной переменной или не предполагавшие, что у массива есть элементы, когда он на самом деле был пустым. Но компилятор не может предотвратить обнаружения в ходе выполнения программы, что `nil` не понимает полученного сообщения. Такие ошибки с равной долей вероятности случаются в обеих системах типизации.

Динамическая типизация позволяет вам сменить проверку типов в ходе компиляции, являющуюся серьезным ограничением с высокими затратами и сомнительными выгодами, на огромные выгоды, касающиеся эффективности, предоставляемые за счет удаления цикла компиляции-сборки. Этот обмен дорогого стоит. Так воспользуйтесь им.

Неявная типизация строится на динамической типизации, и для использования неявной типизации нужно воспользоваться этим динамизмом.

Выводы

Сообщения являются основой, вокруг которой выстраиваются объектно-ориентированные приложения, и они передаются между объектами по открытым интерфейсам. Неявная типизация отвязывает эти открытые интерфейсы от конкретных классов, создавая виртуальные типы, определяющие, что именно они делают, вместо того чтобы определять, кто они есть на самом деле.

Неявная типизация раскрывает исходные абстракции, которые в противном случае остались бы незамеченными. Зависимость от этих абстракций сокращает риски и придает дополнительную гибкость, что снижает расходы на сопровождение вашего приложения и упрощает внесение в него изменений.

Глава 6

Получение поведения через наследование

Качественно спроектированные приложения построены из кода, пригодного для многократного использования.

По сути, многократно используемыми являются небольшие, заслуживающие доверия автономные объекты с минимальным контекстом, четко выраженными интерфейсами и внедренными зависимостями. До сих пор в данной книге основное внимание уделялось созданию объектов исключительно с такими качествами.

Однако у большинства объектно-ориентированных языков имеется еще одна технология совместного использования кода, встроенная в сам синтаксис языка, — *наследование*.

В этой главе приводится подробный пример написания кода, надлежащим образом использующего наследование. Его задача — обучить вас созданию технически обоснованной иерархии наследования, а цель — помочь вам решить, должны ли вы создавать подобную иерархию.

Когда вы разберетесь с тем, как используется классическое наследование, усвоенный материал можно легко перенести на другие механизмы наследования. Наследованию таким образом посвящены две главы. Текущая глава

содержит руководство по написанию наследуемого кода. В главе 7 эти технологии распространяются на задачи разделения кода с помощью использования Ruby-модулей.

Основные сведения о классическом наследовании

Идея наследования может показаться сложной, но при всей своей сложности она является упрощающей абстракцией. По сути, наследование представляет собой механизм *отправки сообщений в автоматическом режиме*. Оно определяет путь пересылки непонятных сообщений. При этом создаются взаимоотношения, при которых, если объект не способен ответить на полученное сообщение, он пересылает его другому объекту. Вам не нужно создавать код для явной отправки сообщения, вместо этого вы определяете отношения наследования между двумя объектами — и пересылка происходит автоматически.

В классическом наследовании эти взаимоотношения определяются путем создания подклассов. Сообщения пересылаются от подкласса родительскому классу, а общий код определяется в иерархии классов.

Понятие *«классический»* происходит от слова *«класс»*, а не является намеком на архаичность технологии, оно служит для того, чтобы отличить этот механизм, использующий родительские классы и подклассы, от других технологий наследования. В JavaScript, к примеру, есть прототипное наследование, а в Ruby есть *модули* (которые более подробно рассматриваются в следующей главе), и обе эти технологии предоставляют способ совместного использования кода через автоматическую пересылку сообщений.

Когда нужно, а когда не следует применять наследование, лучше всего понять на примере. Пример, приводимый в данной главе, раскрывает основные положения технологии классического наследования. Он начинается с одного класса, и в нем проводится несколько реструктуризаций для получения отвечающего поставленным целям набора подклассов. На каждом этапе проводится небольшая по объему и легко выполнимая работа, но для того, чтобы проиллюстрировать все эти идеи, кода набралось на целую главу.

Как определить, где требуется наследование

Сперва нужно определить, где может пригодиться наследование. В этом разделе показывается, как узнать о наличии проблем, решаемых с помощью наследования.

Предположим, что компания FastFeet проводит велосипедные туры. Дорожные велосипеды имеют небольшой вес, изогнутый руль и тонкие шины, предназначенные для дорог с твердым покрытием. Дорожный велосипед показан на рис. 6.1.

Рис. 6.1. Легкий дорожный велосипед с изогнутым рулем и тонкими шинами

Механики отвечают за то, чтобы велосипеды были на ходу (независимо от количества клиентов), и в каждый тур они берут арсенал запчастей. Какие запчасти им необходимы, зависит от того, какими велосипедами пользуются клиенты.

Начнем с конкретного класса

Приложение для компании FastFeet уже имеет класс `Bicycle`, показанный ниже. Каждый дорожный велосипед, подготавливаемый к туру, представлен экземпляром этого класса.

У велосипедов общий размер, цвет ленты, которой обмотан руль, размер шины и тип цепи. Шины и цепи — неотъемлемая часть велосипеда, поэтому они всегда должны быть в наборе запчастей. Лента на руле может показаться не таким необходимым компонентом, но на самом деле в ней не меньше потребности, ведь ни один уважающий себя велосипедист не станет мириться с грязной или

рваной лентой на руле, поэтому механики должны запастись лентой соответствующего цвета.

```
class Bicycle
 attr_reader :size, :tape_color

 def initialize(args)
   @size       = args[:size]
   @tape_color = args[:tape_color]
 end

# у каждого велосипеда единые установки относительно
# шин и размера цепи
def spares
 { chain:    '10-speed',
 tire_size:  '23',
 tape_color: tape_color}
end

# Множество других методов...
end

bike = Bicycle.new(
 size:       'M',
 tape_color: 'red' )

bike.size    # -> 'M'
bike.spares
# -> {:tire_size  => "23",
#     :chain      => "10-speed",
#     :tape_color => "red"}
```

Экземпляры `Bicycle` могут отвечать за сообщения о запчастях `spares`, о размере `size` и о цвете ленты `tape_color`, а `Mechanic` может выяснить, какие запчасти брать, запрашивая каждый `Bicycle` о его запчастях `spares`. Если не обращать внимания на тот факт, что метод `spares` встраивает исходные строки непосредственно в себя, показанный выше код весьма рационален. Похоже, этой модели велосипеда не хватает нескольких болтов, не имеющих непосредственного отношения к самой *езде*, но она будет дополнена в ходе упражнения, выполняемого в этой главе.

Этот класс работает достаточно хорошо до первых изменений. Представим, что компания FastFeet начала организовывать туры на горных велосипедах.

Горные и дорожные велосипеды во многом похожи, но имеют и ярко выраженные различия. Горные велосипеды приспособлены для езды по грунтовым дорогам, а не по дорогам с твердым покрытием. У них прочные рамы, толстые шины, прямые рули (с резиновыми рукоятками вместо ленты) и подвеска. У велосипеда на рис. 6.2 имеется только передняя подвеска, но у некоторых горных велосипедов есть также задняя подвеска («полная» подвеска).

Рис. 6.2. Более крепкий горный велосипед с прямым рулем, передней подвеской и толстыми шинами

Ваша задача как проектировщика заключается в добавлении поддержки горных велосипедов к приложению для компании FastFeet.

Многое из необходимого для этого поведения уже имеется; у горных велосипедов общий размер, а также размеры цепи и шин. Дорожные велосипеды отличаются от горных только тем, что дорожным велосипедам нужна лента на руль, а у горных велосипедов есть подвеска.

Встраивание нескольких типов

Когда уже существующий конкретный класс содержит более одной нужной вам формы поведения, возникает желание решить эту задачу путем добавления кода к этому классу. В следующем примере именно так и делается, он изменяет существующий класс `Bicycle` таким образом, что его работа распространяется как на дорожные, так и на горные велосипеды.

Как показано ниже, в код добавлены три новые переменные, а также соответствующие средства доступа к ним. В новых переменных `front_shock` и `rear_shock` содержатся запчасти, подходящие только для горного велосипеда. В новой переменной `style` определяется, какие пункты появляются в списке запчастей. Каждая из этих новых переменных обрабатывается с помощью метода `initialize`.

Код для добавления этих трех переменных весьма прост и даже неинтересен; более интересные изменения происходят в `spares`. Теперь метод `spares` содержит инструкцию `if`, с помощью которой проверяется содержимое переменной `style`. Эта переменная нужна для разделения экземпляров `Bicycle` на две категории — на те, чей стиль обозначен как `:road`, и на те, чей стиль обозначен как-то по-другому.

Если при просмотре кода вы испытываете настороженность, то, уверяю вас, она вскоре развеется. Этот пример всего лишь обходной путь, иллюстрирующий *антишаблон проектирования*, то есть весьма распространенный шаблон, который кажется полезным, но на самом деле является вредным и для которого имеется общеизвестная альтернатива.

ПРИМЕЧАНИЕ

Если вас будут смущать показанные ниже размеры шин, знайте, что по сложившейся традиции размеры велосипедных шин обозначаются по-разному. Дорожные велосипеды появились в Европе, и для их обозначения используется метрическая система; 23-миллиметровая шина немного меньше дюймовой. Горные велосипеды появились в США, и размер их шин измеряется в дюймах. В приведенном ниже примере 2,1-дюймовая шина горного велосипеда более чем в два раза шире 23-миллиметровой шины дорожного велосипеда.

```
class Bicycle
 attr_reader :style, :size, :tape_color,
             :front_shock, :rear_shock

 def initialize(args)
   @style       = args[:style]
   @size        = args[:size]
   @tape_color  = args[:tape_color]
   @front_shock = args[:front_shock]
   @rear_shock  = args[:rear_shock]
 end

 # Проверка «стиля» проходит по скользкой дорожке
 def spares
  if style == :road
    { chain:      '10-speed',
      tire_size:  '23', # миллиметры
      tape_color: tape_color }
  else
    { chain:      '10-speed',
```

```
      tire_size:  '2.1', # дюймы
      rear_shock: rear_shock }
  end
 end
end

bike = Bicycle.new(
        style:         :mountain,
        size:          'S',
        front_shock:   'Manitou',
        rear_shock:    'Fox')

bike.spares
# -> {:tire_size  => "2.1",
#     :chain      => "10-speed",
#     :rear_shock => 'Fox'}
```

Этот код принимает решение относительно запчастей на основе значения, содержащегося в переменной `style`; такое структурирование кода влечет за собой множество негативных последствий. При добавлении нового стиля придется изменять инструкцию `if`. При написании кода, не обеспечивающего полную ответственность, где последним вариантом будет действие по умолчанию (как это сделано в показанном выше коде), неожиданный стиль вызовет *какие-то* действия, но, вероятно, совсем не те, что вы ожидаете. А еще метод `spares` начинается с того, что в нем содержатся встроенные строки, используемые по умолчанию; некоторые из них на данный момент дублируются в каждой из сторон инструкции `if`.

Класс `Bicycle` имеет исходный открытый интерфейс, включающий `spares`, `size` и другие отдельные детали. Метод `size` по-прежнему работает, `spares` в целом тоже работает, а вот на методы, относящиеся к деталям, теперь положиться уже нельзя. Для любого конкретного экземпляра `Bicycle` невозможно предсказать, будет ли инициализирована конкретная запчасть. В отношении объектов, хранящихся в экземпляре `Bicycle`, может, к примеру, возникать соблазн проверки стиля перед отправкой им цветной ленты `tape_color` или заднего амортизатора `rear_shock`.

Это не самый подходящий код для начала работы, и данное изменение не приведет к его улучшению.

Исходный класс `Bicycle` далек от совершенства, но его недостатки были скрыты за счет инкапсуляции внутри класса. Появившиеся новые изъяны имеют более серьезные последствия. У класса `Bicycle` теперь имеется более одной

обязанности, включая то, что по многим причинам может измениться; в данном состоянии он не может быть использован многократно.

Этот шаблон программирования может вызвать разочарование, но и в нем есть определенная ценность. Он служит яркой иллюстрацией антишаблона, вместо которого, как уже было замечено, предлагается более удачная конструкция.

В этом коде содержится инструкция `if`, которая проверяет *атрибут, содержащий собственную категорию* для определения, какое сообщение отправить в *собственный* адрес. Это должно вызвать у вас воспоминания о шаблоне, рассмотренном в предыдущей главе, который касался неявной типизации, где была показана инструкция `if`, проверяющая *класс объекта* для определения, какое именно сообщение отправлять *этому объекту*.

В обоих этих шаблонах объект принимает решение о том, какое сообщение отправить, основываясь на категории получателя. О *классе объекта* можно думать просто как об особом случае *атрибута, содержащего собственную категорию*; рассуждая таким образом, можно посчитать эти шаблоны одинаковыми. В каждом случае, если бы отправитель мог говорить, то он бы сказал: «Я знаю, *кто ты есть на самом деле*, и поэтому я знаю, *что мне делать*». Это знание является зависимостью, повышающей затраты на внесение изменения.

Такой шаблон не может не настораживать. Его присутствие (иногда по наивности, иногда вполне обоснованно) может стать для вашей конструкции весьма разорительным. В главе 5 этот шаблон использовался для обнаружения нераспознанной возможности применения неявной типизации, а здесь шаблон выступает признаком нераспознанного подтипа, более известного как подкласс.

Поиск встраиваемых типов

Инструкция `if` в показанном выше методе `spares` осуществляет передачу управления на основе значения переменной по имени `style`, но естественнее было бы назвать эту переменную `type` или `category`. Переменные с именами такого рода позволяют вам заметить положенный в основу шаблон. Слова *«тип»* и *«категория»* имеют опасное сходство со словами, которые вы будете использовать при описании класса. В конце концов, что же такое класс, если не категория или тип?

Переменная `style` эффективно делит экземпляры класса `Bicycle` на две разновидности, которые совместно используют основную часть поведения, но различаются по характеристикам стиля. Часть поведения, определенного в классе `Bicycle`, применяется ко всем велосипедам, другая часть используется только к до-

рожным велосипедам, а еще одна часть — только к горным велосипедам. Один этот класс содержит несколько различных, но родственных типов.

В общем, это та самая проблема, которая решается с помощью наследования. Она касается типов с высокой степенью родства, которые имеют общие черты поведения, но разнятся по некоторым показателям.

Выбор наследования

Перед тем как перейти к следующему примеру, стоит изучить наследование более подробно. Наследование может показаться тайным искусством, но, как и большинство идей проектирования, при взгляде на него под правильным углом оно оказывается не таким уж и сложным.

Само собой разумеется, что объекты получают сообщения. Независимо от сложности кода объект в конечном счете обрабатывает любое сообщение одним из двух способов: либо сам дает ответ, либо передает сообщение другому объекту, чтобы тот дал ответ.

Наследование — это способ определения двух объектов как имеющих родственную связь, при которой, когда первый получает непонятное ему сообщение, он *автоматически* переправляет (или делегирует) его второму. Ничего сложного.

Слово *«наследование»* навевает образ генеалогического древа, где прародители обозначены верхними ветвями, а потомки — нижними. Но этот образ может ввести в заблуждение. В реальном мире чаще всего у потомков есть два предка (например, у вас, скорее всего, два родителя). Языки, позволяющие объектам иметь нескольких родителей, называют языками со *множественным наследованием*, и разработчики этих языков сталкиваются с интересными проблемами. Когда объект, имеющий нескольких родителей, получает непонятное ему сообщение, кому из родителей он должен его переслать? Если реализовать сообщение могут оба его родителя, кому отдать предпочтение? Нетрудно догадаться, что проблемы растут как снежный ком.

Многие объектно-ориентированные языки обходят эти проблемы, предоставляя *уникальное наследование*, когда подклассу разрешено иметь только один родительский класс. Именно этому правилу следует Ruby — в нем применяется уникальное наследование. У родительского класса может быть множество подклассов, но каждому подклассу разрешено иметь только один родительский класс.

Если рассматривать классическое наследование, то пересылка сообщения осуществляется между *классами*. Поскольку неявные типы характерны для классов, для общего поведения они не используют классическое наследование.

Неявные типы взаимодействуют с помощью модулей Ruby (более подробно они рассмотрены в следующей главе).

Даже если вам никогда не приходилось явным образом создавать иерархию классов, вы все равно пользовались наследованием. При определении нового класса без указания его родительского класса Ruby автоматически назначает вашему классу родительский класс `Object`. Каждый создаваемый вами класс по определению является чьим-нибудь подклассом.

Вы также получаете преимущества от автоматической пересылки сообщений родительским классам. Когда объект получает непонятное ему сообщение, Ruby автоматически пересылает его вверх по цепочке родительских классов в поисках соответствующей реализации метода. На рис. 6.3 дан простой пример, показывающий, как объекты Ruby откликаются на сообщение `nil?`.

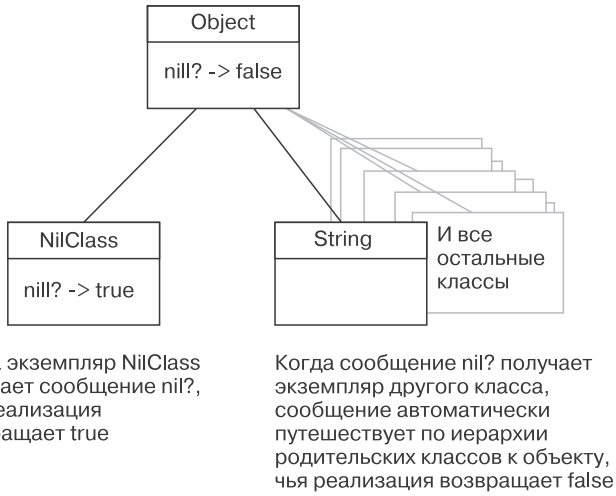

Рис. 6.3. NilClass на сообщение nil? дает ответ true, String (и все остальные) дает ответ false

Следует напомнить, что в Ruby `nil` является экземпляром класса `NilClass`, то есть это объект, ничем не отличающийся от всех остальных объектов. В Ruby имеется две реализации `nil?`: одна находится в классе `NilClass`, а другая — в классе `Object`. Реализация в `NilClass`, безусловно, возвращает `true`, а та реализация, которая имеется в Object, возвращает `false`.

Когда сообщение `nil?` отправляется экземпляру `NilClass`, на него, очевидно, следует ответ `true`. Когда сообщение `nil?` отправляется в какой-нибудь другой адрес, оно путешествует по иерархии от одного родительского класса к другому,

пока не достигнет `Object`, где будет вызвана реализация, дающая ответ `false`. Этот `nil` отвечает, что значение в нем отсутствует (nil), а все остальные объекты отвечают, что в них есть значение. Такое простое решение служит хорошей иллюстрацией эффективности и полезности наследования.

Тот факт, что непонятное сообщение пересылается вверх по иерархии родительских классов, означает, что подклассы всегда являются еще и своими родительским классами и даже *более того*. Экземпляр `String` *это* `String`, но это также и `Object`. Предполагается, что каждый `String`-объект содержит весь открытый интерфейс класса `Object` и должен отвечать соответствующим образом на любое сообщение, определенное в этом интерфейсе. Таким образом, подклассы являются конкретизацией своих родительских классов.

Текущий экземпляр `Bicycle` имеет внутри класса несколько встроенных типов. Настала пора отказаться от этого кода и вернуться к исходной версии `Bicycle`. Вполне возможно, что горный велосипед является конкретизацией `Bicycle`. Эти проблемы проектирования могут быть решены с использованием наследования.

Прорисовка наследственных связей

Точно так же, как в главе 4 для демонстрации связей с помощью передачи сообщений использовались UML-диаграммы последовательностей, здесь для иллюстрации взаимоотношений классов могут применяться UML-диаграммы классов.

Диаграмма класса показана на рис. 6.4. Классы обозначены прямоугольниками. Соединительная линия показывает связи классов. Пустой треугольник означает, что связь является наследованием. Указующая вершина треугольника примыкает к прямоугольнику, содержащему родительский класс. Таким образом, на рисунке показано, что для `MountainBike` родительским классом является `Bicycle`.

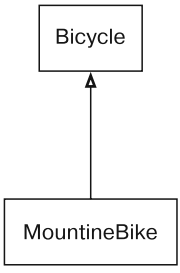

Рис. 6.4. MountainBike является подклассом Bicycle

Ошибочное применение наследования

Следуя правилу, что пользы больше от самого процесса путешествия, нежели от прибытия в конечную точку маршрута, и что к ошибкам снисходительнее относятся, когда их допустил кто-то другой, в следующем разделе продолжим рассмотрение кода, повторять который крайне нежелательно. Код показывает трудности, с которыми довольно часто сталкиваются новички. Если у вас богатый опыт в работе с наследованием и вы чувствуете, что освоили эту технологию, то можете данный раздел пропустить. Но если вы новичок или все ваши попытки освоения наследования оказались тщетными, отнеситесь к материалу серьезно.

Следующий код показывает первую попытку превратить `MountainBike` в подкласс. Этот новый подкласс является прямым наследником исходного класса `Bicycle`. В нем реализованы два метода — `initialize` и `spares`. Оба они уже реализованы в `Bicycle`, поэтому подкласс `MountainBike` их *переопределяет*.

В следующем фрагменте кода каждый из переопределенных методов отправляет сообщение `super`.

```
1 class MountainBike < Bicycle
2  attr_reader :front_shock, :rear_shock
3
4  def initialize(args)
5     @front_shock = args[:front_shock]
6     @rear_shock  = args[:rear_shock]
7     super(args)
8  end
9
10 def spares
11    super.merge(rear_shock: rear_shock)
12 end
13 end
```

Отправка `super` в любом методе передает это сообщение вверх по цепочке родительских классов. Так, к примеру, отправка `super` в принадлежащем `MountainBike` методе `initialize` (строка 7) вызывает метод `initialize` его родительского класса `Bicycle`.

Подгонка нового класса `MountainBike` непосредственно под существующий класс `Bicycle` была делом рискованным, и при выполнении кода предсказуемо выявляются недостатки. У экземпляров `MountainBike` имеется несколько вариантов поведения, в которых нет смысла. В следующем примере показано, что

получится, если вы запросите у `MountainBike` его размер и запчасти. Свой размер он укажет правильно, но сообщит о том, что у него тонкие шины, и даст понять, что ему нужна лента для обмотки руля; оба последних сообщения будут неверными.

```
mountain_bike = MountainBike.new(
                    size: 'S',
                    front_shock: 'Manitou',
                    rear_shock: 'Fox')

mountain_bike.size # -> 'S'

mountain_bike.spares
# -> {:tire_size  => "23", <- неверно!
#     :chain      => "10-speed",
#     :tape_color  => nil, <- неприменимо
#     :front_shock => 'Manitou',
#     :rear_shock  => "Fox"}
```

Неудивительно, что экземпляры класса `MountainBike` содержат смесь поведения дорожного и горного велосипедов. Класс `Bicycle` является конкретным классом, который создавался не как подкласс. В нем сочетается поведение, являющееся общим для всех велосипедов, с поведением, специфичным для дорожных велосипедов. Когда `MountainBike` выводится из класса `Bicycle`, наследуется все поведение (общее и специфическое) независимо от того, применимо оно или нет.

Чтобы проиллюстрировать эту мысль, на рис. 6.5 допущены вольности в составлении диаграмм классов, показано встраивание поведения дорожного велосипеда в класс `Bicycle`. Способ построения этого кода заставляет класс `MountainBike` наследовать нежелательное или ненужное ему поведение.

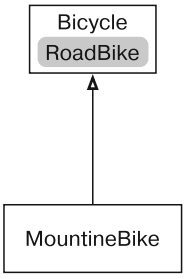

Рис. 6.5. В классе Bicycle объединено общее поведение с поведением, специфичным для дорожных велосипедов

В классе `Bicycle` содержится поведение, которое одинаково подходит обоим велосипедам и родителю класса `MountainBike`. Часть этого поведения в `Bicycle` подходит для `MountainBike`, часть не подходит, а часть и вовсе к нему неприменима. В том виде, в котором он написан, класс `Bicycle` не может служить родительским классом для `MountainBike`.

Поскольку конструкция эволюционирует, подобная ситуация возникает постоянно. Проблема здесь начинается с имен классов.

Поиск абстракции

Сначала была одна задумка — велосипед, и он был смоделирован в виде отдельного класса `Bicycle`. Программист, начавший эту работу, выбрал общее имя для объекта, который, по сути, имел более конкретизированный характер. Существующий класс `Bicycle` не является представителем *любой* разновидности велосипеда, он представляет конкретный вид — дорожный велосипед.

Выбор для класса такого имени полностью соответствовал тому приложению, в котором каждый `Bicycle` — *это* дорожный велосипед. Когда есть только один вид велосипеда, не обязательно выбирать для класса имя `RoadBike` (дорожный велосипед), которое, возможно, несколько специфично. Даже если есть предположение, что когда-либо у вас появятся горные велосипеды, `Bicycle` — вполне подходящее имя для первого класса, соответствующее текущему моменту.

Но теперь, когда появился класс для горного велосипеда `MountainBike`, имя `Bicycle` вводит в заблуждение. Эти два имени класса *подразумевают* наследование; вы ожидаете, что `MountainBike` является конкретизацией `Bicycle`. Вполне естественно написать код, создающий `MountainBike` в виде подкласса `Bicycle`. Структура получится подходящей, с именами классов тоже все в порядке, а вот код в `Bicycle` теперь никуда не годится.

Подклассы являются *конкретизациями* своих родительских классов. `MountainBike` должен быть всем, чем является `Bicycle`, плюс дополнительные особенности. Любой объект, ожидающий `Bicycle`, должен иметь возможность взаимодействовать с `MountainBike`-объектом в счастливом неведении о его фактическом классе.

Это правила наследования — и нарушать их вы можете только на свой страх и риск. Чтобы наследование работало, следует всегда соблюдать два правила. Во-первых, моделируемые объекты должны на самом деле иметь взаимоотно-

шения «обобщение — конкретизация». Во-вторых, нужно использовать правильные технологии программирования.

Вполне резонно смоделировать горный велосипед в виде конкретизации велосипеда, тогда взаимоотношения будут правильными. Но показанный выше код беспорядочен, и его распространение приведет к печальным последствиям. Текущий класс `Bicycle` смешивает код, общий для всех велосипедов, с кодом конкретного дорожного велосипеда. Настало время разделить код этого класса, убрав код для дорожного велосипеда из `Bicycle` в отдельный подкласс `RoadBike`.

Создание абстрактного родительского класса

На рис. 6.6 показана новая диаграмма классов, где `Bicycle` является родительским классом как для `MountainBike`, так и для `RoadBike`. Это именно то, что вам нужно, — структура наследования, которую вы намеревались создать. `Bicycle` будет содержать общее поведение, а `MountainBike` и `RoadBike` добавят конкретики. Открытый интерфейс класса `Bicycle` должен включить `spares` и `size`, а интерфейсы его подклассов добавят свои собственные части.

Теперь `Bicycle` представляет собой *абстрактный* класс. В главе 3 абстрактность определялась как несвязанность с любым конкретным экземпляром, и данное определение остается в силе. Эта новая версия класса `Bicycle` не станет определять целиком весь велосипед, в ней будут только определения чего-то общего для всех велосипедов. Можно ожидать создание экземпляров `MountainBike` и `RoadBike`, но `Bicycle` не является классом, к которому когда-либо будут отправляться новые сообщения. В этом не будет никакого смысла; теперь `Bicycle` больше не представляет велосипед целиком.

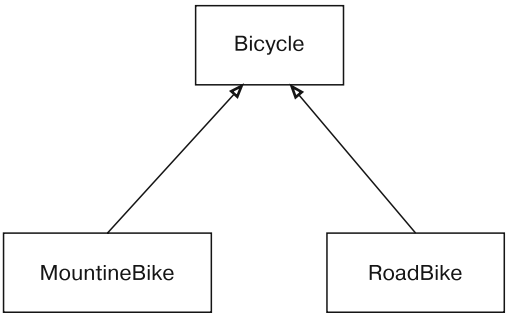

Рис. 6.6. Bicycle в качестве родительского класса MountainBike и RoadBike

В некоторых объектно-ориентированных языках программирования имеется синтаксис, позволяющий объявлять классы абстрактными явным образом. В Java, к примеру, если ключевое слово `abstract`. Компилятор Java препятствует созданию экземпляров классов, к которым было применено это ключевое слово. Ruby не содержит такого ключевого слова и не накладывает таких ограничений. Других программистов от создания экземпляров класса `Bicycle` удерживает только здравый смысл; в реальной жизни это срабатывает на удивление неплохо.

Абстрактные классы существуют для того, чтобы на их основе создавались подклассы. Это их единственное предназначение. Они предоставляют общее хранилище для поведения, совместно используемого набором подклассов (тех самых подклассов, которые предоставляют конкретизацию).

Вряд ли есть смысл создавать абстрактный родительский класс всего лишь с одним подклассом. Даже если в исходном варианте класса `Bicycle` содержится общее и конкретное поведение, позволяющее представить себе моделирование двух классов с самого начала, так делать не нужно. Независимо от того, как скоро вы ожидаете наличие других видов велосипедов, этот день может не наступить никогда. До тех пор пока у вас не появятся конкретные требования, заставляющие иметь дело с другими велосипедами, текущий класс `Bicycle` вас будет вполне устраивать.

Даже притом что теперь у вас имеются требования к работе с двумя видами велосипедов, возможно, это *все еще* неподходящий момент для осуществления наследования. Создание иерархии влечет затраты; наилучший способ сведения этих затрат к минимуму — максимальное увеличение ваших шансов на получение абстракции непосредственно перед тем, как позволить подклассам приобрести зависимость от нее. Хотя два велосипеда, о которых вам известно, предоставляют изрядное количество информации об общей абстракции, от трех велосипедов ее может поступить еще больше. Если можно было бы отложить это решение до того, как компании FastFeet потребуется третья разновидность велосипеда, ваши шансы на то, чтобы найти верную абстракцию, существенно бы возросли.

Решение отложить создание иерархии `Bicycle` обязывает написать классы `MountainBike` и `RoadBike`, в которых дублируется существенный объем кода. Решение о том, чтобы приступить к созданию иерархии, влечет риск нехватки информации для выявления правильной абстракции. Ваш выбор (отложить или осуществить сейчас) зависит от того, насколько скоро ожидается третья раз-

новидность велосипеда и насколько высокой будет цена дублирования. Если появление третьей разновидности велосипеда наверняка произойдет, то, возможно, лучше продублировать код и дождаться более полной информации. Но если продублированный код будет нуждаться в ежедневных изменениях, дешевле создать иерархию. По возможности займите выжидательную позицию, но не следует бояться продвижения вперед на основе двух конкретных случаев, если вы считаете это более рациональным решением.

В данном случае предположим, что у вас есть веские причины для создания иерархии `Bicycle`, даже притом что вам известно только о двух разновидностях велосипедов. Первым шагом станет определение структуры, являющейся зеркальным отражением структуры, показанной на рис. 6.6. Если временно не обращать внимания на правильность кода, то проще всего внести нужные изменения путем переименования `Bicycle` в `RoadBike` и создания нового пустого класса `Bicycle`. Именно это и сделано в следующем примере.

```
class Bicycle
   # Теперь этот класс пуст.
   # Весь прежний код был перемещен в RoadBike.
end

class RoadBike < Bicycle
   # Теперь это подкласс класса Bicycle.
   # Он содержит весь код из старого класса Bicycle.
end

class MountainBike < Bicycle
  # Это по-прежнему подкласс класса Bicycle (который теперь пуст).
  # Код не изменился.
end
```

Новый класс `RoadBike` определен в качестве подкласса `Bicycle`. Существующий класс `MountainBike` уже является подклассом `Bicycle`. Его код не изменился, но его поведение, конечно же, изменилось, поскольку его родительский класс теперь пуст. Код, от которого зависит `MountainBike`, был удален из его родительского класса и помещен в класс, равный по уровню.

Такое перестроение кода просто переместило проблему, что и показано на рис. 6.7. Теперь вместо излишних линий поведения в `Bicycle` их нет вообще. Общее поведение, необходимое всем велосипедам, ушло вниз в `RoadBike` и поэтому стало недоступно для `MountainBike`.

Данное перестроение существенно улучшило ситуацию, поскольку проще поднять код вверх, в родительский класс, чем опустить вниз, в подкласс. Причина этого пока не столь очевидна, но по мере рассмотрения примера она прояснится.

Следующие несколько шагов направлены на создание новой структуры классов путем перемещения общего поведения в `Bicycle` и эффективного использования этого поведения в подклассах.

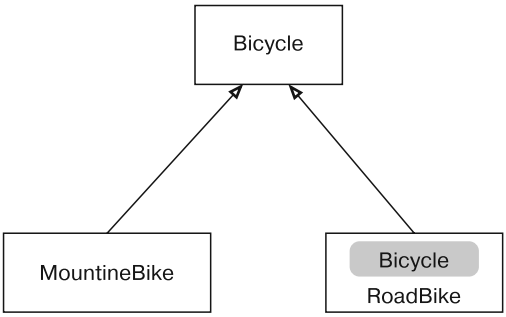

Рис. 6.7. Теперь все общее поведение содержится в RoadBike

В `RoadBike` по-прежнему содержится все, что ему нужно, поэтому он сохраняет работоспособность, а вот `MountainBike` пришел в полную негодность. В качестве примера посмотрим, что произойдет, если создать экземпляры каждого подкласса и запросить у них размер (`size`). `RoadBike` даст правильный ответ, а `MountainBike` даст сбой.

```
road_bike = RoadBike.new(
              size: 'M',
              tape_color: 'red' )

road_bike.size # => "M"

mountain_bike = MountainBike.new(
              size: 'S',
              front_shock: 'Manitou',
              rear_shock: 'Fox')

mountain_bike.size
# NoMethodError: метод 'size' не определен
```

Причина сбоя очевидна: `size` не реализован ни в `MountainBike`, ни в его родительском классе.

Перемещение вверх абстрактного поведения

Методы `size` и `spares` являются общими для всех велосипедов. Это поведение принадлежит открытому интерфейсу `Bicycle`. Оба метода в данный момент спущены вниз в `RoadBike`, и теперь стоит задача переместить их вверх, в `Bicycle`, чтобы они получили возможность совместного использования. Код, работающий с размером, самый простой, поэтому вполне естественно начать именно с него.

Перемещение вверх (в родительский класс) поведения, связанного с размером, требует трех изменений, показанных в следующем примере. Из `RoadBike` в `Bicycle` переместился код, читающий атрибут и проводящий инициализацию (строки 2 и 5), а в метод `initialize` класса `RoadBike` добавляется отправка `super` (строка 14).

```
 1 class Bicycle
 2   attr_reader :size # <- перемещено из RoadBike
 3
 4   def initialize(args={})
 5      @size = args[:size] # <- перемещено из RoadBike
 6   end
 7 end
 8
 9 class RoadBike < Bicycle
10   attr_reader :tape_color
11
12   def initialize(args)
13      @tape_color = args[:tape_color]
14      super(args) # <- RoadBike теперь ДОЛЖЕН отправлять 'super'
15   end
16   # ...
17 end
```

Теперь `RoadBike` наследует `size` из `Bicycle`. Когда `RoadBike` получает сообщение `size`, Ruby самостоятельно делегирует это сообщение вверх по цепочке родительских классов, проводя поиск реализации и находя ее в `Bicycle`. Это делегирование сообщения происходит автоматически, поскольку `RoadBike` является подклассом `Bicycle`.

А совместное использование кода инициализации, устанавливающего значение переменной `@size`, требует от вас более серьезных усилий. Значение этой переменной устанавливается в методе `initialize` класса `Bicycle`, то есть в методе, также реализуемом или переопределяемом в `RoadBike`.

Когда `RoadBike` переопределяет `initialize`, он предоставляет получателя этого сообщения, что вполне устраивает Ruby и препятствует автоматическому делегированию сообщения в `Bicycle`. Если нужно, чтобы запускались оба метода `initialize`, делегирование возлагается на сам класс `RoadBike`. Чтобы сделать пересылку сообщения в адрес `Bicycle` явным, он должен отправить сообщение `super`, что, собственно, и показано в строке 14.

До внесения этих изменений `RoadBike` правильно откликался на `size`, а `MountainBike` давал сбой. Теперь общее для них поведение определено в `Bicycle` — их общем родительском классе. Магия наследования заключается в том, что теперь, как показано ниже, правильно отвечают на сообщение `size` оба класса.

```
road_bike = RoadBike.new(
              size: 'M',
              tape_color: 'red' )

road_bike.size # -> ""M""

mountain_bike = MountainBike.new(
              size: 'S',
              front_shock: 'Manitou',
              rear_shock: 'Fox')

mountain_bike.size # -> 'S'
```

Бдительный читатель заметит, что код, обрабатывающий размер велосипеда, был перемещен дважды. Он присутствовал в исходной версии класса `Bicycle`, был перемещен *вниз*, в `RoadBike`, а теперь опять возвращен *наверх*, в `Bicycle`. Код не изменился, он просто дважды перемещен.

Может возникнуть соблазн пропустить промежуточные действия и просто начать с того, чтобы оставить этот фрагмент кода в `Bicycle`, но примененная стратегия «сначала все вниз, а затем что-то вверх» является важной частью реорганизации. Многие трудности наследования вызваны неспособностью четкого разграничения конкретного и абстрактного. В исходном коде класса `Bicycle` было перемешано и то и другое. Если начинать реорганизацию с первой версии `Bicycle`, то любая небрежность с вашей стороны при попытке изолировать код конкретики и спустить его *вниз*, в `RoadBike`, приведет к наличию в родительском классе опасных остатков конкретики. А если начать с перемещения всего, что было в `Bicycle`, в класс `RoadBike`, то затем можно будет четко иденти-

фицировать и переместить наверх абстрактную часть, не опасаясь оставить вверху какую-либо конкретику.

При выборе стратегии реорганизации и общей стратегии проектирования полезно ответить на вопрос: «Что случится, если я ошибусь?» При создании пустого родительского класса и перемещении абстрактных фрагментов кода выше в этот класс худшее, что может произойти, — это неудачный поиск и перемещение наверх всего, что является абстрактным.

Такие недочеты в «продвижении наверх» создают проблему, которую, однако, легко выявить и устранить. Когда наверх попадает не вся общая часть кода, недочеты видны сразу же, как только точно такое же поведение, которое не было перемещено наверх, понадобится другому подклассу. Чтобы предоставить доступ к поведению всем подклассам, вы будете вынуждены либо продублировать код (в каждом подклассе), либо переместить его наверх (в общий родительский класс). Поскольку даже начинающих программистов учат не дублировать код, проблема будет замечена независимо от того, кто будет работать с приложением в будущем. Согласно естественному ходу событий все абстрактное обнаруживается и переносится наверх, а код улучшается. Таким образом, недочеты в продвижении кода наверх не вызывают серьезных последствий.

Однако если попытаться провести реорганизацию в обратном порядке, превращая существующий класс из конкретного в абстрактный путем опускания его конкретной части *вниз*, в новый подкласс, можно случайно оставить в нем часть конкретного поведения. По определению этот остаток конкретного поведения не распространяется на любой возможный новый подкласс. Следовательно, подклассы начинают нарушать основное правило наследования: они не являются настоящими конкретизациями своих родительских классов. Иерархия становится не заслуживающей доверия.

Такие ненадежные иерархии вынуждают взаимодействующие с ними объекты быть в курсе их особенностей. Неопытные программисты не способны разобраться в дефектной иерархии и внести в нее исправления; когда их просят воспользоваться такой иерархией, они вносят сведения об этих особенностях в свой код зачастую путем явной проверки классов объектов. Необходимость знать структуру иерархии просачивается во все остальное приложение, создавая зависимости, повышающие затраты на внесение изменений. Решение проблемы нельзя оставлять на потом. Последствия недочетов при опускании конкретики вниз могут иметь широкое распространение и приводить к возникновению целого ряда серьезных проблем.

Основное правило проведения реорганизации в иерархии наследования заключается в перестановке кода таким образом, чтобы была возможность поднять вверх абстрактную часть кода, а не опустить вниз его конкретную составляющую.

В свете этих фактов намного полезнее сделать акцент на вопросе, который был задан ранее: «Что случится, *когда* я ошибусь?» Каждое принятое вами решение предполагает два вида затрат: одни — на его реализацию, другие — на его изменение, когда обнаружится, что вы ошиблись. Когда при выборе между альтернативными вариантами берутся в расчет оба вида затрат, появляется мотивация выбрать консервативный вариант (в этом случае затраты на изменение сводятся к минимуму).

Теперь переключимся на запчасти (spares).

Отделение абстрактного от конкретного

Версия `spares` реализована как в `RoadBike`, так и в `MountainBike`. Показанное ниже повторение определения этого метода в классе `RoadBike` представляет собой исходный вариант, скопированный из ранее бывшего конкретным класса `Bicycle`. Он обладает автономностью, поэтому сохранил работоспособность.

```
class RoadBike < Bicycle
  # ...
  def spares
    { chain:      '10-speed',
      tire_size:  '23',
      tape_color: tape_color}
  end
end
```

Определение `spares` в классе `MountainBike` (также повторно показанное ниже) осталось от первой попытки выделения подкласса. Этот метод отправляет сообщение `super` в ожидании того, что в родительском классе также реализован метод `spares`.

```
class MountainBike < Bicycle
  # ...
  def spares
```

```
    super.merge({rear_shock: rear_shock})
  end
end
```

Но в `Bicycle` пока не реализован метод `spares`, поэтому отправка сообщения `spares` в адрес `MountainBike` приводит к выдаче следующего исключения `NoMethodError`:

```
mountain_bike.spares
# NoMethodError: super: в родительском классе отсутствует метод 'spares'
```

Очевидно, для устранения этой проблемы требуется добавить к классу `Bicycle` метод `spares`, но сделать это не так-то просто, для этого недостаточно поднять существующий код из `RoadBike`.

Реализация `spares`, имеющаяся в `RoadBike`, обладает излишними сведениями. Такие атрибуты, как `chain` (цепь) и `tire_size` (размер шин), являются общими для всех велосипедов, а вот `tape_color` (цвет ленты) должен быть известен только дорожным велосипедам. Жестко заданные значения `chain` и `tire_size` не станут подходящими исходными данными для каждого возможного подкласса. У этого метода есть масса проблем, и его нельзя поднять вверх в существующем виде. В нем многое смешалось. Когда это было спрятано внутри отдельно взятого класса, с этим еще можно было смириться или даже (в зависимости от вашего терпения) проигнорировать, но теперь, когда вы намерены сделать общей только часть данного поведения, клубок нужно распутать и отделить абстрактные части от конкретных. Абстрактные части должны быть подняты наверх, в `Bicycle`, а конкретные — остаться в `RoadBike`.

Пока не думайте насчет общего метода `spares` и сконцентрируйтесь на подъеме вверх только частей, общих для всех велосипедов — `chain` и `tire_size`. Они являются такими же атрибутами, как `size`, и должны быть представлены не жестко заданными значениями, а выступать средством доступа к значениям и установки этих значений. Требования приобретают следующий вид:

- у велосипедов имеется цепь и размер шин;
- исходные данные относительно цепи одинаковы для всех велосипедов;
- для размера шин подклассы предоставляют свои собственные исходные данные;
- конкретным экземплярам подклассов разрешается игнорировать исходные значения и предоставлять значения, специфичные для экземпляра.

Код для одних и тех же элементов должен следовать одному и тому же шаблону. Новый код работает с `size`, `chain` и `tire_size` аналогичным образом.

```
1  class Bicycle
2     attr_reader :size, :chain, :tire_size
3
4     def initialize(args={})
5        @size      = args[:size]
6        @chain     = args[:chain]
7        @tire_size = args[:tire_size]
8     end
9     # ...
10 end
```

`RoadBike` и `MountainBike` наследуют определения `attr_reader` в `Bicycle`, оба этих класса отправляют в своих методах `initialize` сообщение `super`. Теперь сообщения `size`, `chain` и `tire_size` понятны всем подклассам, и каждый из них может предоставить для этих атрибутов значения, специфичные для конкретного подкласса. Таким образом, первое и последнее из перечисленных выше требований выполнены.

Несмотря на увеличение объема кода, в нем нет ничего необычного. Здравый смысл подсказывает, что он должен быть написан так с самого начала, и этой версии пора уже появиться. Конечно же, она может наследоваться подклассами, но ничего в коде не подсказывает, что от нее *ожидается* возможность наследования. Однако выполнение двух требований относительно исходных значений добавляет кое-что весьма интересное.

Использование схемы шаблонного метода

Следующее изменение касается метода `initialize` класса `Bicycle`, позволяющего отправлять сообщения для получения исходных значений. В строках 6 и 7 показанного ниже кода имеются два новых сообщения — `default_chain` и `default_tire_size`.

Хотя заключение исходных значений в методы — устоявшаяся практика, отправка этих двух новых сообщений поможет достичь двух целей. Основная цель класса `Bicycle` при отправке этих сообщений заключается в предоставлении подклассам возможности внести собственный вклад в конкретизацию путем их перезаписи. Технология определения основной структуры в родительском

классе и отправки сообщений для приобретения вкладов, специфичных для того или иного подкласса, известна как *схема шаблонного метода*.

В следующем примере кода классы `MountainBike` и `RoadBike` пользуются только одной из возможностей специализации. В обоих классах реализуется исходное значение для размера шин `default_tire_size`, но ни в одном из них не реализуется исходное значение для цепи `default_chain`. Таким образом, каждый подкласс предоставляет свое собственное значение для размера шин, но наследует общее исходное значение для цепи.

```
class Bicycle
  attr_reader :size, :chain, :tire_size

  def initialize(args={})
     @size      = args[:size]
     @chain     = args[:chain] || default_chain
     @tire_size = args[:tire_size] || default_tire_size
  end

  def default_chain # <- общее исходное значение
    '10-speed'
  end
end

class RoadBike < Bicycle
  # ...
  def default_tire_size # <- исходное значение, принадлежащее подклассу
    '23'
  end
end

class MountainBike < Bicycle
  # ...
  def default_tire_size # <- исходное значение, принадлежащее подклассу
    '2.1'
  end
end
```

Теперь класс `Bicycle` предоставляет общий алгоритм для своих подклассов. Там, где он разрешает им влиять на алгоритм, он отправляет сообщения. Подклассы делают свой вклад в алгоритм путем реализации совпадающих по имени методов.

Теперь все велосипеды используют одно и то же исходное значение для цепи, но разные исходные значения для размера шин.

```
road_bike = RoadBike.new(
             size: 'M',
             tape_color: 'red' )

road_bike.tire_size # => '23'
road_bike.chain     # => "10-speed"

mountain_bike = MountainBike.new(
                 size: 'S',
                 front_shock: 'Manitou',
                 rear_shock: 'Fox')

mountain_bike.tire_size # => '2.1'
road_bike.chain         # => "10-speed"
```

Однако успех пока праздновать рано, поскольку в коде еще остается одна ловушка, поджидающая тех, кто ведет себя неосмотрительно.

Реализация каждого шаблонного метода

Метод `initialize` класса `Bicycle` отправляет `default_tire_size`, но в самом `Bicycle` этот метод не реализуется. Это упущение может вызвать лавину проблем. Представьте, что компания FastFeed добавит еще один тип велосипедов — рикамбент (recumbent). Это низкие длинные велосипеды, позволяющие велосипедисту ехать лежа на спине. Они весьма быстроходные и снижают нагрузку на спину и шею.

Что произойдет, если какой-нибудь программист по наивности создаст новый подкласс `RecumbentBike`, но проигнорирует предоставление в нем реализации метода `default_tire_size`? Он столкнется со следующей ошибкой.

```
class RecumbentBike < Bicycle
   def default_chain
    '9-speed'
   end
end

bent = RecumbentBike.new
# NameError: неопределенная локальная переменная или метод
# 'default_tire_size'
```

Тот, кто занимался проектированием с самого начала, вряд ли столкнется с подобной проблемой. Он *создавал* `Bicycle`, понимая все требования о соответствии подклассов его конструкции. Созданный код работает. Эти ошибки проявятся впоследствии, когда приложение изменится, подстраиваясь под новые требования, и с ошибками столкнутся другие программисты, меньше разбирающиеся в происходящем.

Суть проблемы в том, что `Bicycle` накладывает на свои подклассы требования, которые неочевидны при взгляде на код. Судя по тому, как написан класс `Bicycle`, подклассы *должны* реализовывать метод `default_tire_size`. Подклассы вроде `RecumbentBike` могут дать сбой из-за невыполнения тех требовании, о которых они ничего не знают.

Потенциальных пострадавших можно заранее успокоить, если выполнить одно простое правило. Любой класс, использующий схему шаблонного метода, должен предоставлять реализацию для каждого отправляемого им сообщения, даже если единственной разумной реализацией в классе, отправляющем сообщение, будет следующая.

```
1  class Bicycle
2    #...
3    def default_tire_size
4      raise NotImplementedError
5    end
6  end
```

При явном утверждении, что от подклассов требуется реализация сообщения, предоставляются полезная документация для тех, кто может ею воспользоваться, и полезные сообщения об ошибках для тех, кто этого сделать не может.

Поскольку `Bicycle` предоставляет эту реализацию `default_tire_size`, создание нового экземпляра класса `RecumbentBike` даст сбой со следующим сообщением об ошибке.

```
1 bent = RecumbentBike.new
2 # NotImplementedError: NotImplementedError
```

Хотя вполне допустимо просто выдать эту ошибку и полагаться на отслеживание стека для обнаружения ее источника, можно также явно, как показано ниже в строке 5, предоставить дополнительную информацию.

```
1  class Bicycle
2    #...
3    def default_tire_size
4      raise NotImplementedError,
```

```
5             "Этот класс #{self.class} не может ответить на:"
6     end
7   end
```

Эта дополнительная информация четко обозначает суть проблемы. Запуск кода показывает, что классу `RecumbentBike` нужен доступ к реализации `default_tire_size`.

```
1 bent = RecumbentBike.new
2 #  NotImplementedError:
3 #    Этот класс RecumbentBike не может ответить на:
4 #               'default_tire_size'
```

Сколько бы времени ни прошло с написания класса `RecumbentBike` — две минуты или два месяца, — суть ошибки будет понятна и ее исправление не составит труда.

Создание кода, который выдает толковые сообщения об ошибках, требует минимальных усилий, зато предоставляет ценную информацию. Отдельное сообщение об ошибках вроде бы мелочь, но из таких мелочей складывается целое, к тому же подобное внимание к мелочам говорит о том, что вы серьезный программист. Нужно всегда документировать требования, предъявляемые шаблонными методами, реализуя совпадающий по имени метод, выдающий ошибки с полезными описаниями.

Управление связанностью родительских классов и подклассов

Теперь в классе `Bicycle` содержится основная часть абстрактного поведения велосипедов. В нем имеется код для управления общим размером велосипеда, цепью и размером шин; его структура предлагает подклассам применить для этих атрибутов общие исходные значения. До завершения родительского класса ему не хватает только реализации `spares` (запчасти).

Написать реализацию `spares` в родительском классе можно несколькими способами — они отличаются тем, насколько тесно связывают вместе подклассы и родительские классы. Управление связанностью играет важную роль, поскольку тесно связанные классы прикрепляются друг к другу и их независимое изменение часто невозможно.

В этом разделе показаны две разные реализации запчастей — простая и более сложная (зато более надежная).

Общие сведения о связанности

Первая реализация `spares` — самая простая в написании, но создающая в результате наиболее тесно связанные классы.

Вспомним, что текущая реализация класса `RoadBike` выглядит следующим образом.

```
class RoadBike < Bicycle
  # ...
  def spares
    { chain:      '10-speed',
      tire_size:  '23',
      tape_color: tape_color}
  end
end
```

В этом методе все перемешано, и последняя попытка его продвижения вверх вынудила идти обходным путем, чтобы очистить код. При этом жестко заданные значения для цепи и шин были извлечены в переменные и сообщения, и эти части были перемещены вверх, в класс `Bicycle`. Теперь методы, работающие с цепью и размером шин, доступны в родительском классе.

Текущая реализация `spares` в классе `MountainBike` имеет следующий вид.

```
class MountainBike < Bicycle
  # ...
  def spares
   super.merge({rear_shock: rear_shock})
  end
end
```

В `MountainBike` метод `spares` отправляет сообщение `super` в ожидании, что в одном из его родительских классов реализован метод `spares`. Класс `MountainBike` объединяет хеш своих собственных запчастей с результатами, возвращенными `super`, справедливо ожидая, что они также будут в виде хеша.

При условии, что теперь `Bicycle` может отправлять сообщения для получения данных о цепи и размере шин и что его реализация `spares` должна вернуть хеш, добавление метода `spares` отвечает потребностям подкласса `MountainBike`.

```
class Bicycle
  #...
  def spares
   { tire_size: tire_size,
```

```
    chain: chain}
  end
end
```

Как только этот метод будет помещен в `Bicycle`, подкласс `MountainBike` станет работать в полном режиме. Чтобы поставить в один ряд с ним подкласс `RoadBike`, нужно изменить его реализацию метода `spares` на зеркальное отражение реализации этого метода в подклассе `MountainBike`, то есть заменить код для цепи и размера шин отправкой сообщения `super` и добавить к возвращаемому хешу конкретику дорожного велосипеда.

С учетом того, что последние изменения в `MountainBike` уже были внесены, в следующем листинге показан весь написанный до сих пор код, которым завершается первая реализация этой иерархии.

Заметьте, что код следует четкой схеме. Каждый шаблонный метод, отправляемый классом `Bicycle`, реализован в самом классе `Bicycle`, и оба подкласса `MountainBike` и `RoadBike` отправляют в своих методах `initialize` и `spares` сообщения `super`.

```
class Bicycle
  attr_reader :size, :chain, :tire_size

  def initialize(args={})
     @size      = args[:size]
     @chain     = args[:chain] || default_chain
     @tire_size = args[:tire_size] || default_tire_size
  end

  def spares
   { tire_size: tire_size,
     chain: chain}
  end

  def default_chain
   '10-speed'
  end

  def default_tire_size
   raise NotImplementedError
  end
end

class RoadBike < Bicycle
```

```
25   attr_reader :tape_color
26
27   def initialize(args)
28     @tape_color = args[:tape_color]
29     super(args)
30   end
31
32   def spares
33     super.merge({ tape_color: tape_color})
34   end
35
36   def default_tire_size
37     '23'
38   end
39 end
40
41 class MountainBike < Bicycle
42   attr_reader :front_shock, :rear_shock
43
44   def initialize(args)
45     @front_shock = args[:front_shock]
46     @rear_shock  = args[:rear_shock]
47     super(args)
48   end
49
50   def spares
51     super.merge({rear_shock: rear_shock})
52   end
53
54   def default_tire_size
55     '2.1'
56   end
57 end
```

Такая иерархия классов работает, и у вас может появиться желание на этом и остановиться. Но только то, что она работает, еще не дает гарантию, что она достаточно хороша. В ней по-прежнему есть ловушка, от которой лучше избавиться.

Заметьте, что подклассы `MountainBike` и `RoadBike` следуют одной и той же схеме. Каждый из них знает о своих особенностях (о специализации своих запчастей) и об особенностях своего родительского класса (о том, что в нем реализован метод `spares` для возвращения хеша и что он отвечает на сообщение `initialize`).

Как всегда, знание особенностей других классов создает зависимости, а те, в свою очередь, связывают объекты друг с другом. Зависимости в показанном выше коде также являются ловушками — обе они создаются из-за отправки в подклассах сообщения `super`.

В чем тут ловушка? Если кто-нибудь создает новый подкласс и забывает отправить в его методе `initialize` сообщение `super`, он сталкивается со следующей проблемой.

```
class RecumbentBike < Bicycle
  attr_reader :flag

  def initialize(args)
     @flag = args[:flag] # забыл отправить 'super'
  end

   def spares
     super.merge({flag: flag})
  end

  def default_chain
   '9-speed'
  end

  def default_tire_size
   '28'
  end
end

bent = RecumbentBike.new(flag: 'tall and orange')
bent.spares
# -> {:tire_size => nil, <- не инициализируются
#     :chain     => nil,
#     :flag      => "tall and orange"}
```

Когда в ходе инициализации `RecumbentBike` не отправляет `super`, он пропускает общую инициализацию, предоставляемую классом `Bicycle`, и не получает правильных значений для размера цепи или размера шин. Эта ошибка может проявиться далеко от того места, где кроется ее причина, существенно затрудняя отладку.

Не менее сложная проблема возникает, если подкласс `RecumbentBike` забывает отправить сообщение `super` в своем методе `spares`. Тут получается неверный

хеш `spares` (запчасти) — и этот недостаток может не проявиться, пока `Mechanic` не остановится на дороге рядом со сломанным велосипедом, тщетно выискивая в своих запасах запчасти.

Забыть отправить `super` и спровоцировать тем самым ошибки может любой программист, но наиболее вероятными виновниками (и жертвами) могут оказаться те программисты, которые недостаточно хорошо знакомы с кодом, но получат в будущем задание создать новые подклассы родительского класса `Bicycle`.

Схема кода в этой иерархии требует, чтобы подклассы не только знали, что они делают, но также предполагали взаимодействие со своим родительским классом. Конечно, есть смысл в том, что подклассы знают о вносимой ими конкретике (вполне очевидно, что они — единственные классы, которые *могут* знать о ней), но принуждение подкласса к знанию порядка взаимодействия с его абстрактным родительским классом вызывает множество проблем.

Алгоритм проникает вниз, в подклассы, заставляя каждый из них явным образом отправлять `super` соучастнику. Это приводит к дублированию кода в подклассах и требует, чтобы все они отправляли `super` в точности в одних и тех же местах. Кроме того, это повышает риск, что в будущем программисты при написании новых подклассов станут допускать ошибки, потому что будут надеяться на правильную конкретизацию, но запросто забудут отправить сообщение `super`.

Когда подкласс отправляет сообщение `super`, он фактически объявляет о том, что ему известен алгоритм; он *зависит* от этого знания. Если алгоритм изменяется, работа подкласса может нарушиться, даже если не затрагивается его собственная конкретика.

Устранение связанности подклассов с использованием хук-сообщений

Всех этих проблем можно избежать с помощью еще одной (последней) реорганизации. Вместо того чтобы позволить подклассам знать алгоритм и требовать от них отправки сообщения `super`, подклассы могут отправлять *хук*-сообщения, которые существуют исключительно для того, чтобы предложить подклассам место для предоставления информации за счет реализации совпадающих по имени методов. Применение этой стратегии удаляет знание алгоритма из подклассов и возвращает контроль родительскому классу.

В следующем примере эта технология используется для предоставления подклассу способа внесения своего вклада в инициализацию. Метод `initialize`

класса `Bicycle` теперь отправляет сообщение `post_initialize` и, как всегда, реализует совпадающий по имени метод, который в данном случае ничего не делает.

Подкласс `RoadBike` предоставляет свою собственную конкретизированную инициализацию путем переопределения метода `post_initialize`.

```
class Bicycle

 def initialize(args={})
     @size      = args[:size]
     @chain     = args[:chain] || default_chain
     @tire_size = args[:tire_size] || default_tire_size

     post_initialize(args) # Bicycle отправляет обоим подклассам
 end

 def post_initialize(args) # и реализует вот это
   nil
 end
 # ...
end

class RoadBike < Bicycle

  def post_initialize(args)         # RoadBike может
    @tape_color = args[:tape_color] # при необходимости
  end                               # переопределить этот метод
  # ...
end
```

Такое изменение не просто удаляет отправку сообщения `super` из метода `initialize` подкласса `RoadBike`, оно вообще удаляет метод `initialize`. Подкласс `RoadBike` больше не управляет инициализацией, вместо этого он предоставляет конкретику более крупному абстрактному алгоритму. Этот алгоритм определен в абстрактном родительском классе `Bicycle`, который, в свою очередь, отвечает за отправку сообщения `post_initialize`.

Подкласс `RoadBike` по-прежнему отвечает за то, *какая именно* инициализация ему нужна, но больше уже не отвечает за то, *когда* происходит его инициализация. Это изменение позволяет подклассу `RoadBike` меньше знать о классе `Bicycle`, сокращая тем самым связанность между ними и делая каждый из них более гибким в условиях неопределенного будущего. Подкласс `RoadBike` не знает, когда будет вызван его метод `post_initialize`, и ему все равно, какой имен-

но объект отправляет сообщение. Класс `Bicycle` (или любой другой объект) может отправить это сообщение в любое время; требования о том, что оно должно быть отправлено в ходе инициализации объекта, не существует.

Делегирование контроля за выбором определенного времени родительскому классу означает, что алгоритм может изменяться, не требуя изменений в подклассах.

Точно такая же технология может использоваться для удаления отправки сообщения `super` из метода `spares`. Вместо того чтобы принуждать подкласс `RoadBike` к знанию, что класс `Bicycle` реализует метод `spares` и что эта реализация в классе `Bicycle` возвращает хеш, можно ослабить связанность путем реализации хука, возвращающего управление классу `Bicycle`.

В следующем примере в класс `Bicycle` вносятся изменения для отправки сообщения `local_spares`. Класс `Bicycle` предоставляет исходную реализацию, возвращающую пустой хеш. Подкласс `RoadBike` использует этот хук и переопределяет его для возвращения своей собственной версии `local_spares` с добавлением запчастей, специфичных для дорожного велосипеда.

```
class Bicycle
  # ...
  def spares
     { tire_size: tire_size,
     chain: chain}.merge(local_spares)
  end

  # хук для переопределения в подклассах
  def local_spares
    {}
  end

end

class RoadBike < Bicycle
  # ...
  def local_spares
    {tape_color: tape_color}
  end

end
```

Новая реализация метода `local_spares` в подклассе `RoadBike` заменяет прежнюю реализацию метода `spares`. Это сохраняет конкретику, предоставляемую подклассом `RoadBike`, но сокращает его связанность с классом `Bicycle`. Подкласс

RoadBike больше не должен знать, что класс Bicycle реализует метод spares, он просто ожидает, что его собственная реализация метода local_spares будет когда-либо вызвана каким-либо объектом.

После внесения аналогичных изменений в подкласс MountainBike заключительный вариант иерархии приобретает следующий вид.

```
class Bicycle
  attr_reader :size, :chain, :tire_size

  def initialize(args={})
     @size      = args[:size]
     @chain     = args[:chain]         || default_chain
     @tire_size = args[:tire_size] || default_tire_size
     post_initialize(args)
  end

  def spares
    { tire_size: tire_size,
    chain:       chain}.merge(local_spares)
  end

  def default_tire_size
    raise NotImplementedError
  end

  # subclasses may override
  def post_initialize(args)
    nil
  end

  def local_spares
   {}
  end

  def default_chain
   '10-speed'
  end

end

class RoadBike < Bicycle
  attr_reader :tape_color
```

```
37
38   def post_initialize(args)
39     @tape_color = args[:tape_color]
40   end
41
42   def local_spares
43     {tape_color: tape_color}
44   end
45
46   def default_tire_size
47     '23'
48   end
49 end
50
51 class MountainBike < Bicycle
52   attr_reader :front_shock, :rear_shock
53
54   def post_initialize(args)
55     @front_shock = args[:front_shock]
56     @rear_shock  = args[:rear_shock]
57   end
58
59   def local_spares
60     {rear_shock: rear_shock}
61   end
62
63   def default_tire_size
64    '2.1'
65   end
66 end
```

Теперь в подклассах `RoadBike` и `MountainBike` намного легче разобраться, поскольку они содержат только конкретику. Сразу видно, чем они занимаются, и понятно, что они являются конкретизацией класса `Bicycle`.

Новым подклассам требуется лишь реализовать шаблонные методы. Этот последний пример показывает, как легко может создать новый подкласс даже программист, незнакомый с приложением. Вот как выглядит подкласс `RecumbentBike`, являющийся новой конкретизацией класса `Bicycle`.

```
1 class RecumbentBike < Bicycle
2   attr_reader :flag
3
4   def post_initialize(args)
```

```
    @flag = args[:flag]
  end

  def local_spares
    {flag: flag}
  end

  def default_chain
    "9-speed"
  end

  def default_tire_size
    '28'
  end
end

bent = RecumbentBike.new(flag: 'tall and orange')
bent.spares
# -> {:tire_size => "28",
#     :chain => "9-speed",
#     :flag => "tall and orange"}
```

Код в `RecumbentBike` вполне понятен, привычен и предсказуем, так что может даже сойти со сборочного конвейера. Он служит хорошей демонстрацией силы и ценности наследования, которое правильно организовано. Успешное создание нового подкласса по плечу любому программисту.

Выводы

Наследование решает проблемы родственных типов, совместно использующих значительную часть общего поведения, но отличающихся по некоторым показателям. Наследование позволяет изолировать совместно используемый код и реализовать общие алгоритмы в абстрактном классе, а также создать структуру, позволяющую подклассам предоставлять конкретику.

Самый лучший способ создания абстрактного родительского класса — перемещение кода вверх из конкретных подклассов. Проще всего идентифицировать правильную абстракцию при наличии доступа минимум к трем существующим конкретным классам. В этой главе пример приводился в отношении всего двух подклассов, но в реальном мире лучше дождаться дополнительной информации, предоставляемой тремя вариантами подклассов.

Чтобы побудить наследников к предоставлению конкретики, в абстрактных родительских классах используются схема шаблонных методов и хук-методы, позволяющие этим наследникам предоставить свою конкретику без принуждения к отправке сообщения `super`. Хук-методы помогают подклассам предоставлять конкретику без знания абстрактного алгоритма. С их помощью устраняется необходимость отправки подклассами сообщения `super`, ослабляется связанность между уровнями иерархии и повышается их терпимость к изменениям.

Тщательно продуманное наследование легко поддается расширению новыми подклассами (справятся даже те программисты, которые очень мало знают о приложении). Такая легкость расширения — главная сила наследования. Когда перед вами стоит задача, требующая наличия для многочисленной конкретики стабильной общей абстракции, наследование может стать недорогим решением.

Глава 7
Разделение ролевого поведения с помощью модулей

Предыдущая глава завершилась таким многообещающим кодом, что можно удивиться, почему за всю вашу жизнь все это вам еще не встречалось. Но перед тем как воспользоваться классическим наследованием для решения всех мыслимых проблем проектирования, задайтесь следующим вопросом: «Что произойдет, когда у компании FastFeet появится потребность использовать лежачие горные велосипеды?»

Если решение этой новой задачи проектирования не складывается в четкую картину, это неудивительно. Создание подкласса для лежачих горных велосипедов требует сочетания качеств двух уже существующих подклассов, а с ходу приспособить наследование под эти обстоятельства невозможно. Еще большее беспокойство вызывает тот факт, что данная неудача иллюстрирует только одно из множества направлений, в которых наследование может не получиться.

Чтобы извлекать выгоду из применения наследования, нужно разобраться не только в том, как пишется соответствующий код, но и в том, когда есть смысл этим наследованием воспользоваться. Без классического наследования всегда можно обойтись (любая проблема, решаемая с его помощью, может быть решена и другим способом). Поскольку ни одна технология проектирования не обходится без расходов, создание самых бюджетных приложений требует идти на

компромиссы между затратами и предполагаемыми выгодами от использования альтернативных вариантов.

В этой главе исследуется альтернативный вариант с применением технологии наследования для разделения *роли*. Исследование начинается с примера, использующего модуль Ruby для определения общей роли, и дается практический совет насчет способов написания полностью наследуемого кода.

Основные сведения о ролях

Некоторые задачи требуют совместного использования поведения несколькими никак не связанными объектами. Это общее поведение не зависит от класса, это *роль*, которую играет объект. Многие из ролей, необходимых приложению, выявятся в ходе проектирования, но часто неожиданные роли обнаруживаются в ходе написания кода.

Когда ранее ничем не связанные объекты начинают играть общую роль, они входят во взаимоотношения с объектами, для которых они играют эту роль. Эти взаимоотношения не настолько видны, как те, которые создаются требованиями по линии родительского класса и подкласса при классическом наследовании. Использование роли создает зависимости между вовлеченными в нее объектами, и эти зависимости привносят риски, которые следует брать в расчет, принимая решение насчет вариантов проектирования.

В этом разделе выявляется скрытая роль и создается код для разделения соответствующего ей поведения среди всех исполнителей (наряду с минимизацией возникающих в результате этого зависимостей).

Поиск ролей

Неявный тип `Preparer` из главы 5 является ролью. Эту роль играют объекты, реализованные в интерфейсе `Preparer`. В классах `Mechanic`, `TripCoordinator` и `Driver` реализован метод `prepare_trip`, поэтому другие объекты могут с ним взаимодействовать, как будто они относятся к типу `Preparer`, не обращая внимания на свои базовые классы.

Существование выполняющего подготовку `Preparer` предполагает параллельно наличие роли подготавливаемого `Preparable` (роли часто появляются парами). В примере, рассматриваемом в главе 5, в качестве `Preparable` фигурирует

класс `Trip`; в нем реализуется интерфейс `Prepareable`. Этот интерфейс включает все сообщения, отправка которых в адрес `Preparable` ожидается от любого `Preparer` (речь идет о методах `bicycles`, `customers` и `vehicle`). Роль `Preparable` слабо выраженная, потому что `Trip` является ее единственным исполнителем, но выявить ее существование очень важно. В главе 9 «Проектирование экономичных тестов» предлагаются технологии для тестирования и документирования роли `Preparable`, чтобы отличить ее от класса `Trip`.

Хотя у роли `Preparer` имеется несколько исполнителей, она настолько проста, что ее полностью можно определить с помощью ее же интерфейса. Чтобы играть эту роль, объекту нужно лишь реализовать свою собственную версию метода `prepare_trip`. Объекты, исполняющие роль `Preparer`, в качестве общего признака имеют только этот интерфейс. Они совместно используют сигнатуру метода и не используют в этом качестве никакого другого кода.

`Preparer` и `Preparable` — самые настоящие неявные типы. Но гораздо чаще приходится выявлять более сложные роли, требующие не только конкретных сигнатур сообщений, но и специфического поведения. Когда роль требует совместно используемого поведения, вы сталкиваетесь с проблемой организации совместно используемого кода. В идеале этот код должен определяться в одном месте, но быть доступным для использования любым объектом, желающим проявлять себя как неявный тип и играть роль.

Способ определения именованной группы методов, независимых от класса и примешиваемых к любому объекту, имеется во многих объектно-ориентированных языках. В Ruby такие примешиваемые методы, или миксины, называются *модулями*. Методы могут определяться в модуле, а затем модуль может добавляться к любому объекту. Таким образом модули предоставляют великолепный способ, позволяющий объектам различных классов играть общую роль, используя единый набор кода.

Когда в объект включается модуль, определенные в нем методы становятся доступными через автоматическое делегирование. Это выглядит как классическое наследование (по крайней мере с точки зрения включающего модуль объекта). Ему поступают сообщения, он их не понимает, и они автоматически направляются куда-то еще (волшебным образом обнаруживается реализация соответствующего им метода, который выполняется и возвращает ответ).

Когда вы начинаете помещать код в модули и добавлять модули к объектам, вы расширяете набор сообщений, на которые этот объект в состоянии ответить,

и вводите новую область сложности конструкции. У объекта, непосредственно реализующего всего несколько методов, может все же иметься очень большой список сообщений, на которые он откликается. Общий набор сообщений, на которые может откликаться объект, включает:

- реализованное в нем самом;
- реализованное во всех объектах выше него в иерархии;
- реализованное в любом модуле, который был к нему добавлен;
- реализованное во всех модулях, добавленных к любому объекту, находящемуся выше него в иерархии.

Если этот набор ответов кажется большим и запутанным, значит, у вас есть четкое представление о проблеме. Понять полную картину поведения в глубоко вложенной иерархии зачастую невозможно.

Организация обязанностей

Теперь, когда у вас сформировалось мрачное представление о возможном развитии событий, настало время взглянуть на вполне управляемый пример. Как и при классическом наследовании, перед тем как получить возможность выбора места создания неявного типа и помещения совместно используемого поведения в модуль, нужно узнать о том, как это правильно делается. К счастью, нам вполне может пригодиться пример классического наследования из главы 6 «Приобретение поведения за счет наследования» (наш пример будет построен на этих технологиях и станет заметно короче).

Рассмотрим задачу планирования путешествия. Путешествия проходят в конкретные моменты времени, и к ним привлекаются механики, используются велосипеды и автотехника (это реальные объекты, которые не могут одновременно находиться сразу в двух местах). Компании FastFeet требуется способ организации этих объектов в виде плана, чтобы в любой момент можно было определить, какие объекты доступны, а какие уже задействованы.

Определение доступности (возможности участвовать в путешествии) не сводится к простому определению, что велосипед, механик или автотехника нигде не задействованы в те сроки, в которые планируется провести путешествие. Эти объекты реального мира нуждаются в перерыве между путешествиями (они не могут сегодня завершить путешествие, а уже завтра

отправиться в новое): велосипеды и автотранспорт должны пройти техническое обслуживание, а механики должны отдохнуть (от работы с клиентами) и постирать униформу.

Согласно требованиям, в эксплуатации велосипедов должен быть перерыв как минимум один день, автотранспорта — три дня, а механику на отдых дается четыре дня.

Код планирования использования этих объектов может быть написан с помощью разных способов, но, как уже повелось в данной книге, наш пример будет развиваться. Он начнется с кода, вызывающего некоторые опасения, и продолжится его доработкой до вполне удовлетворительного состояния (и все это будет делаться в интересах разоблачения возможных антишаблонов).

Предположим, что у нас уже есть класс `Schedule`. Его интерфейс включает в себя три метода.

```
scheduled?(target, starting, ending)
add(target, starting, ending)
remove(target, starting, ending)
```

Каждый из этих методов получает три аргумента: целевой объект, начальную и конечную даты интересующего нас периода. В обязанность класса `Schedule` входит знание о том, запланирован ли уже поступающий аргумент `target`, а также добавление и удаление целевых объектов из плана. Эти обязанности вполне обоснованно возлагаются на сам класс `Schedule`.

С методами все в порядке, но, к сожалению, в этом коде есть пробел. Все было бы хорошо, если бы знание о фигурировании объекта в плане в определенный период времени было всей информацией, необходимой для предотвращения повторного внесения в план уже занятого объекта. Но знание о том, что объект *не* фигурирует в плане в определенный период времени, не является достаточной информацией, чтобы знать, что его *можно* включить в план в течение какого-то периода. Чтобы правильно определить, можно ли объект внести в план, нужно рассчитать время на его подготовку.

На рис. 7.1 показана реализация класса `Schedule`, в которой он сам берет на себя обязанность по знанию реального времени на подготовку. Методу `schedulable?` известны все возможные значения, и он проверяет класс поступающего к нему аргумента `target`, чтобы решить, какое время на подготовку использовать.

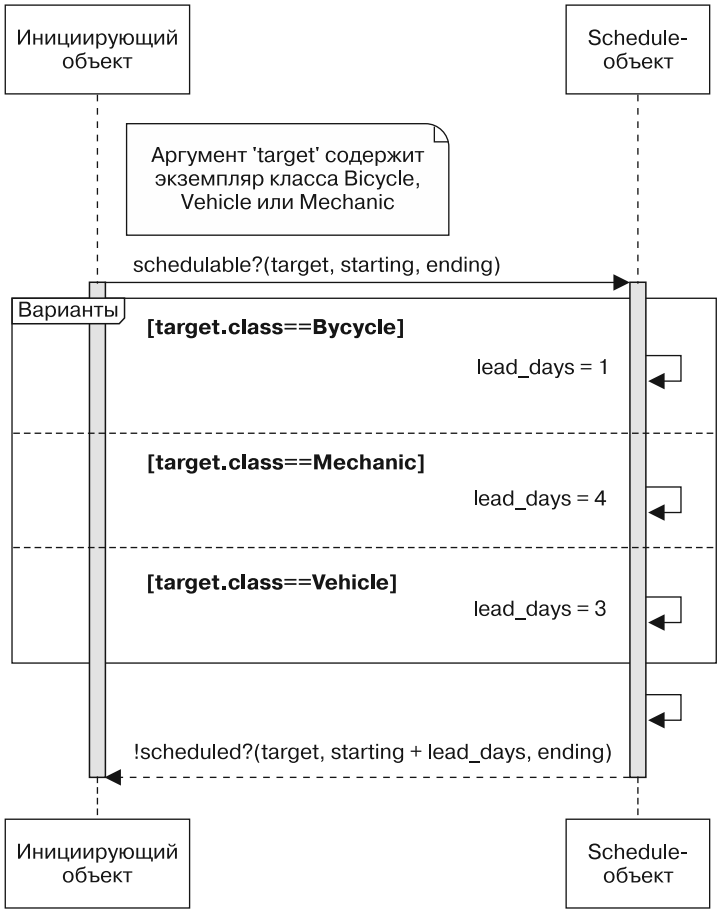

Рис. 7.1. Классу планирования известны сроки подготовки других объектов

Вы уже видели шаблон проверки класса с целью определения, какое именно *сообщение* нужно отправить; здесь `Schedule` проверяет класс, чтобы узнать, какое именно *значение* нужно использовать. В обоих случаях `Schedule` знает слишком много. Это знание не принадлежит `Schedule`, оно принадлежит классам, имена которых проверяются `Schedule`-объектом.

Такая реализация наводит на мысль о простом и очевидном усовершенствовании, которое подсказано самой схемой кода. Вместо знания подробностей, касающихся других классов, `Schedule` должен отправлять им сообщения.

Устранение ненужных зависимостей

Тот факт, что `Schedule` проверяет множество имен классов, чтобы определить, какое именно значение поместить в одну и ту же переменную, подсказывает, что имя переменной должно быть превращено в сообщение, которое должно быть отправлено каждому поступающему объекту.

Выявление неявного типа, подходящего для планирования

На рис. 7.2 показана диаграмма последовательности для нового кода, в которой из метода `schedulable?` удаляется проверка класса, и метод изменяется для отправления сообщения `lead_days` своему аргументу, содержащему поступающий целевой объект. Инструкция `if`, проверяющая класс объекта, заменяется отправкой сообщения этому же объекту. Это изменение упрощает код и перекладывает обязанность знания реального количества дней, необходимых на подготовку, на последний упомянутый объект, которому и принадлежит эта обязанность.

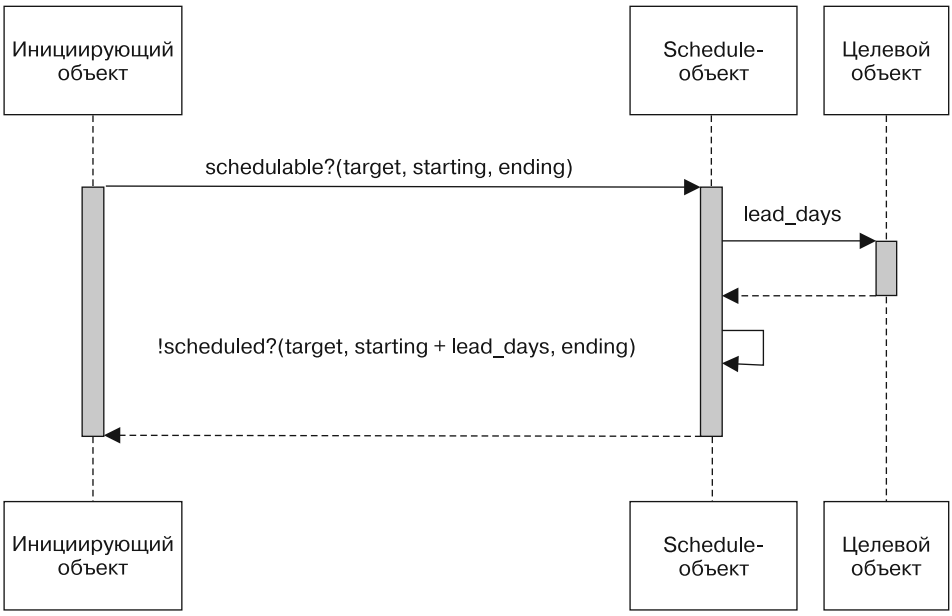

Рис. 7.2. Класс планирования ожидает, что целевые объекты знают, сколько времени им требуется на подготовку

На рис. 7.2 можно заметить кое-что интересное: на диаграмме имеется прямоугольник с надписью «целевой объект». Прямоугольники на диаграмме последовательности предназначены для обозначения объектов и зачастую носят имена классов, например `Schedule`-объект или `Bicycle`-объект. На рис. 7.2 `Schedule`-объект намерен отправить сообщение `lead_days` своему целевому объекту, но тот может быть экземпляром любого из нескольких классов. Поскольку классы целевых объектов неизвестны, не вполне понятно, какую надпись использовать для прямоугольника с получателем этого сообщения.

Нарисовать диаграмму без этой проблемы проще всего, если изобразить прямоугольник с именем переменной и показать отправку сообщения `lead_days` этому целевому объекту, не уточняя его принадлежности к какому-либо классу. `Schedule`-объект совершенно не интересует класс целевого объекта, он просто ждет от него ответ на конкретное сообщение. Эти ожидания на основе сообщения уже не имеют никакого отношения к классу и раскрывают единую роль, исполняемую всеми целевыми объектами, которая теперь стала явно прослеживаться на диаграмме последовательности.

`Schedule`-объект ожидает от своего целевого объекта поведения, при котором ему будет понятно сообщение `lead_days`, то есть поведения, позволяющего отнести его к разряду пригодных для планирования (schedulable). Следовательно, вы выявили неявный тип.

Именно сейчас новый неявный тип приобрел очертания, и это произошло практически так же, как было в главе 5 с неявным типом `Preparer`; вновь выявленный неявный тип состоит всего лишь из интерфейса. Объекты неявного типа `Schedulable` должны иметь реализацию `lead_days` и на данный момент не имеют никакого другого общего кода.

Нужно позволить объектам говорить самим за себя

Выявление и использование этого неявного типа улучшает код за счет удаления зависимости класса `Schedule` от конкретных имен классов, что придает приложению больше гибкости и упрощает его сопровождение. Но на рис. 7.2 все еще имеются ненужные зависимости, которые требуется удалить.

Проще всего показать эти зависимости с помощью примера. Представьте себе класс `StringUtils`, реализующий полезные методы управления строками. Узнать у `StringUtils`-объекта о том, является ли строка пустой, можно, отправив сообщение `StringUtils.empty?(some_string)`.

Если у вас солидный опыт создания объектно-ориентированного кода, то данная затея покажется смешной. Для работы со строками совсем не обязательно использовать отдельный класс, ведь строки сами по себе являются объектами со своим собственным поведением и могут сами справиться с этой работой. Требование о том, чтобы некая третья сторона `StringUtils` забирала поведение у строк, усложняет код из-за добавления совершенно ненужной зависимости.

Этот характерный пример иллюстрирует основной замысел: объекты должны быть самоуправляемыми и содержать свое собственное поведение. Если вас интересует объект B, то не нужно вас принуждать к знанию особенностей объекта A (если единственным, для чего он нужен, является выяснение особенностей объекта B).

На диаграмме последовательности, изображенной на рис. 7.2, это правило нарушается. К сожалению, вопрос задается не самому целевому объекту, а третьей стороне — `Schedule`-объекту. Вопрос к `Schedule` о том, можно ли планировать использование целевого объекта, похож на вопрос к `StringUtils`, является ли строка пустой. Он заставляет инициатора знать об этом и, таким образом, зависеть от `Schedule`, даже если его интересует исключительно целевой объект.

Точно так же, как строки способны откликаться на сообщение `empty?` и говорить сами за себя, целевые объекты должны откликаться на сообщение `schedulable?`. Метод `schedulable?` нужно добавить к интерфейсу роли `Schedulable`.

Написание конкретного кода

По состоянию на данный момент роль `Schedulable` содержит только интерфейс. Добавление к этой роли метода `schedulable?` требует написания кода, но сразу разобраться, где именно должен находиться этот код, не получается. Нужно принять два решения: что этот код должен делать и где он должен находиться.

Проще всего начать с разделения двух решений. Выберите произвольный конкретный класс (например, `Bicycle`) и реализуйте непосредственно в нем метод `schedulable?`. Когда у вас будет версия, работающая для класса `Bicycle`, можно будет реорганизовать свой способ расположения кода, позволяющего воспользоваться его поведением всем `Schedulable`-объектам.

На рис. 7.3 показана диаграмма последовательности, где этот новый код находится в классе `Bicycle`. Теперь этот класс реагирует на сообщения насчет его собственной возможности включения в план (schedulability).

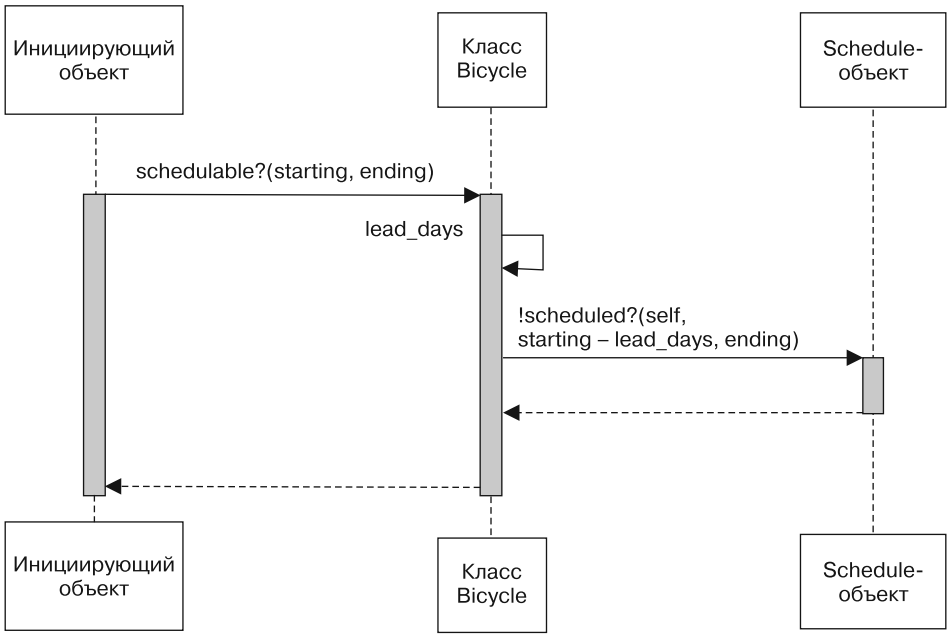

Рис. 7.3. Объекты класса Bicycle знают, могут ли они быть включены в план

До внесения этого изменения каждый инициирующий объект должен был знать об этом и, таким образом, имел зависимость от `Schedule`. Это изменение позволило велосипедам (bicycles) говорить самим за себя (дало возможность инициирующим объектам взаимодействовать с ними без помощи третьей стороны).

Отображенный на диаграмме последовательности код довольно прост. Вот как выглядит очень простая версия класса `Schedule`. Конечно, эта версия недостойна реализации, но зато она предоставляет неплохую замену временно отсутствующей остальной части примера.

```
class Schedule
  def scheduled?(schedulable, start_date, end_date)
     puts "Этот #{schedulable.class} " +
          " в плане\n" +
          " между #{start_date} и #{end_date} отсутствует"
     false
  end
end
```

В следующем примере показана реализация в классе `Bicycle` метода `schedulable?`. Класс `Bicycle` знает свое собственное время на подготовку (оно определено в строке 23, ссылка на это есть в строке 13), а сообщение `scheduled?` перенаправляется самому классу `Schedule`.

```
1  class Bicycle
2     attr_reader :schedule, :size, :chain, :tire_size
3
4     # Внедрение Schedule и предоставление значения по умолчанию
5     def initialize(args={})
6        @schedule = args[:schedule] || Schedule.new
7        # ...
8     end
9
10    # Возвращение true, если данный велосипед доступен
11    # в заданный интервал времени (теперь определенный классом Bicycle)
12    def schedulable?(start_date, end_date)
13      !scheduled?(start_date - lead_days, end_date)
14    end
15
16    # Возвращение ответа от schedule
17    def scheduled?(start_date, end_date)
18      schedule.scheduled?(self, start_date, end_date)
19    end
20
21    # Возвращение количества lead_days (дней на подготовку),
22    # прежде чем велосипед может быть включен в план.
23    def lead_days
24      1
25    end
26
27    # ...
28 end
29
30 require 'date'
31 starting = Date.parse("2015/09/04")
32 ending   = Date.parse("2015/09/10")
33
34 b = Bicycle.new
35 b.schedulable?(starting, ending)
36 # Этот Bicycle
37 # в плане между 2015-09-03 и 2015-09-10 отсутствует
38 # => true
```

При выполнении кода (строки с 30-й по 35-ю) подтверждается, что у `Bicycle`-объекта правильно назначена начальная дата, чтобы учесть дни на подготовку, указанные для велосипедов.

В этом коде скрыто знание о том, что такое `Schedule` и что `Schedule` делает внутри `Bicycle`. Объектам, имеющим отношение к `Bicycle`, больше не нужно знать о существовании `Schedule` или о его поведении.

Извлечение абстракции

Показанный выше код содержит решение первой части текущей проблемы, где было определено, что именно должен делать метод `schedulable?`, но «подлежащими включению в план» являются не только объекты типа `Bicycle`. Эту же роль исполняют `Mechanic` и `Vehicle`, поэтому им требуется такое же поведение. Настала пора реорганизовать код, чтобы его можно было совместно использовать среди объектов различных классов.

В следующем примере показан новый модуль `Schedulable`, содержащий абстракцию, извлеченную из показанного выше класса `Bicycle`. Методы `schedulable?` (строка 8) и `scheduled?` (строка 12) являются практически точными копиями тех, что были прежде реализованы в `Bicycle`.

```
 1 module Schedulable
 2   attr_writer :schedule
 3
 4   def schedule
 5      @schedule ||= ::Schedule.new
 6   end
 7
 8   def schedulable?(start_date, end_date)
 9      !scheduled?(start_date - lead_days, end_date)
10   end
11
12   def scheduled?(start_date, end_date)
13     schedule.scheduled?(self, start_date, end_date)
14   end
15
16   # те, кто будет включать этот модуль, могут это переопределить
17   def lead_days
18     0
19   end
20
21 end
```

По сравнению с кодом, который ранее был в классе `Bicycle`, здесь имеются два изменения. Первое касается добавления метода `schedule` (строка 4). Этот метод возвращает экземпляр всего, что есть в `Schedule`.

На рис. 7.2 инициирующий объект зависит от `Schedule`, что означает возможность наличия в приложении множества мест, нуждающихся в знании о `Schedule`. В следующем, улучшенном варианте, показанном на рис. 7.3, эта зависимость была перенесена в `Bicycle`, что сократило ее распространение в приложении. В приведенном выше коде зависимость от `Schedule` была убрана из `Bicycle` и перемещена в модуль `Schedulable`, что еще больше повысило ее изолированность.

Второе изменение касается метода `lead_days` (строка 17). Прежняя его реализация в `Bicycle` возвращала число, характерное для велосипеда, а теперь его реализация в модуле возвращает более обобщенное значение по умолчанию, равное нулю дней.

Даже при отсутствии приемлемого значения по умолчанию для дней, необходимых на подготовку, модуль `Schedulable` все же должен иметь реализацию метода `lead_days`. Для модулей действуют точно такие же правила, как и для классического наследования. Если модуль отправляет сообщение, он должен предоставить реализацию, даже если эта реализация выдает ошибку, сообщающую, что пользователи модуля должны реализовать этот метод.

Включение этого нового модуля в исходный код класса `Bicycle`, как в показанном ниже примере, добавляет реализованные в модуле методы к набору сообщений, на которые реагирует класс `Bicycle`. Метод `lead_days` является хук-методом, следующим схеме шаблонных методов. Класс `Bicycle` переопределяет этот хук-метод (строка 4), предоставляя конкретику.

При запуске кода обнаруживается, что у `Bicycle` сохраняется то же самое поведение, что и при непосредственной реализации этой роли.

```
1  class Bicycle
2    include Schedulable
3
4    def lead_days
5      1
6    end
7
8    # ...
9  end
10
11 require 'date'
```

```
starting = Date.parse("2015/09/04")
ending   = Date.parse("2015/09/10")

b = Bicycle.new
b.schedulable?(starting, ending)
# Этот Bicycle
# в плане между 2015-09-03 и 2015-09-10 отсутствует
# => true

```

Перемещение методов в модуль `Schedulable`, включение модуля и переопределение `lead_days` позволяют классу `Bicycle` вести себя правильно. Дополнительно создание этого модуля дало возможность другим объектам воспользоваться им, чтобы они сами стали пригодными для включения в план. Они могут играть эту роль без дублирования кода.

Схема сообщений изменилась от отправки `schedulable?` классу `Bicycle` до отправки этого сообщения в адрес модуля `Schedulable`. Теперь вы стали приверженцами неявного типа, и диаграмма последовательности, показанная на рис. 7.3, может быть заменена той, которая показана на рис. 7.4.

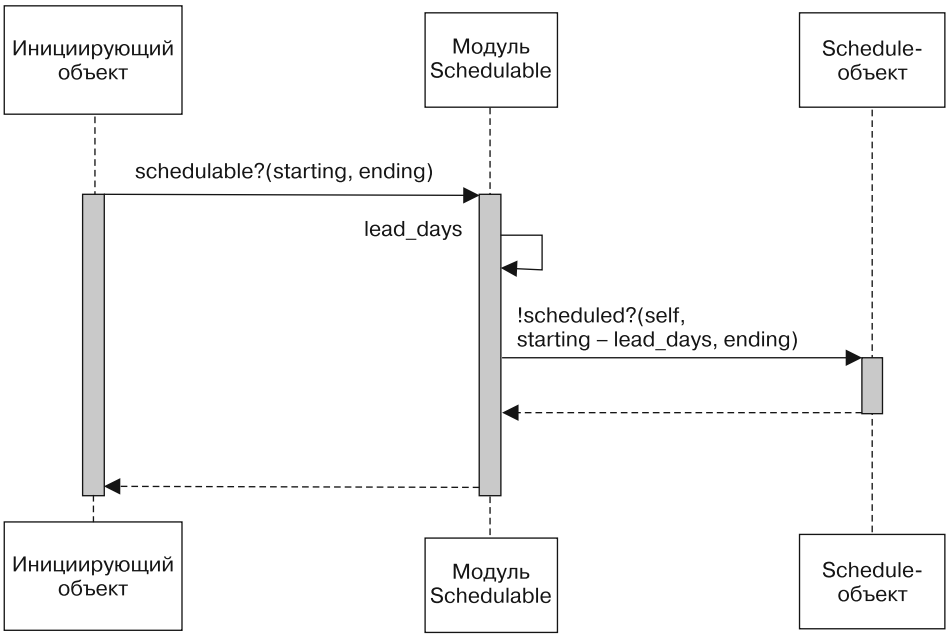

Рис. 7.4. Неявный тип, пригодный для включения в план

Когда этот модуль будет включен во все классы, пригодные для включения в план, структура кода начнет напоминать наследование. В следующем примере показываются классы `Vehicle` и `Mechanic`, включающие модуль `Schedulable` и реагирующие на сообщение `schedulable?`.

```
class Vehicle
   include Schedulable

   def lead_days
     3
   end

   # ...
end

class Mechanic
  include Schedulable

  def lead_days
    4
  end

  # ...
end

v = Vehicle.new
v.schedulable?(starting, ending)
# Этот Vehicle
# в плане между 2015-09-01 и 2015-09-10 отсутствует
# => true

m = Mechanic.new
m.schedulable?(starting, ending)
# Этот Mechanic
# в плане между 2015-02-29 и 2015-09-10 отсутствует
# => true
```

Код в модуле `Schedulable` *является* абстракцией, в нем используется образец шаблонного метода, побуждающий объекты предоставлять конкретику поставляемому им алгоритму. Объекты неявного типа `Schedulable` переопреде-

ляют метод `lead_days`. Когда сообщение `schedulable?` поступает любому `Schedulable`-объекту, оно автоматически перенаправляется методу, определенному в модуле.

Это может не вписываться в строгое определение классического наследования, но относительно того, как должен быть написан код и как происходит разрешение сообщений, это похоже на наследование. Технологии написания программного кода точно такие же, поскольку поиск метода ведется такими же путями.

В этой главе различие между классическим наследованием и совместным использованием кода посредством модулей было проведено весьма осмотрительно. Это различие между *«является»* и *«ведет себя как»*, безусловно, имеет важное значение, у каждого выбранного варианта свои последствия. Тем не менее технологии написания программного кода для обоих вариантов очень похожи, потому что основаны на автоматическом перенаправлении (делегировании) сообщения.

Поиск методов

Понять, чем отличается классическое наследование от включения модулей, будет проще, если разобраться в том, как объектно-ориентированные языки вообще и Ruby в частности ведут поиск реализации метода, соответствующей отправленному сообщению.

Грубое упрощение

Когда объект получает сообщение, объектно-ориентированный язык сначала ведет поиск соответствующей реализации метода в *классе* этого объекта. В этом есть смысл, иначе определения методов должны быть продублированы в каждом экземпляре каждого класса. Хранение методов, известных объекту за пределами его класса, означает, что все экземпляры класса могут совместно использовать один и тот же набор определений методов при условии, что эти определения находятся только в одном месте.

Если вы заметили, в книге чувствуется легкая озабоченность насчет явной констатации, является ли рассматриваемый объект экземпляром класса или самим классом, и ожидание, что его суть прояснится из контекста и вас

вполне устроит концепция, согласно которой сами классы являются полноправными объектами. Описание процедуры поиска метода потребует большей точности.

Как указано выше, поиск метода начинается в классе получающего сообщение объекта. Если этот класс не содержит реализации метода, реагирующего на сообщение, поиск продолжается в его родительском классе. С этого момента все зависит только от родительского класса. Поиск продолжается по цепочке родительских классов с последовательным переходом от одного родительского класса к другому — и так до верхушки иерархии.

На рис. 7.5 показано, как обычный объектно-ориентированный язык будет искать метод `spares` в иерархии `Bicycle`, созданной вами при изучении главы 6. В контексте рассмотрения данного вопроса на верхушке иерархии располагается класс `Object`. Следует учесть, что в Ruby в силу особенностей языка поиск метода будет вестись несколько иначе, но нас пока устроит и эта модель.

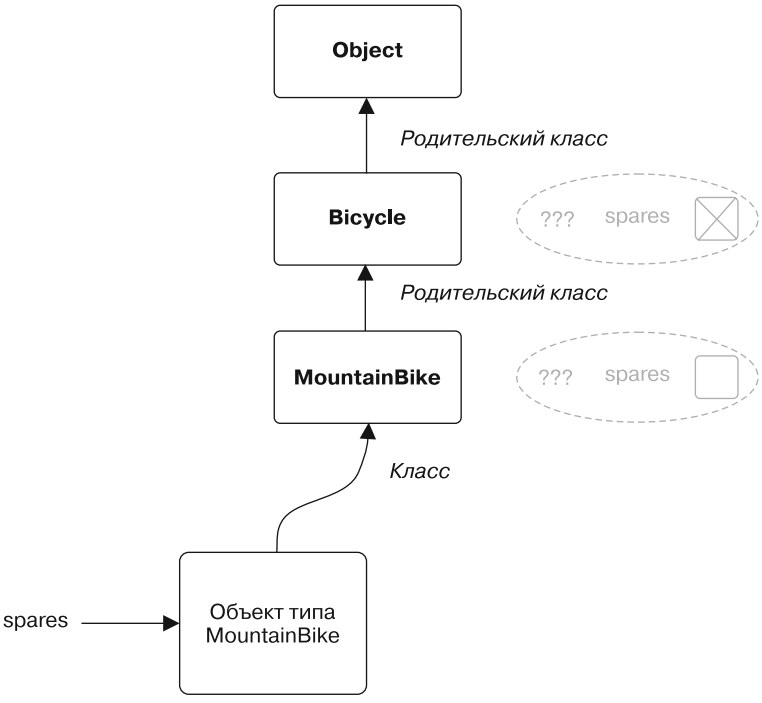

Рис. 7.5. Общее представление о поиске метода

На рис. 7.5 сообщение `spares` отправлено экземпляру класса `MountainBike`. Объектно-ориентированный язык сначала ведет поиск соответствующего метода `spares` в классе `MountainBike`. Если поиск метода `spares` в этом классе не увенчается успехом, он продолжится в родительском для `MountainBike` классе `Bicycle`.

Поскольку в `Bicycle` имеется реализация `spares`, поиск в данном примере на нем и завершится. Но в случае отсутствия реализации в родительском классе поиск перейдет из одного родительского класса в другой, пока не дойдет до верхушки иерархии и не будет проведен в классе `Object`. Если все попытки найти подходящий метод окажутся неудачными, вполне резонно ожидать завершения поиска, но многие языки предпринимают повторную попытку разрешения сообщения.

Ruby дает исходному получателю второй шанс, отправляя новое сообщение `method_missing` и передавая в качестве аргумента значение `:spares`. Попытки разрешения этого нового сообщения еще раз запускают поиск по тому же самому маршруту за исключением того, что теперь ведется поиск для сообщения `method_missing`, а не для сообщения `spares`.

Уточненное объяснение

В предыдущем разделе объяснялся только порядок поиска метода при классическом наследовании. В этом разделе дано расширенное объяснение с привлечением методов, определенных в модуле Ruby. На рис. 7.6 к пути поиска метода добавляется модуль `Schedulable`.

Иерархия объектов, показанная на рис. 7.6, очень похожа на показанную на рис. 7.5. Отличие только в том, что на рис. 7.6 между классами `Bicycle` и `Object` показывается выделенный более ярким тоном модуль `Schedulable`.

Когда `Bicycle` включает `Schedulable`, все методы, определенные в модуле, становятся частью набора сообщений, на которые реагирует `Bicycle`. Имеющиеся в модуле методы включаются в маршрут поиска метода непосредственно над методами, определенными в `Bicycle`. Включение этого модуля не вносит никаких изменений в родительский для `Bicycle` класс (здесь это по-прежнему `Object`), поиск может вестись и в этом классе. Любое сообщение, полученное экземпляром `MountainBike`, теперь имеет шанс на успешное завершение за счет обнаружения соответствующего метода, определенного в модуле `Schedulable`.

Данное обстоятельство имеет весьма серьезные последствия. Если в `Bicycle` реализован метод, определение которого также имеется в `Schedulable`, реализация в `Bicycle` перегружает реализацию, имеющуюся в `Schedulable`. Если `Schedulable` отправляет не реализованные в нем методы, экземпляры `MountainBike` могут столкнуться с непонятными сбоями.

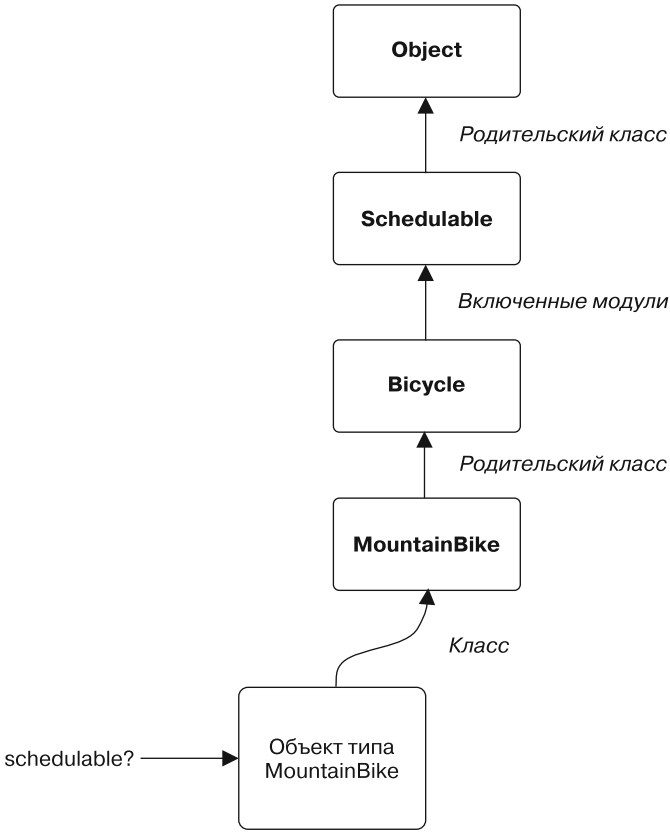

Рис. 7.6. Уточненное объяснение порядка поиска метода

На рис. 7.6 показана отправка сообщения `schedulable?` экземпляру `MountainBike`. Для разрешения этого сообщения Ruby сначала ищет метод в классе `MountainBike`. Затем поиск продолжается по известному маршруту, который теперь включает не только родительские классы, но и модули. Реализация `schedulable?` в конечном итоге обнаруживается в модуле `Schedulable`, который находится на маршруте между `Bicycle` и `Object`.

Почти полное объяснение

После выяснения порядка встраивания модулей в маршрут поиска методов настало время усложнить общую картину.

Вполне возможно, что иерархия будет содержать длинную цепочку родительских классов, в каждый из которых включено множество модулей. Когда один класс включает несколько различных модулей, эти модули включаются в маршрут поиска методов в порядке, *обратном* их включению в класс. Таким образом, последние включенные в класс методы попадаются на маршруте поиска первыми.

До сих пор речь шла о включении модулей в *классы* с помощью имеющегося в Ruby ключевого слова `include`. Как вы уже поняли, включение модуля в класс добавляет имеющиеся в этом модуле методы в набор сообщений, на которые могут откликаться все экземпляры этого класса. Например, на рис. 7.6 модуль `Schedulable` был включен в класс `Bicycle` и в результате экземпляры `MountainBike` получили доступ к методам, определенным в этом модуле.

Но методы, имеющиеся в модуле, можно также добавлять к отдельно взятому объекту, используя для этого имеющееся в Ruby ключевое слово `extend`. Поскольку при использовании `extend` имеющееся в модуле поведение добавляется непосредственно в объект, расширение модулем класса создает методы класса *в этом классе*, а расширение модулем экземпляра класса создает методы экземпляра *в этом экземпляре*. Эти два обстоятельства совершенно одного порядка, ведь не стоит забывать, что классы являются обыкновенными старыми объектами и `extend` ведет себя одинаково для всех.

И наконец, любой объект может также иметь специализированные методы, добавленные непосредственно к его персональному «одноэлементному классу». Эти специализированные методы уникальны для данного конкретного объекта.

Каждый из этих вариантов добавляется к набору сообщений, на которые откликается объект, путем помещения определений методов в конкретные, четко обозначенные в маршруте поиска методов места. Полный список возможностей показан на рис. 7.7.

Прежде чем продолжить повествование, следует кое о чем предупредить. В качестве руководства для большинства разработчиков схема на рис. 7.7 выражает общую картину достаточно точно, но история на этом не заканчивается. Для основной массы программного кода приложений вполне достаточно вести

себя так, будто класс `Object` находится на вершине иерархии, но в зависимости от используемой вами версии Ruby с технической точки зрения это может быть не совсем верно. Если создается код, для которого, как вы думаете, этот вопрос может играть важную роль, убедитесь в том, что вы разбираетесь в иерархии рассматриваемой версии Ruby.

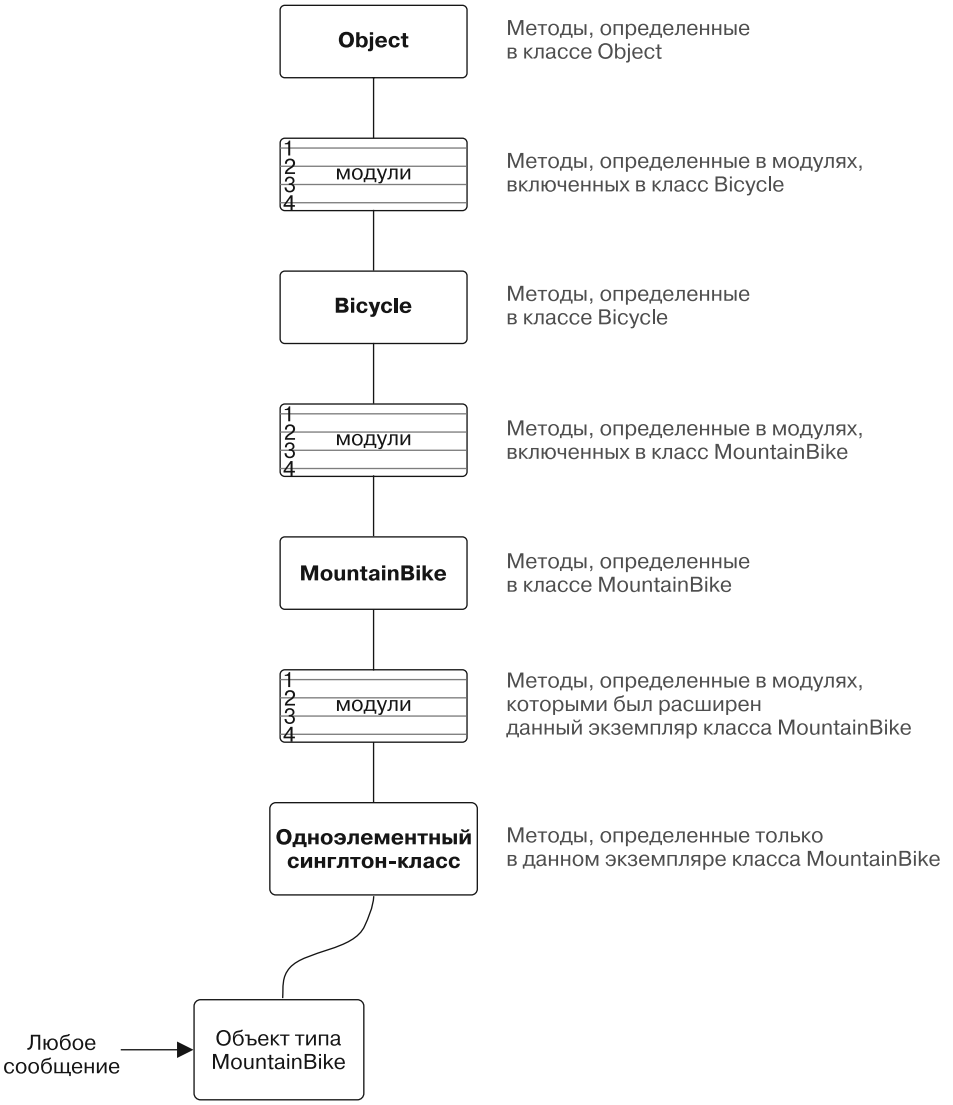

Рис. 7.7. Практически полное объяснение порядка поиска метода

Наследование ролевого поведения

После усвоения способов определения в модуле кода совместно используемых ролей и методов вы готовы написать поистине впечатляющий код. Представьте свои возможности: можно создать модули, включающие другие модули. Можно создать модули, переопределяющие методы, определенные в других модулях. Можно создать класс с глубокими вложениями иерархии наследования, а затем включить эти разнообразные модули в различные уровни иерархии.

Можно написать код, который невозможно будет понять, отладить или расширить.

Это очень эффективное, но весьма опасное в неумелых руках средство. Однако поскольку его эффективность позволяет вам также создавать простые структуры родственных объектов, которые удовлетворяют потребности вашего приложения, перед вами ставится задача не избегать этих технологий, а научиться пользоваться ими в нужных местах и правильным образом.

Первым шагом станет написание надлежаще наследуемого кода.

Написание наследуемого кода

Польза от иерархий наследования и упрощение ее сопровождения находится в прямой зависимости от качества кода. Более чем другие стратегии проектирования, совместное использование наследуемого поведения требует специфичной технологии создания программного кода, рассматриваемой в следующих разделах.

Выявление антишаблонов

Существуют два антишаблона, которые являются признаком того, что код может получить преимущества от наследования. Первый из них представлен объектом, который использует переменную с именем вроде `type` или `category`, предназначенную для определения, какое сообщение следует отправить в адрес `self`, где содержатся два близких, но немного разных типа. Это ночной кошмар для того, кто будет этот код сопровождать (потому что код придется менять при добавлении каждого нового типа). Подобный код должен пройти реорганизацию с целью использования классического наследования путем выделения общего кода в абстрактный родительский класс и создания подклассов для различных типов.

Реорганизация позволит создать новые подтипы путем добавления новых подклассов, которые расширят иерархию без изменения существующего кода.

Второй антишаблон выявляется при проверке отправляющим объектом класса объекта-получателя, чтобы определить, какое именно сообщение следует отправить, при этом упускается из виду наличие неявного типа. Это еще один ночной кошмар для того, кто будет заниматься сопровождением кода, поскольку код придется изменять при каждом появлении нового класса получателя. В этой ситуации все возможные объекты-получатели играют общую роль. Она должна быть запрограммирована в виде неявного типа, и получатели должны реализовывать интерфейс неявного типа. Как только это будет сделано, исходный объект сможет отправлять единое сообщение каждому получателю, оставаясь уверенным, что каждый получатель поймет общее сообщение, поскольку он играет вполне определенную роль.

Неявные типы могут использовать не только общий интерфейс, но и общее поведение. При этом общий код следует поместить в модуль, который нужно включить в каждый класс или объект, играющий роль.

Принуждение к абстракции

Весь код в абстрактном родительском классе должен применяться к каждому наследующему его классу. В родительских классах не должно быть кода, который применяется к отдельным подклассам. Это же ограничение применимо и к модулям: код в модуле должен применяться ко всем, кто его использует.

Из-за неверно определенной абстракции наследующие ее объекты содержат неверное поведение; попытки обхода этого неправильного поведения снизят качество кода. При взаимодействии с такими неудобными объектами программисты вынуждены быть в курсе их странностей и возможных зависимостей, которых лучше избегать.

Симптомом подобной проблемы могут послужить подклассы, переопределяющие метод для выдачи исключения типа «функция не реализована». Конечно, все определяется целесообразностью и иногда экономически выгоднее построить код именно таким образом, но вам все же следует отказываться от этих приемов. Когда подклассы переопределяют метод ради объявления, что *они этого не делают*, они опасно приближаются к заявлению, что они *не являются тем, чем должны быть*. Ничего хорошего из этого не выйдет.

Если четко определить абстракцию невозможно, то она, наверное, отсутствует; если общей абстракции не существует, то наследование вряд ли станет решением проблемы проектирования.

Соблюдение контракта

Подклассы выполняют условия *контракта*; они обещают стать заменой своим родительским классам. Замена возможна только тогда, когда объекты ведут себя ожидаемо и от подклассов *ожидается* согласованность с интерфейсом их родительских классов. Они должны откликаться на каждое сообщение в этом интерфейсе, получая те же виды входных данных и возвращая те же виды данных на выходе. Им не разрешается делать что-либо, что заставит других проверять их тип, чтобы узнать, как с ними обращаться или чего от них ждать.

Когда родительские классы накладывают ограничения на входные аргументы и возвращаемые значения, подклассы могут быть более свободными без нарушения контракта. Подклассы могут принимать входные параметры, на которые распространяются более широкие ограничения, и могут возвращать результаты, имеющие более узкие ограничения, и все это наряду с сохранением абсолютной заменимости для своих родительских классов.

Подклассы, не соблюдающие свой контракт, трудно использовать. Они слишком конкретизированы и не могут служить свободной заменой своим родительским классам. Эти подклассы объявляют, что они не настоящие *разновидности* своих родительских классов, и ставят под сомнение правильность всей иерархии.

ПРИНЦИП ПОДСТАНОВКИ ЛИСКОВ — LISKOV SUBSTITUTION PRINCIPLE (LSP)

Когда выполняются условия контракта, вы следуете принципу подстановки, названному в честь его создателя Барбары Лисков.

В предложенном ею принципе утверждается следующее.

Пусть q(x) является свойством, верным относительно объектов x некоторого типа T. Тогда q(y) также должно быть верным для объектов y типа S, где S является подтипом типа T.

Математики смогут разобраться в этом утверждении сразу же, как только его прочитают, а все остальные должны запомнить: чтобы в системе типов был какой-то смысл, подтипы должны служить заменой своим родительским типам, то есть годиться в качестве подстановки.

Соблюдение этого принципа позволяет создавать приложения, в которых подкласс может использоваться везде, где может использоваться его родительский класс и где объектам, включающим модули, можно доверять как выходящим на замену исполнителям роли модуля.

Использование схемы шаблонного метода

В основу технологии программирования для создания наследуемого кода положена схема шаблонного метода. Она позволяет отделить абстрактное от конкретного. Абстрактный код определяет алгоритмы, а конкретные наследники этой абстракции привносят конкретику путем переопределения этих шаблонных методов.

Шаблонные методы представляют изменяющиеся части алгоритма, и их создание заставляет вас принимать конкретные решения о том, что изменяется, а что — нет.

Превентивное отделение классов

Избегайте написания кода, требующего от его наследников отправлять сообщение `super`; вместо этого используйте хук-сообщения, чтобы позволить подклассам быть участниками, при этом освобождая их от обязанности знания абстрактного алгоритма. Наследование из-за своей собственной природы добавляет сильные зависимости от структуры и расположения кода. Написание кода, требующего от подклассов отправлять сообщение `super`, создает дополнительную зависимость, появления которой нужно по возможности избегать.

Хук-методы решают проблему отправки сообщения `super`, но, к сожалению, только для смежных уровней иерархии. Например, в главе 6 `Bicycle` отправляет хук-метод `local_spares`, который `MountainBike` переопределяет для предоставления конкретики. Этот хук-метод превосходно справляется со своей задачей, но исходная проблема проявится снова, если к иерархии добавить еще один уровень, создав под `MountainBike` подкласс `MonsterMountainBike`. Чтобы объединить собственные запасные части с запчастями родительского класса, подклассу `MonsterMountainBike` придется переопределить `local_spares` и внутри этого метода отправить сообщение `super`.

Создание неглубоких иерархий

Ограничения, связанные с хук-методами, являются одной из многих причин создания неглубоких иерархий.

Каждую иерархию можно представить в виде пирамиды, имеющей глубину и ширину. Глубина объекта определяется количеством родительских классов между ним и вершиной. А ширина определяется количеством его непосредственных подклассов. Форма иерархии определяется ее общей шириной и глубиной, и именно она обусловливает простоту использования, сопровождения и расширения. Несколько возможных разновидностей формы показаны на рис. 7.8.

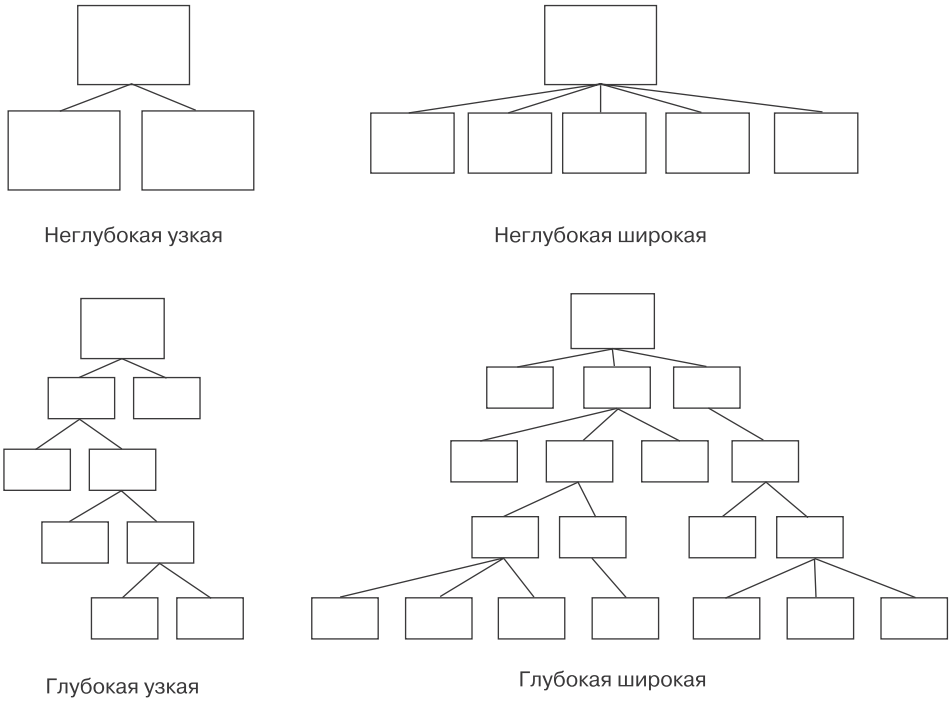

Рис. 7.8. Иерархии могут иметь различные формы

В неглубоких узких иерархиях проще разобраться. Неглубокие широкие иерархии немного сложнее. Глубокие узкие иерархии еще сложнее и, к сожалению, склонны к расширению (побочный эффект глубины). В глубоких широких иерархиях весьма трудно разобраться, их сопровождение обходится недешево, поэтому создания таких иерархий лучше избегать.

Проблема глубоких иерархий заключается в том, что они определяют весьма длинный маршрут поиска мест разрешения сообщений и предоставляют массу возможностей для объектов, находящихся по этому маршруту, добавлять поведение по мере прохождения мимо них сообщения. Поскольку объекты зависят от *всего*, что находится над ними, глубокая иерархия обладает большим набором встроенных зависимостей, каждая из которых со временем может измениться.

Еще одна проблема глубоких иерархий заключается в том, что программисты знакомы, как правило, только с классами в их верхних и нижних частях, то есть им известно только то поведение, которое реализовано на границах маршрута поиска. А классы в средней части испытывают дефицит внимания. И риск допустить ошибки при изменениях в этих смутно понимаемых средних классах намного выше.

Выводы

Когда объектам, играющим общую роль, требуется совместно используемое поведение, они получают его посредством модуля Ruby. Код, определенный в модуле, можно добавить к любому объекту, будь то экземпляр класса, сам класс или другой модуль.

Когда класс включает модуль, методы этого модуля попадают в тот же маршрут поиска, что и методы, получаемые благодаря наследованию. Поскольку в маршруте поиска методы модуля и унаследованные методы чередуются, приемы программирования модулей являются зеркальным отражением приемов, используемых для наследования. Поэтому в модулях должна применяться схема шаблонных методов, чтобы предложить объектам, которые их включают, предоставлять конкретику, а также в них должны быть реализованы хук-методы, чтобы не заставлять включающий модули объект отправлять сообщение `super` (и знать, таким образом, алгоритм).

Когда объект приобретает поведение, которое было определено где-либо в другом месте, независимо от того, что именно подразумевается под этим *«где-либо»*, родительский класс или включенный модуль, приобретающий объект, берет на себя обязательства соблюдать подразумеваемый контракт. Этот контракт определяется исходя из принципа подстановки Лисков, являющегося математическим понятием, утверждающим, что подтип должен служить заменой своему родительскому типу (в Ruby это означает, что объект должен действовать в соответствии с тем, чем он себя провозглашает).

Глава 8
Объединение объектов путем составления композиции

Составление композиции заключается в объединении отдельных частей в сложное целое, при котором это целое становится чем-то более существенным, чем просто сумма его составляющих. Музыка, к примеру, появляется в результате составления композиции.

Можно, конечно, не думать о своем приложении как о музыке, но аналогия вполне уместна. Музыкальная партитура пятой симфонии Бетховена представляет собой длинный перечень отдельных, не зависящих друг от друга нот. Но достаточно прослушать эту симфонию всего лишь раз, чтобы понять: она, конечно, *состоит* из нот, но это не просто ноты. Это нечто большее.

Вы можете создавать свои программы точно так же, составляя объектно-ориентированные композиции для объединения простых независимых объектов в более крупное и сложное целое. В композиции более крупный объект подключается к своим частям посредством отношения обладания *has-a* (то есть через отношение, когда более крупный обладает частями). Велосипед состоит из частей. Велосипед является объектом, содержащим составные части. Определению композиции присуща идея, что велосипед не только имеет составные части, но и что эти части связываются с ним через интерфейс. Часть является *ролью*, а велосипеды охотно сотрудничают с любым объектом, играющим роль.

Эта глава обучит вас технологиям составления объектно-ориентированных композиций. Она начинается с примера, затем мы перейдем к рассмотрению соотносящихся друг с другом сильных и слабых сторон композиции и наследования и я дам рекомендации, как выбирать между альтернативными технологиями проектирования.

Составление композиции Bicycle (велосипед) из Parts (частей)

Этот раздел начинается с того места, на котором закончился пример `Bicycle` в главе 6. Если код этого примера вы уже подзабыли, вернитесь к окончанию главы 6. В данном разделе приводится несколько реорганизаций этого кода, позволяющих постепенно заменить наследование композицией.

Обновление класса Bicycle

На данный момент класс `Bicycle` в иерархии наследования представляет собой абстрактный родительский класс, вам требуется преобразовать его под использование композиции. На первом этапе нужно отвлечься от имеющегося кода и подумать о том, как составить композицию под названием «велосипед».

В обязанность класса `Bicycle` входит ответ на сообщение `spares`. Это сообщение должно привести к возврату списка запчастей. Велосипеды состоят из частей, и отношение «велосипед — части» вполне естественно воспринимается как композиция. Если создается объект для хранения всех частей велосипеда, то сообщение `spares` можно переслать этому новому объекту.

Новый класс имеет смысл назвать «части» (Parts). На объект типа `Parts` можно возложить обязанность по хранению списка составных частей велосипеда и по знанию того, какие из этих частей могут нуждаться в замене запасными частями (spares). Заметьте, что этот объект представляет собой коллекцию частей, а не отдельную часть.

Этот замысел отражен в диаграмме последовательности на рис. 8.1. Здесь класс `Bicycle` отправляет сообщение `spares` своему объекту `Parts`.

Каждому велосипеду `Bicycle` нужен объект частей `Parts`; в частности, быть велосипедом `Bicycle` означает *обладать (have-a)* коллекцией частей `Parts`. Это отношение отражено в диаграмме классов на рис. 8.2.

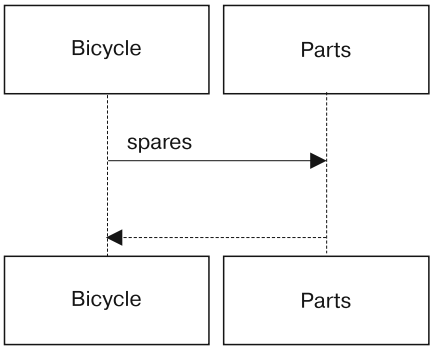

Рис. 8.1. Велосипед Bicycle задает вопрос частям Parts относительно запчастей spares

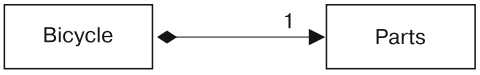

Рис. 8.2. У велосипеда Bicycle имеется отношение обладания (has-a) к частям Parts

На этой диаграмме показаны классы `Bicycle` и `Parts`, соединенные прямой линией. В месте соединения ее с `Bicycle` изображен черный ромб, показывающий наличие *композиции* — это означает, что велосипед `Bicycle` состоит из частей `Parts`. В месте соединения линии с `Parts` стоит цифра 1 — это означает, что на каждый объект типа `Bicycle` приходится только один объект типа `Parts`.

Преобразовать существующий класс `Bicycle` в эту новую конструкцию совсем нетрудно. Следует удалить основную часть его кода, добавить переменную `parts`, чтобы в ней содержался объект типа `Parts`, и перенаправить `spares` к `parts`. Новый класс `Bicycle` имеет следующий вид.

```
class Bicycle
  attr_reader :size, :parts

  def initialize(args={})
     @size  = args[:size]
     @parts = args[:parts]
  end

  def spares
    parts.spares
  end
end
```

Теперь обязанности `Bicycle` следующие: он должен знать свой размер, иметь в своем составе объект типа `Parts` и отвечать на сообщение `spares`.

Создание иерархии Parts

Все получилось довольно легко, но только потому, что основная часть кода в классе `Bicycle` занималась составными частями, а начального поведения, присущего велосипедам, было немного. Но потребности в только что удаленном из `Bicycle` поведении частей сохранились, и заставить этот код снова работать проще всего путем его включения в показанную ниже новую иерархию `Parts`.

```
class Parts
  attr_reader :chain, :tire_size

  def initialize(args={})
     @chain     = args[:chain] || default_chain
     @tire_size = args[:tire_size] || default_tire_size
     post_initialize(args)
  end

  def spares
    { tire_size: tire_size,
    chain: chain}.merge(local_spares)
  end

  def default_tire_size
    raise NotImplementedError
  end

  # это может быть переопределено подклассами
  def post_initialize(args)
    nil
  end

  def local_spares
    {}
  end

  def default_chain
   '10-speed'
  end
end

```

```
class RoadBikeParts < Parts
  attr_reader :tape_color

  def post_initialize(args)
    @tape_color = args[:tape_color]
  end

  def local_spares
    {tape_color: tape_color}
  end

  def default_tire_size
    '23'
  end
end

class MountainBikeParts < Parts
  attr_reader :front_shock, :rear_shock

  def post_initialize(args)
    @front_shock = args[:front_shock]
    @rear_shock  = args[:rear_shock]
  end

  def local_spares
    {rear_shock: rear_shock}
  end

  def default_tire_size
    '2.1'
  end
end
```

Код представляет собой практически точную копию иерархии `Bicycle` из главы 6; разница в том, что классы были переименованы, а переменная `size` удалена.

Произошедшее преобразование отображено в диаграмме класса на рис. 8.3. Теперь у нас есть абстрактный класс `Parts`. Класс `Bicycle` состоит из `Parts`. А у `Parts` имеются два подкласса — `RoadBikeParts` (части дорожного велосипеда) и `MountainBikeParts` (части горного велосипеда).

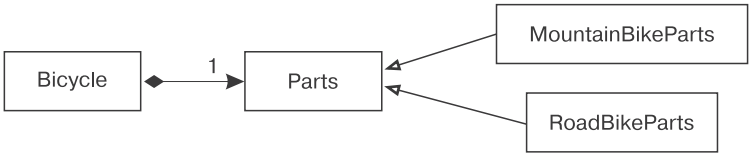

Рис. 8.3. Иерархия Parts

После реорганизации работа кода не нарушилась. Если проанализировать показанный ниже код, станет понятно, что независимо от того, к какому подклассу относится велосипед, `RoadBikeParts` или `MountainBikeParts`, он в состоянии корректно ответить на вопрос о своем размере и запасных частях (`size` и `spares`).

```
road_bike =
   Bicycle.new(
     size:  'L',
     parts: RoadBikeParts.new(tape_color: 'red'))

road_bike.size # -> 'L'

road_bike.spares
# -> {:tire_size=>"23",
#     :chain=>"10-speed",
#     :tape_color=>"red"}

mountain_bike =
  Bicycle.new(
    size:  'L',
    parts: MountainBikeParts.new(rear_shock: 'Fox'))

mountain_bike.size # -> 'L'

mountain_bike.spares
# -> {:tire_size=>"2.1",
#     :chain=>"10-speed",
#    :rear_shock=>"Fox"}
```

Изменения и улучшения не впечатляют своим размахом. Тем не менее эта реорганизация выявила одно полезное обстоятельство. Стало совершенно очевидно, что для начала классу `Bicycle` хватило и столь малого объема конкретного кода. Основная часть показанного выше кода относилась к отдельным частям, а иерархия `Parts` теперь так и просится на очередную реорганизацию.

Составление композиции для объекта Parts

Изначально в списке составных частей содержатся отдельные части. Настало время добавить класс, чтобы представить отдельно взятую часть. Вполне резонно, что в качестве имени этого класса лучше всего подойдет слово `Part`, но введение класса `Part` при наличии класса `Parts` усложняет дальнейшее рассмотрение вопроса. Использование слова parts для ссылки на коллекцию `Part`-объектов, когда это же слово уже применяется для ссылки на отдельный `Parts`-объект, вносит путаницу. Но в предыдущей фразе был показан прием, позволяющий обойти проблему передачи истинного смысла сказанного, когда речь заходит о `Part` и `Parts`; можно за именем класса указывать слово «объект» (при необходимости — во множественном числе).

Проблему передачи смысла можно также обойти, с самого начала выбрав другие имена классов, но они могут быть не настолько говорящими и способны создать еще одну смысловую проблему. Ситуация, аналогичная `Parts` и `Part`, встречается довольно часто, поэтому с ней стоит разобраться, чтобы избежать двусмысленности. Выбор этих имен классов требует точности в обмене информацией, что само по себе вполне достойная цель.

Итак, все просто: есть `Parts`-объект, он может состоять из множества `Part`-объектов.

Создание Part

На рис. 8.4 представлена новая диаграмма последовательности, показывающая диалог между `Bicycle` и его `Parts`-объектом, а также между `Parts`-объектом и его `Part`-объектами. `Bicycle` отправляет сообщение `spares` в адрес `Parts`-объекта, а затем `Parts`-объект отправляет сообщение `needs_spare` каждому `Part`-объекту.

Такое изменение конструкции требует создания нового объекта `Part`. Теперь `Parts`-объект состоит из `Part`-объектов, что отображено в диаграмме класса на рис. 8.5. Пометка 1..* показывает, что у `Parts`-объекта может быть от одного до нескольких `Part`-объектов.

Введение нового класса `Part` упрощает содержимое существующего класса `Parts`, который теперь становится простой оболочкой массива из `Part`-объектов. `Parts` может отфильтровать свой список `Part`-объектов и вернуть только те объекты, которым нужны запчасти. Расположенный ниже код показывает три

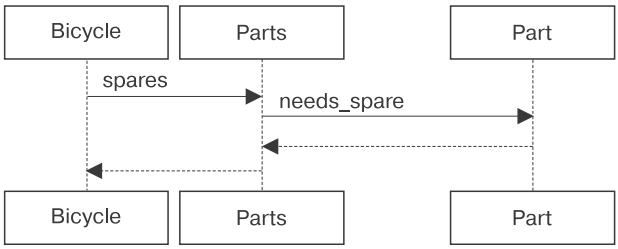

Рис. 8.4. Bicycle отправляет сообщение spares в адрес Parts, а Parts отправляет сообщение needs_spare каждому Part

Рис. 8.5. Bicycle обладает одним Parts-объектом, который, в свою очередь, обладает множеством Part-объектов

класса: существующий класс `Bicycle`, обновленный класс `Parts` и только что введенный класс `Part`.

```
class Bicycle
  attr_reader :size, :parts

  def initialize(args={})
     @size  = args[:size]
     @parts = args[:parts]
  end

  def spares
    parts.spares
  end
end

class Parts
  attr_reader :parts

  def initialize(parts)
    @parts = parts
  end

  def spares
    parts.select {|part| part.needs_spare}
  end
```

```
end

class Part
  attr_reader :name, :description, :needs_spare

  def initialize(args)
    @name        = args[:name]
    @description = args[:description]
    @needs_spare = args.fetch(:needs_spare, true)
  end
end
```

Теперь при наличии этих трех классов можно создавать отдельные `Part`-объекты. Следующий код создает несколько различных частей и сохраняет каждую из них в переменной экземпляра.

```
chain =
   Part.new(name: 'chain', description: '10-speed')

road_tire =
   Part.new(name: 'tire_size', description: '23')

tape =
   Part.new(name: 'tape_color', description: 'red')

mountain_tire =
  Part.new(name: 'tire_size', description: '2.1')

rear_shock =
  Part.new(name: 'rear_shock', description: 'Fox')

front_shock =
  Part.new(
    name: 'front_shock',
    description: 'Manitou',
    needs_spare: false)
```

Отдельные `Part`-объекты могут быть сгруппированы в `Parts`. Показанный ниже код объединяет `Part`-объекты дорожного велосипеда в подходящую для этого велосипеда коллекцию `Parts`.

```
road_bike_parts =
  Parts.new([chain, road_tire, tape])
```

Разумеется, этот промежуточный шаг можно пропустить и собрать `Parts`-объект на лету при создании `Bicycle`, как показано в строках 4–6 и 22–25.

```
1  road_bike =
2     Bicycle.new(
3       size:  'L',
4       parts: Parts.new([chain,
5                         road_tire,
6                         tape]))
7
8  road_bike.size # -> 'L'
9
10 road_bike.spares
11 # -> [#<Part:0x00000101036770
12 # @name="chain",
13 # @description="10-speed",
14 # @needs_spare=true>,
15 # #<Part:0x0000010102dc60
16 # @name="tire_size",
17 # etc ...
18
19 mountain_bike =
20   Bicycle.new(
21     size:  'L',
22     parts: Parts.new([chain,
23                       mountain_tire,
24                       front_shock,
25                       rear_shock]))
26
27 mountain_bike.size # -> 'L'
28
29 mountain_bike.spares
30 # -> [#<Part:0x00000101036770
31 #        @name="chain",
32 #        @description="10-speed",
33 #        @needs_spare=true>,
34 #     #<Part:0x0000010101b678
35 #         @name="tire_size",
36 #         etc ...
```

Судя по строкам 8–17 и 27–34, это новое построение кода работает весьма неплохо и ведет себя *почти* также, как и старая иерархия `Bicycle`. Однако есть одно отличие: имевшийся в старом `Bicycle` метод `spares` возвращал хеш, а новый метод `spares` возвращает массив `Part`-объектов.

Возможно, кто-то будет считать эти объекты экземплярами класса `Part`, но композиция подсказывает, что их нужно рассматривать в качестве объектов, которые просто играют роль `Part`. Они не должны представлять собой *нечто вроде* класса `Part`, им просто нужно играть его роль, то есть реагировать на имя, описание и сообщение `needs_spare`.

Придание объекту Parts большей схожести с массивом

Код, конечно, работает, но в нем определенно есть что улучшить. Вернемся на минутку назад и подумаем о методах `parts` и `spares` класса `Bicycle`. Возвращаемые на соответствующие сообщения объекты ведут себя по-разному, хотя создается впечатление, что возвращаться должны однотипные данные. Посмотрим, что получится, если каждого из них спросить о размере.

В строке 1 показанного ниже кода метод `spares` охотно отвечает, что размер объекта равен 3. Но когда тот же вопрос задается методу `parts`, ничего хорошего, как видно из строк 2–4, из этого не получается.

```
1 mountain_bike.spares.size # -> 3
2 mountain_bike.parts.size
3 # -> NoMethodError:
4 # undefined method 'size' for #<Parts:...>
```

Код в строке 1 работает, потому что `spares` возвращает массив (из `Part`-объектов), а экземпляр класса массива `Array` понимает сообщение `size`. Код в строке 2 дает сбой, потому что `parts` возвращает экземпляр класса `Parts`, который этого сообщения не понимает.

Подобные сбои будут преследовать вас, пока вы не избавитесь от этого кода. Оба результата *кажутся* массивами. И вы неизбежно будете считать их массивами, притом что это будет подтверждаться лишь в половине случаев, заставляя вас снова наступать на те же грабли. `Parts`-объект не ведет себя как массив, и все попытки считать его таковым неизменно провалятся.

Данную проблему можно решить путем добавления к классу `Parts` метода `size`. Нужно реализовать метод, пересылающий сообщение `size` настоящему массиву.

```
1  def size
2    parts.size
3  end
```

Однако это изменение способно свернуть класс `Parts` на скользкую дорожку. Если вы именно так и сделаете, то вскоре вам захочется, чтобы `Parts` реагировал на сообщение `each`, затем на `sort`, а следом и на все остальное, что есть в экземплярах класса `Array`. И это никогда не кончится; чем больше вы будете делать `Parts` похожим на массив, тем выше будут ожидания от него качеств, присущих массиву.

Возможно, `Parts` по своей сути и есть массив, хотя и с добавлением дополнительных особенностей поведения. И вы можете сделать из него массив. В следующем примере показана новая версия класса `Parts`, который теперь является подклассом `Array`.

```
1  class Parts < Array
2    def spares
3      select {|part| part.needs_spare}
4    end
5  end
```

Данный код — весьма простое выражение идеи о том, что `Parts` является специализацией `Array`; в идеальном объектно-ориентированном языке это решение было бы абсолютно правильным. К сожалению, язык Ruby еще не достиг полного совершенства, и у данной конструкции имеется скрытый дефект.

Суть проблемы показана в следующем примере. Когда `Parts` становится подклассом `Array`, он наследует все поведение `Array`. Это поведение включает и метод `+`, который объединяет два массива и возвращает третий массив. В строках 3 и 4 метод `+` объединяет два существующих экземпляра `Parts` и сохраняет результат в переменной `combo_parts`.

Создается впечатление, что этот код работает, и теперь `combo_parts` содержит соответствующее предположениям количество частей (строка 7). Но кое-что явно не в порядке. Как показано в строке 12, `combo_parts` не в состоянии ответить на вопрос о запчастях `spares`.

Основная причина проблемы раскрывается в строках 15–17. Хотя объекты, объединенные с помощью метода `+`, были экземплярами `Parts`, объект, воз-

вращенный методом +, стал экземпляром Array, а Array не понимает сообщения spares.

```
# Parts наследует '+' из Array, поэтому вы можете
# собрать два объекта Parts вместе.
combo_parts =
   (mountain_bike.parts + road_bike.parts)

# '+' конечно же объединяет Parts
combo_parts.size # -> 7

# но объект, возвращаемый '+',
# не понимает сообщения 'spares'
combo_parts.spares
# -> NoMethodError: метод 'spares'
#      для #<Array:...> не определен

mountain_bike.parts.class # -> Parts
road_bike.parts.class     # -> Parts
combo_parts.class         # -> Array !!!
```

Дело в том, что в Array имеется множество методов, возвращающих новые экземпляры класса Array, но не новые экземпляры ваших подклассов. Класс Parts по-прежнему вводит вас в заблуждение, получается, вы просто подменили одну проблему другой. Ранее вы расстраивались, обнаружив отсутствие в классе Parts реализации метода size, а теперь можете удивиться, обнаружив, что объединение двух Parts-объектов возвращает результат, которому непонятно сообщение spares.

Вы уже видели три различные реализации класса Parts. Первая реагировала только на сообщения spares и parts и своим поведением не была похожа на массив, она просто содержала массив данных. Во вторую реализацию класса Parts в качестве небольшого усовершенствования был добавлен метод size, возвращающий размер его внутреннего массива. Самая последняя версия реализации класса Parts стала подклассом Array, в результате чего в ней появилось полноценное поведение, присущее массиву, но, как показал приведенный ранее пример, экземпляр Parts по-прежнему демонстрирует неожиданные стороны своего поведения.

Становится ясно, что идеального решения не существует, поэтому настала пора принять трудное решение. Пусть исходная реализация не в состоянии реагировать на сообщение size, но она способна проявить себя достаточно

хорошо, при этом можно смириться с отсутствием у нее поведения, присущего массивам, и вернуться к ее использованию. Если нужна реакция на `size` (и только на `size`), возможно, лучше добавить только этот один метод и довольствоваться предложенным вторым вариантом реализации. Если же вы можете смириться с возможностью возникновения досадных ошибок или абсолютно точно знаете, что никогда не столкнетесь с ними, то, возможно, есть смысл сделать реализацию подклассом `Array` и закрыть вопрос.

Следующее решение находится где-то посередине между сложностью и практичностью. Класс `Parts`, показанный ниже, перенаправляет сообщения `size` и `each` своему массиву `@parts` и включает `Enumerable` для получения общих методов обхода элементов массива и осуществления поиска в нем. Эта версия `Parts` не обладает полноценным поведением, присущим экземплярам класса `Array`, но она по крайней мере выполняет все заявленные действия.

```
require 'forwardable'
class Parts
  extend Forwardable
  def_delegators :@parts, :size, :each
  include Enumerable

  def initialize(parts)
    @parts = parts
  end

 def spares
   select {|part| part.needs_spare}
 end
end
```

Отправка сообщения `+` экземпляру *этого* класса `Parts` приводит к выдаче исключения, связанного с отсутствием метода — `NoMethodError`. Но поскольку теперь `Parts` реагирует на `size`, `each` и на все, что связано с перечислениями (Enumerable), а также неизменно выдает исключения, когда этот объект ошибочно принимают за настоящий массив, с применением этого кода можно согласиться. В следующем примере показано, что теперь на сообщение `size` могут реагировать и `spares`, и `parts`.

```
mountain_bike =
   Bicycle.new(
     size:  'L',
```

```
 4      parts: Parts.new([chain,
 5                        mountain_tire,
 6                        front_shock,
 7                        rear_shock]))
 8
 9 mountain_bike.spares.size # -> 3
10 mountain_bike.parts.size  # -> 4
```

Итак, в вашем распоряжении опять появились работоспособные версии классов `Bicycle`, `Parts` и `Part`. Настало время переосмыслить конструкцию.

Изготовление Parts-объектов

Посмотрите на строки 4–7 предыдущего примера. `Part`-объекты, хранящиеся в переменных `chain`, `mountain_tire` и т. д., были созданы настолько давно, что вы о них могли уже забыть. Подумайте об основе тех знаний, которые представлены этими четырьмя строками: где-то в вашем приложении некий объект должен знать, как создавать эти `Part`-объекты. А в строках 4–7 именно *это* место должно быть в курсе, что именно эти четыре конкретных объекта используются в горных велосипедах.

Такой большой объем знаний может весьма легко распространиться по всему приложению, что крайне нежелательно, да и в этом нет никакой необходимости. Несмотря на обилие различных отдельных частей, подходящих комбинаций этих частей не так уж и много. Все могло бы существенно упроститься, если бы можно было дать описание различным велосипедам, которые затем использовать для изготовления неким магическим образом надлежащих `Parts`-объектов для любой разновидности велосипеда.

Дать описание комбинациям составных частей любого конкретного велосипеда совсем нетрудно. В показанном ниже примере кода это делается с помощью простого двумерного массива, в котором каждая строка состоит из трех столбцов. В первом содержится название составной части (`'chain'`, `'tire_size'` и т. д.), во втором дается ее описание (`'10-speed'`, `'23'` и т. д.), в третьем (необязательном) содержится булево значение, указывающее потребность данного компонента в запчастях. Значение для третьего столбца имеется только для переднего амортизатора `'front_shock'` в строке 9; что же касается других составных

частей, то лучше считать, что для них по умолчанию используется значение `true`, поскольку они нуждаются в запчастях.

```
road_config =
  [['chain', '10-speed'],
   ['tire_size', '23'],
   ['tape_color', 'red']]

mountain_config =
  [['chain', '10-speed'],
   ['tire_size', '2.1'],
   ['front_shock', 'Manitou', false],
   ['rear_shock', 'Fox']]
```

В отличие от хеша, этот простой двумерный массив не дает никакой структурной информации. Но *вы сами* разбираетесь в организации этой структуры и можете ее запрограммировать в новом объекте, создающем `Parts`-объекты.

Создание модуля PartsFactory

Как уже упоминалось в главе 3, объект, создающий другие объекты, называется фабрикой. Возможно, в силу вашего опыта работы с другими языками программирования это слово вызывает у вас настороженность, но данный случай поможет восстановить его репутацию. Слово *«фабрика»* (factory) не служит признаком чего-то труднопонимаемого, или искусственного, или слишком сложного, это слово разработчики объектно-ориентированных приложений употребляют для краткого обозначения сути объекта, создающего другие объекты. В Ruby фабрики не отличаются особой сложностью и нет смысла избегать их применения. В приводимом ниже примере кода показывается новый модуль `PartsFactory`. В его задачу входят получение массива, подобного ранее упомянутому, и изготовление `Parts`-объекта. Попутно он также может создавать `Part`-объекты, но это действие является закрытым. Его открытая обязанность — создание `Parts`-объекта.

Первая версия `PartsFactory` получает три аргумента: `config`, а также имена классов, используемых для `Part`-объектов и `Parts`-объектов. В строке 6 создается новый экземпляр `Parts`-объекта, инициализируемый массивом, состоящим

из Part-объектов и создаваемым на основе информации, которая находится в аргументе config.

```
module PartsFactory
  def self.build(config,
                 part_class  = Part,
                 parts_class = Parts)

  parts_class.new(
      config.collect {|part_config|
        part_class.new(
        name:        part_config[0],
        description: part_config[1],
        needs_spare: part_config.fetch(2, true))})
  end
end
```

Структура массива config этой фабрике известна. В строках 9–11 фабрика ожидает, что name будет в первом столбце, description — во втором, а needs_spare — в третьем.

Присутствие в фабрике знаний о структуре config влечет за собой два последствия. Во-первых, config может быть выражен очень сжато. Поскольку PartsFactory понятна внутренняя структура config, этот аргумент может быть указан в виде массива, а не хеша. Во-вторых, согласившись с тем, что config будет массивом, придется всегда создавать новые Parts-объекты только с использованием фабрики. Создание нового Parts-объекта с помощью иного механизма потребует продублировать те знания, которые запрограммированы в строках 9–11.

Теперь при наличии PartsFactory вы можете воспользоваться определенными выше конфигурационными массивами для того, чтобы без особого труда создать новый Parts-объект.

```
road_parts = PartsFactory.build(road_config)
# -> [#<Part:0x00000101825b70
#       @name="chain",
#       @description="10-speed",
#       @needs_spare=true>,
#    #<Part:0x00000101825b20
#       @name="tire_size",
#          и т. д. ...
```

```
 9
10 mountain_parts = PartsFactory.build(mountain_config)
11 # -> [#<Part:0x0000010181ea28
12 #        @name="chain",
13 #        @description="10-speed",
14 #        @needs_spare=true>,
15 #     #<Part:0x0000010181e9d8
16 #        @name="tire_size",
17 #        и т. д. ...
```

`PartsFactory` в сочетании с новыми конфигурационными массивами изолирует все знания, необходимые для создания надлежащего `Parts`-объекта. Прежде эта информация была разбросана по приложению, а теперь содержится только в одном этом классе и в этих двух массивах.

Применение PartsFactory

Давайте после создания и запуска фабрики `PartsFactory` еще раз взглянем на класс `Part` (код которого повторяется ниже). Для него используется весьма простой код. Кроме того, единственная *более или менее* сложная строка кода (`fetch` в строке 7) продублирована в `PartsFactory`. Если фабрика `PartsFactory` создает каждый `Part`-объект, то классу `Part` этот код не понадобится. А если этот код из класса `Part` удалить, то в нем почти ничего не останется, и тогда весь класс `Part` можно заменить простым объектом класса `OpenStruct`.

```
1 class Part
2    attr_reader :name, :description, :needs_spare
3
4    def initialize(args)
5      @name        = args[:name]
6      @description = args[:description]
7      @needs_spare = args.fetch(:needs_spare, true)
8    end
9 end
```

Имеющийся в Ruby класс `OpenStruct` очень похож на уже встречавшийся ранее класс `Struct`, он предоставляет удобный способ объединения нескольких атрибутов в объект. Разница между этими классами в том, что `Struct` получает аргументы инициализации в позиционированном порядке, а `OpenStruct` получает для инициализации хеш, а затем извлекает из него аргументы.

В удалении класса `Part` есть вполне определенный смысл: код упрощается, и вам уже может никогда не понадобиться ничего сложнее того, что есть сейчас. Удалив класс `Part`, можно будет удалить все его следы, а затем изменить `PartsFactory` под использование `OpenStruct` для создания объекта, играющего роль `Part`. В следующем примере кода показана новая версия фабрики `PartFactory`, где создание части было реорганизовано в собственный метод самой фабрики (строка 9).

```
1 require 'ostruct'
2 module PartsFactory
3   def self.build(config, parts_class = Parts)
4     parts_class.new(
5       config.collect {|part_config|
6         create_part(part_config)})
7   end
8
9   def self.create_part(part_config)
10    OpenStruct.new(
11      name:        part_config[0],
12      description: part_config[1],
13      needs_spare: part_config.fetch(2, true))
14  end
15 end
```

Теперь в строке 13 осталось единственное место в приложении, в котором исходное значение для `needs_spare` устанавливается в `true`, поэтому фабрика `PartsFactory` должна нести единоличную ответственность за изготовление `Parts`-объектов.

Новая версия `PartsFactory` вполне работоспособна. Как показано ниже, она возвращает `Parts`-объект, содержащий массив `OpenStruct`-объектов, каждый из которых играет роль какой-либо части (`Part`).

```
1 mountain_parts = PartsFactory.build(mountain_config)
2 # -> <Parts:0x000001009ad8b8 @parts=
3 #      [#<OpenStruct name="chain",
4 #                    description="10-speed",
5 #                    needs_spare=true>,
6 #      #<OpenStruct name="tire_size",
7 #                   description="2.1",
8 #                   и т.д. ...
```

Bicycle в виде композиции

В следующем примере кода демонстрируется, что теперь `Bicycle` использует композицию. В этом примере показаны `Bicycle`, `Parts` и `PartsFactory`, а также конфигурационные массивы для дорожного и горного велосипедов.

Велосипед `Bicycle` *обладает (has-a)* объектом составных частей `Parts`, который, в свою очередь, *обладает* коллекцией из `Part`-объектов. `Parts` и `Part` могут существовать в виде классов, но объекты, в которых они содержатся, считают их ролями. `Parts` является классом, играющим `Parts`-роль; в нем реализуется метод `spares`. Роль `Part` исполняется классом `OpenStruct`, в котором реализуются `name`, `description` и `needs_spare`.

Следующие 54 строки кода полностью заменяют 66-строчную иерархию наследования из главы 6.

```
class Bicycle
  attr_reader :size, :parts

  def initialize(args={})
     @size  = args[:size]
     @parts = args[:parts]
  end

  def spares
    parts.spares
  end
end

require 'forwardable'
class Parts
  extend Forwardable
  def_delegators :@parts, :size, :each
  include Enumerable

  def initialize(parts)
    @parts = parts
  end

  def spares
    select {|part| part.needs_spare}
```

```
  end
end

require 'ostruct'
module PartsFactory
  def self.build(config, parts_class = Parts)
    parts_class.new(
    config.collect {|part_config|
    create_part(part_config)})
  end

  def self.create_part(part_config)
    OpenStruct.new(
      name:        part_config[0],
      description: part_config[1],
      needs_spare: part_config.fetch(2, true))
  end
end

road_config =
  [['chain',       '10-speed'],
   ['tire_size',   '23'],
   ['tape_color',  'red']]

mountain_config =
  [['chain',       '10-speed'],
   ['tire_size',   '2.1'],
   ['front_shock', 'Manitou', false],
   ['rear_shock',  'Fox']]
```

Новый код работает почти так же, как и предыдущая иерархия `Bicycle`. Единственное отличие заключается в том, что сообщение `spares` теперь вместо хеша возвращает массив похожих на `Part` объектов, в чем можно убедиться, изучив показанные ниже строки 7 и 15.

```
road_bike =
   Bicycle.new(
     size:  'L',
     parts: PartsFactory.build(road_config))

road_bike.spares
```

```
# -> [#<OpenStruct name="chain", и т. д. ...

mountain_bike =
  Bicycle.new(
  size:  'L',
  parts: PartsFactory.build(mountain_config))

mountain_bike.spares
# -> [#<OpenStruct name="chain", и т. д. ...
```

Теперь при наличии нового класса существенно упрощается создание новой разновидности велосипеда.

Добавление поддержки лежачего велосипеда потребовало в главе 6 добавления новых 19 строк кода. Сейчас эту же задачу можно выполнить с помощью трех строк конфигурации (строки 2–4).

```
recumbent_config =
   [['chain', '9-speed'],
    ['tire_size', '28'],
    ['flag', 'tall and orange']]

recumbent_bike =
   Bicycle.new(
     size:  'L',
     parts: PartsFactory.build(recumbent_config))

recumbent_bike.spares
# -> [#<OpenStruct
#       name="chain",
#       description="9-speed",
#       needs_spare=true>,
#     #<OpenStruct
#       name="tire_size",
#       description="28",
#       needs_spare=true>,
#     #<OpenStruct
#       name="flag",
#       description="tall and orange",
#       needs_spare=true>]
```

Как показано в строках 11–23, теперь новый велосипед можно создать путем описания его составных частей.

АГРЕГАЦИЯ: ОСОБЫЙ ВИД КОМПОЗИЦИИ

Вам уже знакомо понятие делегирования, это когда один объект получает сообщение и переправляет его другому объекту. Делегирование создает зависимости: получающий объект должен распознать сообщение и знать, куда его переслать.

Композиция зачастую применяет делегирование, но это понятие имеет более широкое толкование. Созданный в виде композиции объект состоит из частей, с которыми он ожидает взаимодействия через четко определенные интерфейсы.

Композиция дает описание отношению обладания (has-a). В блюдах есть приправы, в университетах имеются кафедры, а у велосипедов есть составные части. Блюда, университеты и велосипеды являются композиционными объектами, приправы, кафедры и части — ролями. Композиционные объекты зависят от интерфейса роли.

Поскольку блюда взаимодействуют с приправами с помощью интерфейса, новым объектам, желающим играть роль приправ, нужно всего лишь реализовать этот интерфейс. Ранее неизвестные приправы беспрепятственно и взаимозаменяемо входят в состав блюд.

Термин «композиция» может создать небольшую путаницу, поскольку он применяется к двум несколько отличающимся друг от друга понятиям. Приведенное ранее определение относится к самому широкому толкованию этого термина. В большинстве же случаев, когда встречается термин «композиция», он означает всего лишь общую форму обладания (has-a) одного объекта другим.

В соответствии с формальным определением этот термин означает нечто более конкретное: он показывает отношение обладания, где содержащийся объект лишен существования, независимого от содержащего его объекта. Учитывая это более четко выраженное определение, вы понимаете, что суть термина заключается не только в том, что блюдо имеет приправы, но и в том, что при приготовлении блюда расходуются и приправы.

Образовавшийся в определении пробел заполняется термином «агрегация». Понятие агрегации практически совпадает с понятием композиции, за исключением того, что содержащийся объект имеет свое собственное независимое существование. Университеты состоят из кафедр, за которыми закреплены преподаватели. Если ваше приложение управляет множеством университетов и ему известны тысячи преподавателей, то вполне разумно ожидать, что даже при ликвидации кафедры (например, из-за закрытия университета) преподаватели никуда не денутся.

Отношение «университет — кафедра» — пример композиции (в ее строгом понимании), а отношение «кафедра — преподаватель» — пример агрегации. Ликвидация кафедры не приводит к ликвидации преподавателей, они существуют сами по себе.

Такая разница между композицией и агрегацией может не оказать существенного влияния на ваш код. Но теперь, когда вам знакомы оба этих термина, понятие композиции можно применять для ссылки на обе разновидности отношений и уточнять их истинную суть только по мере необходимости.

Выбор между наследованием и композицией

Следует помнить, что классическое наследование является *технологией создания упорядоченной структуры кода*. Поведение разбросано по объектам, а эти объекты организованы в отношения между классами, представляющие собой автоматическую пересылку сообщений, вызывающих надлежащее поведение. Все это нужно представлять следующим образом: издержки на упорядочение объектов в иерархию окупаются бесплатной пересылкой сообщений.

Композиция является альтернативой, меняющей местами издержки и выгоды. В композиции отношения между объектами не оформляются созданием программного кода в классовой иерархии, вместо этого объекты существуют сами по себе, в результате чего они должны точно знать обстановку и пересылать сообщения друг другу. Композиция позволяет объектам обладать структурной независимостью, но за счет издержек на явную пересылку сообщений.

Изучив примеры наследования и составления композиции, можно обдумать, когда именно их следует применять. Здесь действует общее правило: столкнувшись с проблемой, которую можно решить с помощью составления композиции, нужно отдать приоритет применению композиции. Если четко определить наследование в качестве наилучшего решения невозможно, используйте композицию. В композиции намного меньше встроенных зависимостей, чем в наследовании, и зачастую именно она — наилучший вариант.

Наследование *является* самым подходящим решением, когда его использование сулит большие выгоды при низкой степени риска. В данном разделе проведено сравнительное исследование издержек и выгод от применения наследования и композиции и предложены рекомендации по выбору более выгодной системы отношений.

Приемлемость наследования

Для взвешенного выбора в пользу наследования требуется четкое понимание издержек и выгод его применения.

Выгоды использования наследования

В главе 2 были озвучены четыре цели процесса программирования: код должен быть понятным, целесообразным, пригодным для повторного использования и образцовым. При правильном применении наследования можно преуспеть в достижении второй, третьей и четвертой целей.

Методы, определение которых находится ближе к высшей точке иерархии наследования, имеют повсеместное влияние, поскольку вершина иерархии действует как уровень, многократно увеличивающий их влияние. Изменения, вносимые в эти методы, распространяются вниз по дереву наследования. Поэтому грамотно смоделированные иерархии отличаются *целесообразностью*, серьезные изменения в поведении могут достигаться за счет незначительных изменений в коде.

Используйте результаты наследования в коде, который может быть описан как *открытый-закрытый*: иерархии открыты для расширения, но остаются закрытыми для изменений. Добавление нового подкласса к существующей иерархии требует отказаться от внесения изменений в существующий код. Иерархии таким образом *пригодны для повторного использования*, потому что позволяют легко и просто создавать новые подклассы для адаптации приложения под новые варианты.

Правильно выстроенная иерархия легко поддается расширению. Иерархия является воплощением абстракции, и каждый новый подкласс подключает новые конкретности. Существующей схемы придерживаться довольно просто, поэтому она может стать вполне естественным выбором любого программиста, которому вменяется в обязанность создание новых подклассов. Следовательно, иерархии могут быть *образцовыми*, по своей природе они сами же и предоставляют руководство по написанию кода для их расширения.

Чтобы осознать ценность организации кода с применением наследования, достаточно обратиться к истокам самих объектно-ориентированных языков. Отличным примером может послужить имеющийся в Ruby класс `Numeric`. Классы `Integer` и `Float` смоделированы в виде подклассов `Numeric`, образующееся при этом отношение можно назвать термином *«является» (is-a)*. Целые числа (integers) и числа с плавающей точкой (floats) по своей сути являются *числами*. Предоставление этим двум классам возможности использования общей абстракции — наиболее выгодный способ организации кода.

Издержки применения наследования

Опасения, касающиеся использования наследования, следующие: во-первых, это страх ошибиться с выбором наследования для решения проблемы, которую невозможно решить с помощью этой технологии. Если допустить подобную ошибку, то неизбежно наступит момент, когда потребуется добавить поведение, но окажется, что сделать это не так-то просто. Из-за выбора неправильной модели новое поведение не впишется в нее, и вам придется дублировать код или заниматься его реструктуризацией. Во-вторых, даже при наличии весомых

аргументов в пользу решения проблемы с помощью наследования вы можете создать код, который будет применяться другими в совершенно неожиданных для вас целях. У программистов может появиться желание воспользоваться созданным вами поведением, но они могут не справиться с зависимостями, востребованными наследованием.

В предыдущем разделе, где рассматривались выгоды использования наследования, вполне обоснованно утверждалось, что наследование применимо только к грамотно смоделированной иерархии. *Целесообразность*, *пригодность для повторного использования* и *образцовость* сулят существенные выгоды, предоставляемые наследованием. Но если применить наследование к решению совершенно неподходящей для этого проблемы, то вы столкнетесь с сопутствующим ущербом.

Обратной стороной *целесообразности* являются слишком большие издержки при внесении изменений ближе к вершине неверно смоделированной иерархии. В таком случае тот эффект, который должен был помочь достижению цели, станет действовать в обратном направлении, и небольшие изменения приведут к разрушению всей конструкции.

Обратной стороной *пригодности для повторного использования* является невозможность добавления поведения в том случае, когда новые классы представляют собой помесь типов. Иерархия `Bicycle` в главе 6 потерпела фиаско, когда возникла необходимость в использовании лежачего горного велосипеда. В этой иерархии уже имелись подклассы для горного (`MountainBike`) и лежачего (`RecumbentBike`) велосипедов, но объединить свойства этих двух классов в одном объекте при текущем состоянии иерархии не удалось. Повторно воспользоваться существующим поведением без изменения этой иерархии было невозможно.

Обратной стороной *образцовости* является хаос, возникающий при попытке программиста-новичка расширить неверно смоделированную иерархию. Такие недостаточно проработанные иерархии не должны расширяться, они требуют реорганизации, но у новичков на это не хватает мастерства. Они вынуждены дублировать существующий код или добавлять зависимости от имен классов, но и то и другое только усугубляет существующие проблемы проектирования.

Таким образом, для наследования ответ на вопрос «Что получится, если я ошибусь?» особенно важен. По определению наследование не обходится без глубоко внедренного набора зависимостей. Подклассы зависят от методов,

определение которых находится в их родительских классах, *и* от автоматической пересылки сообщений, осуществляемой в адрес этих родительских классов. В этом и величайшая сила наследования, и его самое слабое место; подклассы в силу своей конструкции неизменно имеют привязку к классам, которые стоят выше них в иерархии. Эти встроенные зависимости усиливают эффекты от изменений, которым подвергаются родительские классы. Широкомасштабные изменения в поведении можно получить с помощью незначительных изменений в коде.

И наконец, при оценке целесообразности использования наследования нужно учесть предположения, кто будет работать с вашим кодом. Если код создается для служебного пользования в той области, с которой вы тесно связаны, вы можете достаточно точно предсказать будущее и быть уверенными, что для решаемых вами задач проектирования наследование станет экономически оправданным решением. Если код создается для более широкого круга пользователей, ваши способности предвидеть их потребности неизбежно снижаются, а вместе с ними и востребованность наследования в качестве части интерфейса.

Лучше обойтись без написания среды, требующей от пользователей вашего кода создавать для получения заданного вами поведения подклассы ваших объектов. Объекты их собственных приложений уже могут быть выстроены в иерархию, что помешает организации наследования от вашей среды.

Приемлемость композиции

Композиционные объекты имеют два основных отличия от объектов, построенных с помощью наследования. Они не зависят от структуры иерархии классов и пересылают свои собственные сообщения. Этими различиями обусловливается и совершенно иной перечень издержек и выгод.

Выгоды использования композиции

При использовании композиции вполне естественно стремиться создавать множество небольших объектов с простыми и понятными обязанностями, доступ к которым может осуществляться через четко определенные интерфейсы. Грамотно составленные объекты выделяются на фоне тех требований к коду, которые были сформулированы в главе 2.

Небольшие объекты имеют единственную обязанность, конкретизирующую их собственное поведение. Они *понятны*, то есть в их коде нетрудно разобраться,

кроме того, нетрудно предвидеть, что случится, если они изменятся. Помимо этого, независимость композиционных объектов от иерархии означает, что ими наследуется весьма незначительный объем кода, поэтому они, как правило, обладают иммунитетом от побочных эффектов, которые возникают в результате изменений классов, стоящих на более высоких ступенях иерархии.

Поскольку композиционные объекты обращаются к своим частям через интерфейс, добавление новой разновидности составной части требует всего лишь подключения нового объекта, соблюдающего интерфейс. С точки зрения композиционного объекта добавление нового варианта существующей части вполне *целесообразно* и не требует внесения изменений в код объекта.

По своей природе объекты, участвующие в композиции, невелики по размеру, обладают структурной независимостью и имеют четко определенные интерфейсы. Это способствует их беспрепятственному превращению во взаимозаменяемые компоненты. Поэтому качественно составленные композиционные объекты вполне *пригодны для повторного использования* в новых и неожиданных контекстах.

В лучшем случае использование композиции приводит к тому, что приложение выстраивается из простых подключаемых объектов, легко поддающихся расширению и устойчивых к изменениям.

Издержки использования композиции

Сильные стороны композиции способствуют проявлению ее слабых сторон. Композиционный объект полагается на множество своих частей. Даже если каждая часть имеет небольшой размер и вполне понятный код, общая работа всех частей в целом может быть не столь очевидной. Каждая отдельная часть может быть сама по себе *понятна*, а вот их совокупность может этим качеством не обладать.

За выгоды от структурной независимости приходится платить отсутствием автоматической пересылки сообщений. Композиционный объект должен абсолютно точно знать, какое сообщение и кому нужно переслать. Один и тот же код пересылки может понадобиться многим различным объектам, поскольку композиция не предоставляет способа его совместного использования.

Изучив выгоды и издержки, можно прийти к выводу, что композиция отлично подходит при наличии заранее прописанных правил сборки объекта из частей, но не сможет оказать помощь в решении задач по упорядочению кода для коллекции, состоящей из почти одинаковых частей.

Выбор характера отношений

Каждая технология — и классическое наследование (глава 6), и совместное использование поведения посредством модулей (глава 7), и композиция — является вполне достойным решением возлагаемых на нее задач. Чтобы снизить затраты, следует каждую из перечисленных технологий применять для решения именно тех задач, которые подходят им в наибольшей степени.

Прислушайтесь к советам мастеров объектно-ориентированного проектирования:

- наследование является специализацией (Бертран Мейер (Bertrand Meyer), Touch of Class: Learning to Program Well with Objects and Contracts);
- наследование больше всего подходит для добавления функциональных возможностей к уже существующим классам, когда будет использоваться основная часть старого кода, а добавляться будет относительно небольшой объем нового кода (Эрик Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес, Design Patterns: Elements of Reusable Object-Oriented Software);
- используйте композицию, когда поведение представляет собой нечто большее, чем просто совокупность ее частей (перефразирование Гради Буча (Grady Booch), Object-Oriented Analysis and Design).

Использование наследования для отношений вида «является»

Когда предпочтение отдается не композиции, а наследованию, делается ставка на то, что выгоды перевесят издержки. Кандидатами на моделирование с использованием классического наследования являются небольшие наборы объектов реального мира, которые вполне естественно можно отнести к статическим иерархиям специализаций.

Представьте себе игру, в ходе которой ее участники устраивают гонку на велосипедах. Они собирают велосипеды, «покупая» составные части. Одна из таких частей — амортизатор. Игрокам предлагаются шесть почти одинаковых амортизаторов (каждый отличается ценой и поведением).

Итак, все эти амортизаторы являются именно *амортизаторами*. Принадлежность к амортизаторам лежит в основе их идентичности; амортизаторы больше ни на что не делятся. Варианты амортизаторов имеют больше сходств,

чем отличий. Про любую из этих моделей можно сказать, что она *является (is-a)* амортизатором.

Для решения данной задачи как нельзя лучше подойдет наследование. Амортизаторы могут быть смоделированы в виде неглубокой узкой иерархии. Небольшой размер иерархии делает ее понятной (с ясными намерениями относительно ее создания) и легко расширяемой. Поскольку эти объекты отвечают критериям успешного использования наследования, риск ошибиться весьма невысок (но если вы все же *допустите* ошибку, затраты на пересмотр вашего подхода также невелики). У вас есть возможность получить выгоды от наследования, не подвергая себя особому риску.

В примере, приведенном в данной главе, каждый отдельно взятый амортизатор играет роль `Part`. Он наследует общее поведение амортизаторов *и* `Part`-роль у своего абстрактного родительского класса `Shock`. В отношении фабрики `PartsFactory` есть предположение, что каждая часть может быть представлена `OpenStruct`-объектом с ролью `Part`, но вы легко сможете расширить конфигурационный массив частей с целью указания имени класса для конкретного амортизатора. Поскольку вы уже думаете о `Part` как об интерфейсе, будет нетрудно подключить новую разновидность части, даже если эта часть для получения некоторых особенностей своего поведения использует наследование.

Если требования изменятся таким образом, что произойдет взрывной рост разновидностей амортизаторов, выбранное проектное решение придется пересмотреть. Может быть, оно по-прежнему будет подходящим, а может, нет. Если моделирование множества новых амортизаторов потребует весьма существенного расширения иерархии или же если новые амортизаторы не смогут вписаться в существующий код, то *в таком случае* следует рассмотреть альтернативные варианты.

Применение неявных типов для отношений вида «ведет себя как»

Некоторые задачи требуют, чтобы множество различных объектов играли общую роль. Вдобавок к основным обязанностям объекты могут играть роли, позволяющие им становиться пригодными для *включения в план*, *проведения над ними подготовительных работ*, *вывода их на печать* и *сохраняемости*.

Чтобы выявить существование роли, можно воспользоваться двумя способами. Во-первых, хотя объект и играет некую роль, она не является его основной

обязанностью. Велосипед *ведет себя как (behaves-like-a)* пригодный к включению в план, но при этом он *является (is-a)* велосипедом. Во-вторых, потребность довольно широко распространена, многие неродственные объекты имеют общее желание играть одну и ту же роль.

Показательнее всего обдумывать роли со стороны (с точки зрения владельца исполнителя роли, а не самого исполнителя). Владелец исполнителя роли, *пригодного к включению в план*, ожидает от исполнителя реализации интерфейса `Schedulable` и соблюдения контракта `Schedulable`. Все *пригодные к включению в план* схожи в том, что они должны отвечать этим ожиданиям.

Ваша проектировочная задача заключается в выявлении существования роли, определении интерфейса ее неявного типа и предоставлении реализации этого интерфейса каждому возможному исполнителю. Некоторые роли состоят только из их интерфейсов, другие же совместно используют общее поведение. Чтобы позволить объектам исполнять роль без дублирования кода, определяйте общее поведение в модуле Ruby.

Использование композиции для отношений в «обладает»

Многие объекты состоят из множества частей, но представляют собой нечто большее, чем сумма этих частей. Велосипеды (`Bicycle`) *обладают (have-a)* частями (`Part`), но сам велосипед представляет собой нечто большее. В дополнение к поведению его частей у велосипеда есть свое отдельное поведение. С учетом текущих требований к велосипеду наиболее экономически целесообразным способом моделирования объекта `Bicycle` является использование композиции.

Эта разница между *«является» (is-a)* и *«обладает» (has-a)* влияет на принятие решения о том, что именно выбрать — наследование или композицию. Чем больше частей у объекта, тем выше вероятность, что он должен быть смоделирован с использованием композиции. Чем глубже приходится внедряться в отдельные части, тем выше вероятность обнаружения особой части, имеющей несколько специализированных вариантов, которая вполне обоснованно станет кандидатом для применения наследования. При решении любой задачи следует оценить издержки и выгоды альтернативных технологий проектирования, а для наиболее разумного выбора — опираться на собственный опыт и здравый смысл.

Выводы

Композиции позволяют объединять небольшие фрагменты для создания более сложных объектов, причем так, чтобы целое представляло собой нечто большее, чем сумма его частей. Композиционные объекты состоят из простых отдельных компонентов, которые без особых затрат можно перестроить в новые комбинации. В простых объектах легче разобраться, они проще поддаются повторному использованию и тестированию, но, поскольку они объединяются в сложное целое, разобраться в работе более крупного приложения значительно труднее, чем в назначении его отдельных частей.

Композиции, классическое наследование и совместное использование поведения посредством модулей являются конкурирующими методами организации кода. У каждого из них свои издержки и выгоды (это и определяет их эффективность при решении отличающихся друг от друга задач).

Эти технологии — не что иное, как инструменты, и ваше мастерство проектировщика неизменно вырастет, если вы воспользуетесь каждой из них. Научиться правильно применять их можно только с опытом (а лучший способ набраться опыта — учиться на собственных ошибках). Применение этих технологий на практике, готовность признавать свои ошибки, независимость от прежних проектировочных решений и безжалостная реорганизация кода помогут вам повысить свое мастерство.

По мере накопления опыта вы все чаще станете с первого раза выбирать наилучшую технологию, затраты станут снижаться, а качество приложений улучшится.

Глава 9
Проектирование экономически эффективных тестов

Написание кода, легко поддающегося изменениям, — это искусство, освоить которое помогут три навыка.

Во-первых, вам следует изучить объектно-ориентированное проектирование. Небрежно спроектированный код, конечно же, трудно поддается изменениям. С практической точки зрения единственное, что действительно дает возможность беспрепятственного внесения изменений, — это конструкция приложения, ведь легко поддается изменениям только грамотно спроектированный код. Поскольку вы дочитали книгу до этой главы, вам не остается ничего иного, как предположить, что настойчивость обязательно окупится и вы приобретете базовые знания, которые позволят приступить к проектированию легко изменяемого кода.

Во-вторых, вам нужны навыки реорганизации кода. Определение реорганизации дал Мартин Фаулер (Martin Fowler) в книге Refactoring: Improving the Design of Existing Code: «Реорганизация представляет собой процесс такого изменения программных систем, при котором совершенствование внутренней структуры не приводит к изменению внешних проявлений поведения кода».

Обратите внимание на фразу *«не приводит к изменению внешних проявлений поведения кода»*. Реорганизация не добавляет нового поведения, а улучшает существующую структуру. Это весьма тонкий процесс внесения в код изменений небольшими, четко выверенными порциями, которые постепенно преобразуют одну конструкцию в другую без появления ошибок.

В хорошо продуманной конструкции сохраняется максимум гибкости при минимуме затрат (за счет того, что принятие решений откладывается при любой малейшей возможности до поступления более конкретных требований). Когда же настанет день X, придется заняться *реорганизацией*, то есть преобразованием структуры кода с целью ее адаптации под новые требования. Новые функциональные средства будут добавлены только после успешной реорганизации кода.

Слабые навыки в реорганизации нужно совершенствовать. Результатом качественного проектирования является необходимость в постоянной реорганизации, а усилия, затраченные на проектирование, полностью окупятся легкостью проведения реорганизации.

И наконец, искусство написания легко изменяемого кода требует навыков написания качественных тестов. Тесты дают уверенность в успешном проведении постоянной реорганизации. Эффективные тесты доказывают, что код, который подвергся изменениям, продолжает вести себя правильно и не требует повышения затрат на свое сопровождение. Качественные тесты уверенно справляются с реорганизацией кода; они создаются таким образом, чтобы изменения кода не вынуждали к переписыванию тестов.

Написание именно таких тестов относится к вопросу проектирования и является темой данной главы.

Освоение объектно-ориентированного проектирования, наличие навыков реорганизации кода, а также способность разрабатывать эффективные тесты закладывают основу для создания легко изменяемого кода. Качественно спроектированный код легко поддается изменениям, реорганизация за счет изменений помогает совершенствовать конструкцию, а тесты позволяют проводить реорганизацию, не опасаясь негативных последствий.

Целенаправленное тестирование

Наиболее часто необходимость использования тестов объясняется тем, что они снижают количество ошибок и предоставляют документацию и что написание *в первую очередь* тестов улучшает конструкцию приложения.

Конечно, все эти преимущества несомненны, но они лишь посредники в достижении более важной цели. Реальной целью тестирования (как и проектирования) является снижение затрат. Если на написание, сопровождение и запуск тестов уходит больше времени, чем потребовалось бы на устранение ошибок,

написание документации и проектирование приложений без применения тестов, то создавать тесты не имеет смысла.

Зачастую новички в вопросах тестирования расстраиваются, когда написанные ими тесты *действительно* не окупаются приносимой ими пользой. У программистов, уверовавших в высокую продуктивность своих прежних приложений без тестов, возникает даже желание оспорить ценность тестов, но на деле они просто застопорились. Их попытки написания тестов не увенчались успехом, а желание наверстать упущенное заставляет их вернуться к старым привычкам и отказаться от тестов.

Однако решение проблемы дорогостоящих тестов не в прекращении тестирования, а в совершенствовании навыков их написания. Чтобы тесты приносили пользу, требуются ясные намерения и четкое осознание, что, когда и как тестировать.

Осознание намерений

У тестирования множество потенциальных выгод (часть их очевидна, другая часть — менее заметна). Четкое понимание этих выгод повысит вашу рабочую мотивацию.

Поиск ошибок

Обнаружение просчетов и ошибок на ранних стадиях разработки может принести немалую выгоду. Ошибку проще исправить сразу после ее возникновения. Полученные как можно раньше сведения о возможности (или невозможности) что-либо сделать помогут выбрать альтернативный вариант, который внесет изменения в варианты проектирования, доступные в будущем. Кроме того, по мере накопления кода возникают зависимости от закравшихся ошибок. Их исправление на более поздних стадиях разработки может повлечь необходимость изменения большого объема кода. Чем раньше устранить ошибки, тем меньше будут затраты.

Предоставление документации

Тесты предоставляют самую достоверную документацию по проекту. То, о чем в ней рассказывается, сохраняет актуальность после того, как уже устареет бумажная документация и нужная информация выветрится из памяти. Тесты следует создавать в расчете на собственную амнезию в будущем. Помните о том, что все забывается, поэтому пишите такие тесты, которые смогут напомнить вам всю историю приложения.

Отсрочка проектных решений

Тесты позволяют без особых опасений отложить проектные решения. По мере совершенствования своего мастерства проектировщика вы станете создавать приложения, заполненные участками, требующими *каких-либо* проектных изменений, но пока не будет накоплено достаточно информации, вы не будете знать, каких именно.

Вы будете ждать дополнительную информацию, мужественно сопротивляясь тем силам, которые принуждают вас окончательно принять за основу конкретную конструкцию. Эти «подвисшие» решения зачастую программируются в виде откровенных трюков, скрытых за абсолютно приличными интерфейсами. Подобная ситуация возникает, когда в настоящее время известен только один конкретный случай, но вы полны ожиданий, что в ближайшем будущем сложатся новые обстоятельства. Вам известно, что настанет момент, когда лучше будет довольствоваться кодом, справляющимся с множеством конкретных случаев и представляющим собой единую абстракцию, но именно сейчас у вас дефицит информации, что не позволяет предвидеть, какой именно должна быть эта абстракция.

Когда ваши тесты зависят от интерфейсов, реорганизацию исходного кода можно проводить, не думая о последствиях для самих тестов. Тесты проверяют факт того, что интерфейс продолжает вести себя подобающим образом и что изменения исходного кода не требуют переписывания тестов. Преднамеренно закладываемая зависимость от интерфейсов позволяет вам использовать тесты для безопасной отсрочки принятия проектных решений без каких-либо негативных последствий.

Поддержка абстракций

Когда наконец-то поступит дополнительная информация и вы примете очередное проектное решение, в код будут внесены изменения, повышающие уровень абстракции. В этом и заключается еще одно преимущество тестов в отношении проектных решений.

При качественном проектировании происходит естественное продвижение в направлении создания небольших независимых объектов, зависящих от абстракций. Поведение грамотно спроектированных приложений постепенно становится результатом взаимодействия между этими абстракциями. Абстракции являются удивительно гибкими конструктивными элементами, но за вносимые ими усовершенствования приходится платить: наряду с тем, что в отдельно взятой абстракции можно довольно легко разобраться, в коде нет какого-либо одного места, позволяющего прояснить их поведение в целом.

По мере расширения кодовой базы и возрастания количества абстракций востребованность тестов возрастает. Существует такой уровень проектной абстракции, на котором практически невозможно безопасное внесение любого изменения, пока код не будет протестирован. Тесты являются вашими записями интерфейса каждой абстракции и служат вашей опорой на пути продвижения разработки. Они позволяют откладывать проектные решения и создавать абстракции на любую полезную глубину.

Выявление конструктивных недостатков

Следующая выгода от применения тестов — выявление с их помощью конструктивных недостатков в исходном коде. Если тест требует мучительных настроек, значит, код ожидает слишком большого объема контекста. Если тестирование одного объекта повлечет за собой тестирование ряда других объектов, значит, у кода слишком много зависимостей. Если есть трудности с написанием теста, значит, код будет нелегко повторно использовать с другими объектами.

Когда конструкция спроектирована из рук вон плохо, тестирование дается с трудом. Однако нет никаких гарантий, что верно обратное утверждение. Тесты, на разработку которых ушло много ресурсов, еще не признак плохо спроектированного кода. И никто не отменял написания плохих тестов для грамотно спроектированного кода. Следовательно, чтобы проводить тестирование с целью снижения затрат, требуются и грамотно спроектированное приложение, и качественно спроектированные тесты.

Ваша задача — извлечь всю выгоду от тестирования при наименьших затратах на разработку тестов. Лучший способ достижения цели — написание слабосвязанных тестов только для тех компонентов программы, которые играют важную роль.

Выявление предмета тестирования

Большинство программистов пишут слишком много тестов. Данный факт не всегда на поверхности, поскольку зачастую затраты на эти ненужные тесты настолько велики, что программисты вообще отказываются от тестирования. Это не означает, что в их распоряжении нет тестов. У них имеется весьма обширный, но устаревший набор тестов, которые практически никогда не запускались. Существует один простой способ повышения ценности тестов, который заключается в написании наименьшего (самого необходимого) их количества. Проще всего эту цель достичь однократным тестированием самых нужных мест.

Удаление из тестов повторяющегося кода снижает расходы на изменение тестов в ответ на изменения, вносимые в приложение, а правильный выбор мест для тестирования гарантирует изменение тестов только в случае крайней необходимости. Избавление тестов от всего, что не составляет их истинную суть, требует абсолютно четкого понимания предметов тестирования (на помощь приходят уже известные вам принципы проектирования).

Представьте объектно-ориентированное приложение в виде серии сообщений, передаваемых между набором «черных ящиков». Представление каждого объекта в качестве «черного ящика» ограничивает разрешенные другим знания об объекте и лимитирует открытые знания о любом объекте сообщениями, пересекающими его границы.

У качественно спроектированных объектов весьма крепкие границы. Каждый такой объект похож на спускаемый космический аппарат, показанный на рис. 9.1. Никто снаружи не может заглянуть внутрь, никто изнутри не может выглянуть наружу, и только несколько строго оговоренных сообщений могут пройти сквозь предопределенные переходные шлюзы.

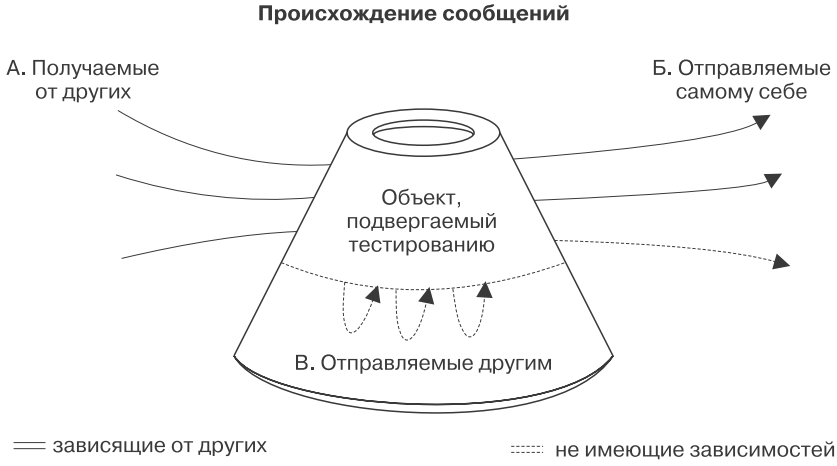

Рис. 9.1. Объекты, подвергаемые тестированию, подобны спускаемому космическому аппарату: сообщения разбиваются об их границы

Это преднамеренное незнание, что внутри любого другого объекта, положено в основу конструкции. Понимание сути объектов исключительно по сообщениям, на которые они способны реагировать, позволяет разрабатывать изменяемые приложения, понимание важности данного взгляда на объекты позволит создавать тесты, предоставляющие максимум выгод при минимуме затрат.

Принципы проектирования, соблюдаемые вами при разработке приложения, применимы и к вашим тестам. Каждый тест является практически еще одним объектом приложения, нуждающимся в использовании существующего класса. Чем больше тест привязывается к этому классу, тем больше они переплетаются и тем более тест уязвим перед вынужденными нежелательными изменениями.

Нужно не только ограничивать количество связей, но и разрешать только те немногочисленные связи, которые относятся к стабильным компонентам объекта. Наиболее стабильный компонент любого объекта — его открытый интерфейс; из этого можно сделать логический вывод, что создаваемые тесты должны быть предназначены для работы с сообщениями, определенными в открытых интерфейсах. Наиболее затратными и наименее эффективными тестами являются те, которые пробивают брешь в защитной оболочке объекта путем связывания с его нестабильными внутренними компонентами. Эти излишне ретивые тесты ни в коей мере не служат доказательством общей безошибочности приложения, но при этом увеличивают общие расходы, поскольку приходят в негодность после каждой реорганизации базового класса.

Тесты должны быть сконцентрированы на входящих и исходящих сообщениях, пересекающих границы объектов. Входящие сообщения формируют открытый интерфейс получающего их объекта. Исходящие сообщения по определению являются входящими для других объектов, являясь, таким образом, как показано на рис. 9.2, частью интерфейса других объектов.

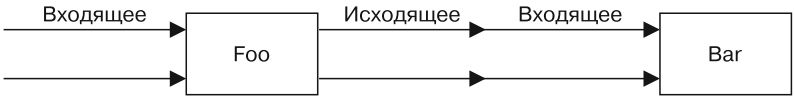

Рис. 9.2. Исходящее сообщение одного объекта становится входящим сообщением другого объекта

На рис. 9.2 сообщения, являющиеся входящими для `Foo`, формируют открытый интерфейс объекта `Foo`. Этот объект несет ответственность за тестирование своего собственного интерфейса и проводит его путем утверждений, касающихся результатов, возвращаемых этими сообщениями. Тесты, содержащие утверждения о возвращаемых этими сообщениями значениях, относятся к тестам *состояния*. Обычно в таких тестах содержатся утверждения о том, что результаты, возвращаемые сообщением, равны ожидаемому значению.

На рис. 9.2 также показано, что `Foo` отправляет сообщения объекту `Bar`. Сообщение, отправляемое объектом `Foo` объекту `Bar`, является исходящим из `Foo`

и входящим в `Bar`. Это сообщение является частью открытого интерфейса объекта `Bar`, поэтому все тесты состояния должны ограничиваться объектом `Bar`. Объекту `Foo` не нужно (и от него не требуется) тестировать эти исходящие сообщения на состояние. Общее правило таково: объекты должны формировать утверждения о состоянии *только* для сообщений в своих собственных открытых интерфейсах. Если следовать этому правилу, то тесты возвращаемых значений проводятся лишь в одном месте, при этом удаляются все ненужные дубликаты — и ваши тесты вполне вписываются в DRY-принцип, призывающий не повторяться. В итоге снижаются затраты на их сопровождение.

Отсутствие потребности в тестировании состояния на основе исходящих сообщений не означает, что исходящие сообщения вообще не нужно тестировать. Существуют две разновидности исходящих сообщений, и для одной из них требуется тест иного рода.

У некоторых исходящих сообщений нет никаких побочных эффектов, они интересуют только своих отправителей. Отправителя, конечно же, заботит получаемый в ответ результат (зачем же еще отправлять сообщение?), но для других частей приложения отправленное сообщение не представляет никакого интереса. Подобные исходящие сообщения называются *запросами* и не нуждаются в тестировании отправляющим объектом. Сообщения-запросы являются частью открытого интерфейса своих получателей, которые уже реализуют все необходимое тестирование состояния.

Однако у многих исходящих сообщений *имеются* побочные эффекты (файлы записываются, записи баз данных сохраняются, наблюдатель предпринимает какое-либо действие), от которых зависит работа вашего приложения. Такие сообщения являются *командами*, и в обязанности отправляющему их объекту вменяется подтверждение надлежащей отправки. Подтверждение отправки сообщения тестирует поведение, а не состояние, используя информацию о том, сколько раз и с какими аргументами отправлено сообщение.

Исходя из этого, можно дать следующие рекомендации о предмете тестирования. Входящие сообщения должны тестироваться в отношении возвращаемого состояния. Исходящие сообщения-команды должны тестироваться, чтобы убедиться, что они были отправлены. Исходящие сообщения-запросы тестироваться не должны.

При условии, что объекты вашего приложения общаются друг с другом исключительно с помощью открытых интерфейсов, вашим тестам не нужно знать ни о чем другом. Когда тестируется этот минимальный набор сообщений, на

любой тест не могут повлиять никакие изменения в закрытом поведении любого объекта.

Когда исходящие сообщения-команды тестируются с целью убедиться, что они отправлены, слабосвязанные тесты могут выдержать изменения, вносимые в приложение без принуждения к изменению их самих. При условии, что открытые интерфейсы остаются стабильными, первоначально созданные тесты будут работать, никогда не вызывая у вас никаких опасений на свой счет.

Умение определять нужный момент для тестирования

Как только появится смысл в том, чтобы создавать сначала тесты, а потом сам код, именно так и следует поступать. К сожалению, принятие решения о том, когда именно появится этот смысл, может стать для начинающих проектировщиков весьма нелегкой задачей, и данный ранее совет вряд ли принесет какую-либо пользу. Новички зачастую создают излишне связанный код, они склонны к объединению не связанных друг с другом обязанностей и привязывают к каждому объекту множество зависимостей. Создаваемые ими приложения выглядят как запутанный код, в котором вообще не встречаются объекты, существующие в изоляции. Провести после этого тестирование таких приложений невероятно трудно, поскольку *тесты предполагают повторное использование*, а повторно использовать такой код невозможно.

Изначально для написания тестов необходимо наличие хотя бы малейшей возможности повторного использования кода для встраивания в объект с момента его появления; в противном случае тесты вообще не смогут быть написаны. Поэтому начинающим проектировщикам нужно стремиться к написанию кода, который с самого начала приспособлен для тестирования. Сделать это при отсутствии навыков проектирования крайне трудно, но если проявить упорство, то получится хотя бы тестируемый код, иначе вообще ничего не выйдет.

Однако следует иметь в виду, что первоочередное создание тестов не является заменой и гарантией качественно спроектированного приложения. Получаемая в результате первоочередного написания тестов возможность повторного использования кода, конечно, лучше, чем вообще ничего, но при этом конструкция приложения все равно может оказаться далекой от совершенства. Новички из лучших побуждений зачастую пишут дорогостоящие в сопровождении и содержащие множество повторений тесты для запутанного, имеющего

сильные связи кода. Как это ни печально, но наиболее сложный код очень часто создается наименее квалифицированными специалистами. Он отнюдь не отражает истинную сложность решаемой задачи, а свидетельствует об отсутствии опыта у программиста. Для написания простого кода начинающим программистам просто не хватает навыков.

Созданные новичками слишком сложные приложения — это скорее результат их настойчивости, чудо уже то, что они вообще работают. Код очень *тяжелый*, приложение трудно поддается изменениям, и все тесты при каждой реорганизации приходят в негодность. Высокие затраты на изменения могут снизить производительность, что обескуражит все заинтересованные стороны. Каскад изменений по всему приложению и затраты на сопровождение тестов создают впечатление, что тесты стоят дороже их реальной цены.

Если вы как начинающий разработчик оказались в подобной ситуации, не стоит терять веру в ценность тестирования. Тестирование и написание кода с начальным прицелом на тестирование нужно проводить своевременно и в разумных объемах, что позволит снизить ваши общие затраты. Получение подобных выгод требует повсеместного применения принципов объектно-ориентированного проектирования — как в коде вашего приложения, *так и* в коде ваших тестов. Приобретенные знания в деле проектирования уже упрощают написание тестируемого кода, а тому, как эти принципы проектирования применяются при создании тестов, посвящена основная часть остального материала данной главы. Поскольку грамотно спроектированные приложения легко поддаются изменениям, а качественно спроектированные тесты запросто могут вообще избежать изменений, совокупность этих конструктивных усовершенствований окупится сторицей.

Опытные проектировщики получают почти незаметные конструктивные улучшения от первых тестов. Но это не значит, что они не могут извлечь из соблюдения данного принципа какую-то пользу или что они никогда при этом не обнаруживают что-либо неожиданное (кроме выгоды, извлекаемой из уже имеющейся навязанной возможности повторного использования кода). Благодаря этому программисты создают слабо связанный, повторно используемый код, а тесты приносят пользу иными путями.

Опытным проектировщикам известен прием контрольной пробы, то есть экспериментального поиска решения задачи на этапе написания кода. Эти эксперименты носят исследовательский характер и касаются задач, для которых еще не найдено конкретное решение. Как только ситуация прояснится и конструкция сама подскажет нужное решение, программисты тут же вернутся к созданию кода с первоочередным написанием тестов.

Ваша конечная цель заключается в создании качественно спроектированных приложений, имеющих вполне приемлемый охват тестами. Наилучший способ достижения этой цели варьируется в зависимости от того, насколько вы опытны.

Умение действовать по собственному усмотрению не следует рассматривать как разрешение на отмену тестирования. Плохо спроектированным кодом, не имеющим средств для тестирования, может быть только устаревший код, вообще не поддающийся тестированию. Не нужно переоценивать свои возможности и использовать в качестве предлога для отказа от тестирования раздутое самомнение.

Умение проводить тестирование

Создать новую среду тестирования на Ruby может любой пользователь, и порой складывается мнение, что это доступно каждому. Но новая, более удачная среда может содержать функции, без которых не обойтись. И если вы способны здраво оценивать все выгоды и издержки, можете свободно выбирать любую подходящую вам среду.

Однако есть немало веских аргументов в пользу того, чтобы оставаться в пределах основного направления тестирования. Наиболее востребованные среды обладают лучшей поддержкой. Они быстро обновляются, чтобы не утрачивать гарантию поддержки совместимости с новыми выпусками Ruby (и Rails) и чтобы не возникало препятствий при поддержке текущей версии. Их обширный контингент пользователей заставляет сохранять и обратную совместимость. Вряд ли их станут обновлять таким образом, чтобы заставить вас переписывать все ваши тесты. И поскольку они нашли довольно широкое применение, нетрудно найти программистов, имеющих опыт их использования.

На момент написания данной книги наиболее востребованными средами считались MiniTest от Райана Дэвиса (Ryan Davis) и `seattle.rb` в комплекте с Ruby версии 1.9, RSpec от Дэвида Челимски (David Chelimsky) и команды RSpec. Философии этих сред отличаются друг от друга, но к какой из них вы бы ни склонялись, обе они вполне достойный выбор.

Правда, одним только выбором среды дело не ограничивается, вам следует также выбрать один из альтернативных стилей тестирования: разработка посредством тестирования Test Driven Development (TDD) или разработка через реализацию поведения Behavior Driven Development (BDD). Решение может быть непростым. TDD и BDD могут показаться диаметрально противоположными альтернативами, но лучше всего их можно рассмотреть в виде сплошной

среды, подобной той, что показана на рис. 9.3, где выбор диктуется вашей оценкой и опытом.

Оба стиля предполагают создание кода за счет первоочередного написания тестов. В BDD используется подход снаружи вовнутрь с созданием объектов на границе приложения и проработкой их прохождения вовнутрь, а также с созданием при необходимости заглушек для предоставления пока еще не созданных объектов. В TDD используется подход изнутри наружу, обычно осуществляемый путем тестирования объектов предметной области с последующим повторным использованием только что созданных объектов предметной области в тестах смежных уровней кода.

Предпочтение какому-либо из стилей может быть отдано на основе накопленного опыта или склонностей, но оба они приемлемы. У каждого свои достоинства и недостатки, некоторые из них рассмотрены в следующих разделах при написании тестов.

В процессе тестирования полезно представлять объекты приложения разделенными на две основные категории. Первая из них содержит *тестируемый объект*, который далее так и будет называться. Вторая категория содержит все остальное.

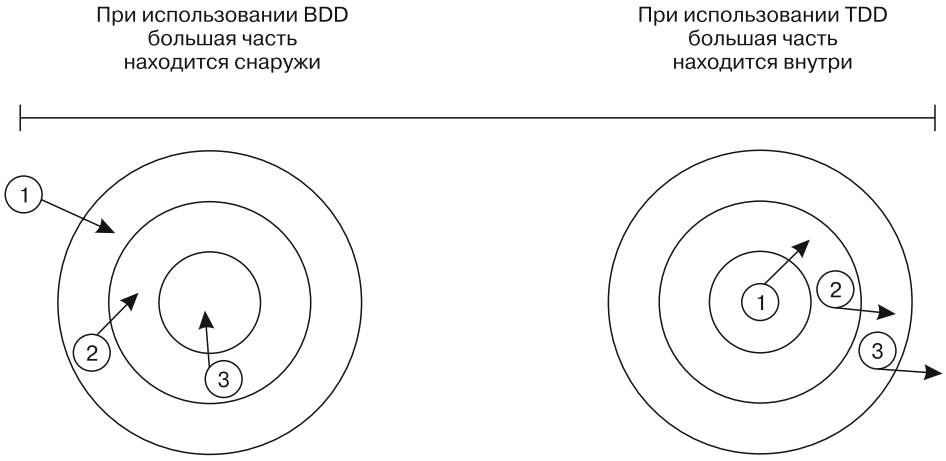

Рис. 9.3. BDD и TDD должны рассматриваться в виде сплошной среды

Вашим тестам должно быть точно известно все о первой категории, то есть о тестируемых объектах, а вот при использовании второй категории они должны игнорировать все, что только можно. Следует делать вид, что все остальное

приложение непрозрачно; единственной доступной в ходе тестирования информацией будет та, которую можно собрать, рассматривая тестируемый объект.

После наведения фокуса тестирования на тестируемый объект нужно выбрать точку, откуда проводится тестирование. Ваши тесты *могут* полностью находиться внутри тестируемого объекта (с полным доступом ко всем его внутренним частям). Но это неудачная идея, поскольку при этом наблюдается утечка знаний (которые должны быть закрыты в объекте) в тесты, повышающая степень связанности между ними и увеличивающая вероятность того, что изменения, вносимые в код, потребуют внесения изменений в тесты. Лучше задать точку, откуда проводится тестирование, на границах тестируемого объекта, где будет известно только о входящих и исходящих сообщениях.

СРЕДА MINITEST

Тесты в этой главе написаны с использованием среды MiniTest. Это не значит, что мы предпочитаем именно это среду; примеры, написанные в MiniTest, работают где угодно (в Ruby 1.9 или в более поздних версиях языка). Вы можете продублировать примеры и проводить с ними эксперименты, не устанавливая никакого дополнительного программного обеспечения.

За время чтения этой главы среда MiniTest могла уже измениться. Улучшения в это приложение может внести кто угодно, предоставив вам полное право распоряжаться ими совершенно свободно (таков уж удел разработчика программы с открытым кодом). Но независимо от того, как может быть усовершенствована среда MiniTest, показанные ниже принципы останутся в силе. Не отвлекайтесь на изменения в синтаксисе, сконцентрируйтесь на понимании основных целей тестирования. Так вы сможете достичь их с помощью любой среды тестирования.

Тестирование входящих сообщений

Входящие сообщения составляют открытый интерфейс объекта. Эти сообщения нужно тестировать, поскольку от их сигнатуры и возвращаемых ими результатов зависят другие объекты приложения.

В этих первых тестах используется код из примеров главы 3 «Управление зависимостями». Ниже приводится напоминание о классах `Wheel` и `Gear` в том виде, в котором они находились, когда были перепутаны. `Gear` создает экземпляр класса `Wheel` глубоко внутри своего метода `gear_inches`, в строке 24.

ПРИМЕЧАНИЕ

В оставшейся части главы содержатся тесты для уже приводившегося в книге кода. Ранее эти примеры кода позволяли объяснить принципы объектно-ориентированного проектирования, а здесь они будут использоваться, чтобы проиллюстрировать способы тестирования различных компонентов конструкции. Следующие тесты не затрагивают каждую строку увиденного вами кода, с их помощью проводится тестирование каждого понятия, изученного в данной книге.

```
class Wheel
  attr_reader :rim, :tire
  def initialize(rim, tire)
     @rim   = rim
     @tire  = tire
  end

  def diameter
     rim + (tire * 2)
  end
  # ...
end

class Gear
  attr_reader :chainring, :cog, :rim, :tire
  def initialize(args)
    @chainring = args[:chainring]
    @cog       = args[:cog]
    @rim       = args[:rim]
    @tire      = args[:tire]
  end

  def gear_inches
    ratio * Wheel.new(rim, tire).diameter
  end

  def ratio
    chainring / cog.to_f
  end
  # ...
end
```

В таблице 9.1 показаны сообщения (кроме тех, что возвращают простые атрибуты), пересекающие границы этих объектов. Класс `Wheel` отвечает за одно входящее сообщение `diameter` (отправляется классом `Gear`, являясь для него исходящим сообщением), а класс `Gear` отвечает за два входящих сообщения — `gear_inches` и `ratio`.

Таблица 9.1. Входящие и исходящие сообщения объекта

Объект	Входящие сообщения	Исходящие сообщения	Имеется ли кто-нибудь зависимый?
Wheel	diameter		Да
Gear		diameter	Нет
	gear_inches		Да
	ratio		Да

В начале этого раздела утверждалось, что каждое входящее сообщение является частью открытого интерфейса объекта, поэтому оно должно быть протестировано. Теперь настало время высказать небольшое предостережение.

Удаление неиспользуемых интерфейсов

У входящих сообщений должен быть тот, кто от них зависит. Из данных табл. 9.1 видно, что это утверждение соблюдается для `diameter`, `gear_inches` и `ratio`, поскольку все они — входящие сообщения. От каждого из этих сообщений зависят некоторые объекты, *не являющиеся их исходными реализаторами*.

Если составить подобную таблицу для тестируемого объекта и найти предполагаемое входящее сообщение, от которого никто не зависит, на него нужно смотреть с особой подозрительностью. Какой цели служит реализация сообщения, которое никем не отправляется? Ведь на самом деле оно вообще не является *входящим*, это надуманная реализация, похожая на догадки о будущих (следовательно, пока что не существующих) требованиях.

Тестировать входящие сообщения, от которых никто не зависит, не имеет никакого смысла, их нужно удалить. Если активно удалять неиспользуемый код, то ваше приложение от этого только выиграет. На такой код зря тратятся средства, он отвлекает силы на тестирование и сопровождение, не принося при этом никакой пользы. Удаление неиспользуемого кода приводит к экономии средств, поскольку если этого не сделать, то код придется тестировать.

Пересильте себя, если нужно; подобное удаление докажет свою ценность. До тех пор пока у вас не сложится твердая убежденность в правильности выбранной стратегии, можете утешаться возможностью восстановления удаленного кода с помощью системы управления версиями. Независимо от того, насколько трудно или легко вы расстаетесь с кодом, его все-таки нужно удалить. Неиспользуемый код дороже оставить, нежели восстановить.

Проверка открытого интерфейса

Входящие сообщения тестируются путем проверки утверждения, касающегося значения или состояния, возвращаемого после их инициирования. Первым требованием к тестированию входящего сообщения является проверка факта возвращения ими надлежащего значения в любой возможной ситуации.

В следующем фрагменте кода показан тест метода `diameter`, определенного в классе `Wheel`. В строке 4 создается экземпляр класса `Wheel`, а в строке 6 утверждается, что у этого `Wheel`-объекта (колеса) имеется `diameter` (диаметр) со значением 29.

```
1 class WheelTest < MiniTest::Unit::TestCase
2
3    def test_calculates_diameter
4      wheel = Wheel.new(26, 1.5)
5
6      assert_in_delta(29,
7                      wheel.diameter,
8                      0.01)
9    end
10 end
```

Этот тест предельно прост и активизируется с помощью небольшого фрагмента кода. У `Wheel` нет скрытых зависимостей, поэтому никакие другие объекты приложения в качестве побочных эффектов при выполнении этого теста не создаются. Конструкция класса `Wheel` позволяет проводить его тестирование независимо от любого другого класса приложения.

Тестирование класса `Gear` проходит интереснее. `Gear` требует больше аргументов, чем `Wheel`, но все равно общая структура этих двух тестов очень похожа. В показанном ниже тесте `gear_inches` в строке 4 создается новый экземпляр `Gear`, а в строке 10 проверяется утверждение о результатах вызова метода.

```
class GearTest < MiniTest::Unit::TestCase

  def test_calculates_gear_inches
    gear = Gear.new(
               chainring: 52,
               cog:       11,
               rim:       26,
               tire:      1.5 )

    assert_in_delta(137.1,
                    gear.gear_inches,
                    0.01)
  end
end
```

Новый тест `gear_inches` во многом похож на тест метода `diameter`, определенного в классе `Wheel`, но внешний вид обманчив. В этом тесте есть усложнения, отсутствующие в тесте `diameter`. Имеющаяся в `Gear` реализация `gear_inches`, конечно же, создает и использует другой объект, — экземпляр класса `Wheel`. Классы `Gear` и `Wheel` связаны кодом и в тестах, хотя здесь это не проявляется так очевидно.

Тот факт, что реализованный в `Gear` метод `gear_inches` создает и использует еще один объект, влияет на продолжительность теста и степень вероятности того, что он пострадает от непредсказуемых последствий в виде изменений в не связанной с ним части приложения. Но связанность, создающая эту проблему, скрыта внутри класса `Gear` и поэтому совершенно невидима в этом тесте. Тест предназначен для проверки того, что `gear_inches` возвращает приемлемый результат, и он целиком и полностью отвечает этому требованию, но способ структуризации базового кода добавляет скрытый риск.

Если создание `Wheel` обойдется дорого, то тест `Gear` заплатит эту цену даже притом, что у него нет никакого интереса к `Wheel`. Если с `Gear` будет все в порядке, а `Wheel` перестанет работать, тест `Gear` может провалиться по совершенно неявной причине в месте, находящемся довольно далеко от тестируемого кода.

Быстрее всего тесты работают, когда в них выполняется меньше кода (объем вовлекаемого в тест внешнего кода напрямую зависит от вашей конструкции). Приложение, построенное из тесно связанных, перегруженных зависимостями объектов, похоже на сотканный вручную ковер, в котором за одной вытянутой нитью тянутся все остальные. При тестировании тесно связанных объектов тест,

проводимый в отношении одного объекта, запускает код во многих других объектах. Если код, подобный коду класса `Wheel`, также связан с другими объектами, проблема усугубится; тогда запуск теста класса `Gear` приведет к созданию огромной сети объектов, любой из которых может привести к отказу по странной причине.

Такое в тестах случается и чем-то из ряда вон выходящим не считается. Поскольку тесты в первую очередь повторно используют код, эта проблема является предвестником грядущих неурядиц для всего приложения.

Изоляция тестируемого объекта

Экземпляр класса `Gear` является довольно простым объектом, но попытка протестировать его метод `gear_inches` выявляет скрытые сложности. Цель этого теста — убедиться в правильном вычислении передаточного отношения в дюймах (gear inches), но оказывается, что запуск `gear_inches` зависит от кода, находящегося в объекте, который не относится к `Gear`.

Это весьма распространенная проблема проектирования; когда протестировать `Gear` в изоляции невозможно, это не сулит ничего хорошего в будущем. Сложности в изоляции `Gear` для тестирования показывают, что его код привязан к конкретному контексту, который накладывает ограничения, мешающие повторному использованию.

В главе 3 эта связь была разорвана путем удаления создания `Wheel`-объекта из кода `Gear`. Далее показан код, в котором сделано такое преобразование. Теперь `Gear` ожидает внедрения в себя объекта, понимающего сообщение `diameter`.

```
class Gear
  attr_reader :chainring, :cog, :wheel
  def initialize(args)
     @chainring = args[:chainring]
     @cog = args[:cog]
     @wheel = args[:wheel]
  end

  def gear_inches
      # Объект в переменной 'wheel'
      # играет роль 'способного понимать сообщение diameter'.
    ratio * wheel.diameter
  end

```

```
15   def ratio
16     chainring / cog.to_f
17   end
18 # ...
19 end
```

Такое преобразование кода происходит параллельно с преобразованием замысла. `Gear` теперь не интересует класс внедренного объекта, ожидается лишь, что в нем реализуется метод `diameter`. Этот метод является частью открытого интерфейса *роли*, которую с полным основанием можно назвать `Diameterizable`, то есть ролью объекта, способного понимать сообщение `diameter`.

Теперь, когда `Gear` разобщен с `Wheel`, экземпляр `Diameterizable` следует внедрять при каждом создании `Gear`-объекта. Но, поскольку `Wheel` является единственным классом приложения, играющим эту роль, варианты, доступные в процессе выполнения кода, строго ограничены. В реальной жизни (раз код в данный момент уже существует) каждый создаваемый `Gear`-объект не может избавиться от необходимости внедрения в него экземпляра класса `Wheel`.

Внедрение `Wheel` в `Gear` — это не то же самое, что внедрение `Diameterizable`. Правда, код приложения выглядит точно так же, но его логическое значение отличается. Отличие не в набираемых вами символах, а в вашем представлении о том, что именно они означают. Мысленное освобождение от привязанности к классу входящего объекта позволяет открыть конструкцию и ранее недоступные возможности *тестирования*. Представление о внедряемом объекте как об экземпляре его роли дает вам богатый выбор разновидностей `Diameterizable`-объектов для внедрения в `Gear` в ходе выполнения тестов.

Одним из возможных `Diameterizable`-объектов является, конечно же, `Wheel`-объект, поскольку в нем абсолютно четко реализован надлежащий интерфейс. В следующем примере делается именно этот весьма прозаичный выбор; в нем обновляется существующий тест, который приспосабливается к изменениям кода путем внедрения экземпляра класса `Wheel` (строка 6) в ходе тестирования.

```
1  class GearTest < MiniTest::Unit::TestCase
2    def test_calculates_gear_inches
3      gear = Gear.new(
4                chainring: 52,
5                cog:       11,
6                wheel:     Wheel.new(26, 1.5))
7
8      assert_in_delta(137.1,
```

```
 9                      gear.gear_inches,
10                     0.01)
11     end
12   end
```

Использование `Wheel`-объекта для внедрения `Diameterizable`-объекта приводит к коду теста, являющемуся зеркальным отражением приложения. Теперь становится вполне очевидным и в приложении, и в тестах, что `Gear` использует `Wheel`. Таким образом открыто продемонстрировано то невидимое связывание, которое существует между этими двумя классами.

Тест работает достаточно быстро, но добиться такой скорости удалось совершенно случайно. Это произошло не потому, что тест `gear_inches` был тщательно изолирован и отвязан от другого кода, а потому, что весь код, связанный с этим тестом, сам по себе выполняется довольно быстро.

Заметьте также, что здесь (или где-либо еще в подобной ситуации) не вполне очевидно, что `Wheel`-объект играет роль `Diameterizable`-объекта. Эта роль носит виртуальный характер, поскольку целиком находится лишь в вашем представлении. Нет даже признаков кода, который мог бы стать руководством для тех, кто будет сопровождать приложение в будущем, позволяющим думать о `Wheel`-объекте как об относящемся к роли `Diameterizable`.

Однако, несмотря на невидимость роли и связанность с `Wheel`, у такой структуризации теста имеется одно вполне реальное преимущество, показанное в следующем разделе.

Внедрение зависимостей с использованием классов

Когда код в ваших тестах использует такие же сотрудничающие объекты, как и ваше приложение, неудачи при прохождении тестов происходят именно тогда, когда они и должны произойти. Значение этого нельзя недооценивать.

Рассмотрим один простой пример. Представьте, что открытый интерфейс `Diameterizable`-объекта изменился. Какой-нибудь другой программист зашел в код класса `Wheel` и заменил, как показано в строке 8, имя метода `diameter` на `width`.

```
1 class Wheel
2    attr_reader :rim, :tire
3    def initialize(rim, tire)
```

```
    @rim  = rim
    @tire = tire
  end

  def width # <— раньше назывался 'diameter'
    rim + (tire * 2)
  end
# ...
end
```

Представим далее, что этот программист по ошибке не обновил имя отправляемого в `Gear`-объекте сообщения. `Gear` по-прежнему в своем методе `gear_inches` отправляет `diameter`, о чем свидетельствует приводимый в качестве напоминания текущий код класса `Gear`.

```
class Gear
  # ...
  def gear_inches
    ratio * wheel.diameter # <-- устаревшее сообщение
  end
end
```

Поскольку тест `Gear` внедряет экземпляр класса `Wheel`, `Wheel` реализует `width`, а `Gear` отправляет `diameter`, то прохождение теста проваливается.

```
Gear
  ERROR test_calculates_gear_inches
        undefined method 'diameter'
```

В провале нет ничего удивительного, именно это и должно было случиться, когда два конкретных объекта сотрудничают друг с другом, но при этом получатель сообщения изменился, а отправитель — нет. Код класса `Wheel` изменился, поэтому нужно изменить и код класса `Gear`. Тест дал сбой именно там, где и должен.

Тест не отличается особой сложностью, и его провал очевиден, потому что код носит абсолютно конкретный характер, но, как и все конкретное, он работает только в данном конкретном случае. Именно для этого кода данный тест является вполне подходящим, но ведь бывают и другие ситуации, когда вам, скорее всего, придется иметь дело с выявлением и тестированием абстракций.

Проблему лучше всего проиллюстрировать с помощью более экстремального примера. Если имеются сотни объектов с ролью `Diameterizable`, то как решить,

какой из них больше подходит для внедрения в ходе проведения теста? Если роль `Diameterizable` обходится слишком дорого, то как избежать запуска большого количества ненужного и затратного по времени кода? Здравый смысл подсказывает, что если на роль `Diameterizable` подходит только `Wheel`-объект и он работает довольно быстро, то тест должен внедрять только `Wheel`-объект. Но что делать, если выбор не столь очевиден?

Внедрение зависимостей в качестве ролей

Класс `Wheel` и роль `Diameterizable` связаны настолько тесно, что их трудно представить по отдельности, но осознание происходящего в предыдущем тесте требует их различать. И у `Gear`, и у `Wheel` имеются взаимоотношения с третьей стороной — с ролью `Diameterizable`. На рис. 9.4 показано, что от `Diameterizable` имеется зависимость у `Gear`, а сама роль реализована в `Wheel`.

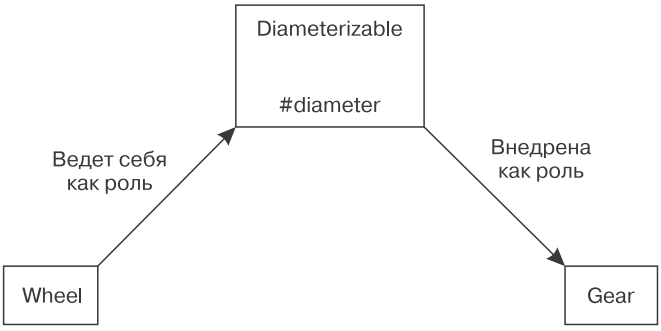

Рис. 9.4. Gear зависит от роли Diameterizable, а Wheel эту роль реализует

Эта роль является абстракцией идеи о том, что у несопоставимых объектов могут быть диаметры. Как и с другими абстракциями, вполне резонно ожидать, что эта абстрактная роль будет куда стабильнее, чем конкретика, из которой она исходит. Но в показанном выше случае справедливо обратное.

В коде есть два места, где объект зависит от знания роли `Diameterizable`. Первое находится там, где `Gear`-объект полагает, что ему известен интерфейс `Diameterizable`, то есть он верит, что может отправить сообщение `diameter` внедренному объекту. Второе относится к коду, создающему внедряемый объект, который верит, что данный интерфейс реализуется в классе `Wheel`, то есть он ожидает, что в `Wheel` реализуется `diameter`. При изменении роли `Diameterizable` возникает проблема. `Wheel` был обновлен для реализации нового ин-

терфейса, но, к сожалению, `Gear` по-прежнему ожидает наличия старого интерфейса.

Изюминка внедрения зависимости в том, что вы можете подставлять различные конкретные классы, не изменяя существующего кода. Новое поведение можно собрать путем создания новых объектов, играющих существующие роли, и внедрения этих объектов туда, где эти роли ожидаются. Объектно-ориентированная конструкция предписывает внедрение зависимостей, поскольку в ней полагается, что отдельно взятые конкретные классы будут больше подвергаться изменениям, чем роли, или наоборот: роли будут более стабильны, чем классы, из которых они были абстрагированы.

К сожалению, только что произошло совершенно обратное. В данном примере изменился не класс внедренного объекта, а интерфейс роли. Внедрение `Wheel`-объекта осталось приемлемым, а отправка в адрес `Wheel` сообщения `diameter` теперь стала недопустимой.

Когда у роли имеется единственный исполнитель, этот исполнитель и абстрактная роль настолько тесно связаны, что границы между ними размываются, но это не имеет никакого практического значения. В данном случае `Wheel` является единственным исполнителем роли `Diameterizable`, и на данный момент вы не ожидаете никаких других исполнителей. Если создание `Wheel`-объекта обходится дешево, внедрение `Wheel` не имеет какого-либо значимого негативного воздействия на ваши тесты.

Если код приложения может быть записан только одним способом, то эффективнее всего создавать тесты путем зеркального отображения той же схемы. Тогда тесты будут проваливаться при нужных обстоятельствах независимо от того, какими будут изменения — конкретными (имя класса `Wheel`) или абстрактными (интерфейс метода `diameter`).

Однако так бывает не всегда. Иногда обстоятельства заставляют отказаться от использования в тестах `Wheel`-объекта. Если приложение содержит много различных объектов, способных исполнять роль `Diameterizable`, у вас может возникнуть желание создать один идеализированный объект, чтобы ваши тесты ясно передавали замысел этой роли. Если создание всех `Diameterizable`-объектов обходится слишком дорого, то, чтобы заставить тесты выполняться быстрее, может потребоваться создание дешевого объекта-подделки. Если вы разрабатываете тесты через реализацию поведения (BDD), в приложении может не быть ни одного объекта, исполняющего эту роль, и тогда вам придется создать *что-нибудь* просто для написания теста.

Создание тестовых дублеров

В следующем примере рассматривается идея создания объекта-подделки (тестового дублера) для исполнения роли `Diameterizable`. Предположим, что для этого теста интерфейс `Diameterizable`-объекта вернулся к использованию исходного метода `diameter` и что `diameter` опять должным образом реализован в классе `Wheel`, а сообщение отправлено из `Gear`-объекта. В строке 2 создается подделка по имени `DiameterDouble`. В строке 13 эта подделка внедряется в `Gear`.

```
1  # Создание исполнителя роли 'Diameterizable'
2  class DiameterDouble
3    def diameter
4      10
5    end
6  end
7
8  class GearTest < MiniTest::Unit::TestCase
9    def test_calculates_gear_inches
10    gear = Gear.new(
11              chainring: 52,
12              cog:       11,
13              wheel:     DiameterDouble.new)
14
15    assert_in_delta(47.27,
16                    gear.gear_inches,
17                    0.01)
18   end
19 end
```

Тестовый дублер — это стилизованный экземпляр исполнителя роли, используемый исключительно для тестирования. Подобные дублеры создаются очень просто, и вам ничто не мешает применять их в каждой возможной ситуации. В каждом отдельном дублере подчеркивается единственная представляющая интерес особенность, а остальные свойства объекта можно перенести на второй план.

Дублер *создает заглушку* метода `diameter`, то есть реализует версию `diameter`, возвращающую законсервированный ответ. `DiameterDouble` выполняет крайне ограниченную функцию, но ничего другого от него и не требуется. Достаточно того, что он всегда возвращает в качестве значения диаметра число 10. Это постоянное возвращаемое значение — надежная основа для построения теста.

Встроенные способы создания дублеров и возвращения постоянных значений имеются у многих сред тестирования. Можно, конечно, воспользоваться и этими специализированными механизмами, но для простых тестовых дублеров вполне достаточно задействовать привычные объекты Ruby, что, собственно, и сделано в показанном выше примере.

`DiameterDouble` *не является* имитатором. Использование слова «имитатор» для описания этого дублера может легко войти в привычку, но имитаторы — совершенно иной механизм (в этой главе они рассмотрены в разделе «Тестирование исходящих сообщений»).

Внедрение дублера устраняет связь теста `Gear` с классом `Wheel`. Теперь неважно, насколько быстро выполняется код `Wheel`, поскольку код `DiameterDouble` выполняется быстро всегда. Этот тест работает вполне удовлетворительно и при запуске показывает следующее.

```
1 GearTest
2    PASS test_calculates_gear_inches
```

В этом тесте используется тестовый дублер, что позволяет ему быть простым, быстрым и изолированным, а также иметь явно выраженные намерения. Так что же может пойти не так?

Витание в облаках

А теперь представим, что в код вносятся прежние изменения: интерфейс `Diameterizable`-объекта изменяется с `diameter` на `width`, код `Wheel` обновляется, а код `Gear` остается прежним. Эти изменения опять нарушают работоспособность приложения. Вспомним, что прежний тест `Gear` (в котором вместо использования дублера проводилось внедрение `Wheel`-объекта) практически сразу же выявлял эту проблему и не проходил, выдавая ошибку вызова неизвестного метода `diameter`.

Но теперь, когда внедрен дублер `DiameterDouble`, при запуске теста произойдет следующее.

```
1 GearTest
2    PASS test_calculates_gear_inches
```

Тест *по-прежнему успешно проходит* даже при явном нарушении работоспособности приложения, потому что `Gear`-объект отправляет сообщение `diameter`, а в `Wheel`-объекте реализуется метод `width`.

Вы создали альтернативную вселенную, где тесты рапортуют, что все хорошо, несмотря на явный изъян в приложении. Возможность создания такой вселенной вызывает у некоторых опасения, что использование заглушек и подделок

сделает тесты хрупкими. Когда так происходит постоянно, это вина программиста, а не инструментария. Повышение качества кода предполагает понимание глубинных причин этой проблемы, что требует более пристального взгляда на его компоненты.

В приложении содержится роль `Diameterizable`. Изначально у этой роли был один исполнитель — `Wheel`. Затем `GearTest` создал `DiameterDouble` — *второго исполнителя роли*. Когда изменился интерфейс роли, новый интерфейс должны принять все исполнители роли. Но упустить из внимания исполнителей роли, созданных специально для тестов, довольно просто, что и произошло в нашем случае. Интерфейс в `Wheel` обновился, а в `DiameterDouble` остался прежним.

Тесты для документирования ролей

В том, что это произошло, нет ничего удивительного, поскольку роль почти невидима. В приложении нет такого места, на которое можно указать пальцем и сказать: «Этим самым определяется роль `Diameterizable`». Иногда помнить о существовании роли затруднительно, в итоге забывчивость неизбежно распространится и на то, что исполняют тестовые дублеры.

Один из способов повышения заметности роли — утверждение о том, что она выполняется классом `Wheel`. Именно это и делается в строке 6: роль документируется, также проверяется, что `Wheel` реализует интерфейс этой роли должным образом.

```
1  class WheelTest < MiniTest::Unit::TestCase
2    def setup
3       @wheel = Wheel.new(26, 1.5)
4    end
5
6    def test_implements_the_diameterizable_interface
7       assert_respond_to(@wheel, :diameter)
8    end
9
10   def test_calculates_diameter
11     wheel = Wheel.new(26, 1.5)
12
13     assert_in_delta(29,
14                     wheel.diameter,
15                     0.01)
16   end
17 end
```

Тест `implements_the_diameterizable_interface` нельзя назвать полностью удовлетворяющим нас решением. Фактически он не завершен. Во-первых, его нельзя совместно использовать с другими `Diameterizable`-объектами. Другим исполнителям данной роли придется этот тест продублировать. К тому же он не делает ничего, чтобы помочь в решении проблемы «витания в облаках» из теста `Gear`. Утверждение о том, что `Wheel` выполняет данную роль, не защищает имеющегося в `Gear` дублера `DiameterDouble` от устаревания и не гарантирует безошибочное прохождение теста методом `gear_inches`.

К счастью, проблема документирования и тестирования ролей имеет весьма простое решение, которое будет полностью рассмотрено в одном из следующих разделов, посвященных проверке корректности неявных типов. А пока достаточно усвоить, что роли нуждаются в своем собственном тестировании.

Целью данного раздела была организация проверки открытых интерфейсов путем тестирования входящих сообщений. Тестирование `Wheel` прошло без особых затрат. Исходный тест `Gear` обошелся дороже, поскольку он зависел от скрытой связанности с `Wheel`. Замена этой связанности внедренной зависимостью от объекта, исполняющего роль `Diameterizable`, привела к изоляции тестируемого объекта и создала дилемму: что именно внедрять — реальный объект или подделку под него?

Выбор между внедрением реального объекта или подделки имеет далеко идущие последствия. Внедрение при тестировании точно таких же объектов, что и при выполнении приложения, гарантирует, что тесты не будут проходить при неверно спроектированном коде, но может привести к тестам, на которые уходит много времени. А внедрение дублеров способно ускорить выполнение тестов, но сохранить при этом их уязвимость от построения иллюзорного мира, где тесты проходят, а приложение работать отказывается.

Следует заметить, что тестирование само по себе не принуждает к усовершенствованию конструкции. Ничто, касающееся тестов, *не заставляет* вас устранять связывание и внедрять зависимость. Хотя подход к тестированию снаружи вовнутрь (BDD) предоставляет более четкую линию проведения, чем TDD, ни один из этих подходов не мешает неопытному проектировщику написать код `Wheel`, а затем внедрить создание `Wheel`-объекта глубоко внутри `Gear`. Это связывание не препятствует созданию теста, но повышает его стоимость. Сокращение степени связанности зависит от вашего понимания принципов конструирования.

Тестирование закрытых методов

Иногда тестируемые объекты отправляют сообщения сами себе. Сообщения, отправляемые в адрес `self`, вызывают методы, определение которых находится в закрытом интерфейсе получателя. Этих закрытых сообщений вроде как и нет — по крайней мере для всего остального приложения. Поскольку отправка сообщений закрытым методам за пределами тестируемого объекта не видна, в первозданном мире идеальных конструкций они в тестировании не нуждаются.

Однако реальный мир более суров, и для него такого простого правила недостаточно. Работа с закрытыми методами требует разумного подхода и гибкости.

Игнорирование закрытых методов при тестировании

В пользу пропуска тестирования закрытых методов существует множество веских причин. Во-первых, подобные тесты излишни. Закрытые методы скрыты внутри тестируемого объекта, и возвращаемые ими результаты не могут быть видны за его пределами. Закрытые методы вызываются открытыми методами, *для которых уже имеются тесты*. Конечно же, ошибка, допущенная в закрытом методе, может привести к сбою всего приложения, но она всегда проявится при выполнении уже существующего теста. Необходимость в тестировании закрытых методов никогда не возникнет.

Во-вторых, закрытые методы нестабильны, поэтому тесты закрытых методов связаны с тем кодом приложения, вероятность внесения изменений в который довольно высока. При изменении приложения придется вносить изменения в тесты. Может запросто создаться ситуация, когда драгоценное время потрачено на выполнение текущего сопровождения никому не нужных тестов.

И наконец, тестирование закрытых методов может ввести в заблуждение и натолкнуть кого-либо другого на их использование. Тесты предоставляют документацию по тестируемому объекту. В них рассказывается о том, какого взаимодействия с миром в целом они ожидают. Включение в эту историю закрытых методов дезориентирует читателей насчет их основного предназначения

и подталкивает к нарушению инкапсуляции и приобретению зависимости от этих методов. Закрытые методы должны скрываться в ваших тестах, а не выставляться ими напоказ.

Удаление закрытых методов из тестируемого класса

Один из способов обойти эту проблему заключается в полном отказе от закрытых методов. Если их не будет, вопрос о тестировании закрытых методов отпадет сам собой.

Объект, имеющий множество закрытых методов, содержит все признаки того, что в его конструкции заложено слишком много обязанностей. Если у вашего объекта много закрытых методов, которые нельзя оставить без тестирования, подумайте, как извлечь методы в новый объект. Извлеченные методы сформируют обязанности нового объекта, составив его открытый интерфейс, который (теоретически) будет стабилен и достаточно безопасен с точки зрения приобретения от него зависимости.

Данная стратегия, конечно, хороша, но, к сожалению, польза от нее будет только при условии реальной стабильности нового интерфейса. Иногда с новым интерфейсом не все так гладко, и тут теория расходится с практикой. Новый открытый интерфейс будет стабилен (или нестабилен) ровно настолько, насколько был стабилен исходный закрытый интерфейс. Методы не становятся волшебным образом надежнее только из-за их перемещения. Связывать нестабильные методы слишком накладно, какими бы они ни были — открытыми или закрытыми.

Выбор в пользу тестирования закрытого метода

Иногда требуются решительные меры — порой вполне оправданно оставить несовершенный код в покое и закрыть глаза на беспорядок до поступления более конкретной информации. Скрыть беспорядок очень просто — нужно лишь заключить не доведенный до совершенства код в закрытый метод.

Если создать несовершенный код и не навести в нем в порядок, издержки пойдут вверх, но в краткосрочной перспективе для решения какой-то определенной

проблемы написание не доведенного до совершенства кода может сэкономить средства. Если вы намерены отложить проектное решение, текущие проблемы следует решать наипростейшими способами: изолируйте код, поместив его за самый лучший интерфейс, который только можете себе представить, и подождите, пока не появится дополнительная информация.

Применение данной стратегии может привести к использованию закрытых методов с неустоявшимся кодом. Раз вы пошли на это, рассмотрим тестирование этих нестабильных методов. Код приложения не отличается красотой и подвержен частым изменениям, поэтому есть риск привести что-либо в негодность.

Тесты получатся весьма затратными; скорее всего, их придется править сразу же после изменения основного кода, но любой другой вариант поддержания работоспособности может обойтись еще дороже.

Чтобы узнать, что изменения нарушили работоспособность, данные тесты не нужны, с этим по-прежнему превосходно справятся тесты открытых интерфейсов. При тестировании закрытых методов выдаются сообщения об ошибках, точно указывающие на неудачные места закрытого кода. Эти конкретные ошибки создают сильные связи и увеличивают расходы на сопровождение кода, но зато упрощают понимание эффектов от изменений и облегчают реструктуризацию сложного закрытого кода.

Преодоление барьеров, мешающих реструктуризации, играет весьма важную роль, поскольку вам все равно придется заниматься реструктуризацией кода. В этом весь смысл. Неразбериха в коде носит временный характер, избавляться от нее придется с помощью реструктуризации. По мере поступления дополнительной проектировочной информации код закрытых методов будет совершенствоваться. Как только туман развеется и конструкция обретет относительно четкие очертания, код в методах станет более стабильным. Как только стабильность повысится, затраты на сопровождение и необходимость в тестировании станут снижаться. Со временем появится возможность извлечь закрытые методы в отдельный класс и без каких-либо опасений представить их остальному миру.

Главное правило тестирования закрытых методов гласит: никогда не создавайте закрытых методов, а если вам все же пришлось это сделать, то никогда их не тестируйте без крайней необходимости. Поэтому относитесь предвзято к написанию подобных тестов, но не опасайтесь их написания, если они принесут несомненную пользу.

Тестирование исходящих сообщений

Исходящие сообщения, как уже известно из раздела «Выявление предмета тестирования», являются либо *запросами*, либо *командами*. Сообщения-запросы имеют значение только для отправившего их объекта, а сообщения-команды оказывают влияние на другие объекты приложения.

Игнорирование сообщений-запросов

Сообщения, не имеющие побочных эффектов, известны как сообщения-запросы. Рассмотрим простой пример, где определенный в классе `Gear` метод `gear_inches` отправляет сообщение `diameter`.

```
1 class Gear
2    # ...
3    def gear_inches
4      ratio * wheel.diameter
5    end
6 end
```

Отправка сообщения `diameter` не вызывает интереса ни у чего, кроме метода `gear_inches`. Метод `diameter` не имеет побочных эффектов, его запуск не оставляет видимых следов, и никакие другие объекты не зависят от его выполнения.

Точно так же, как тесты должны игнорировать сообщения, посланные объектом в свой собственный адрес (`self`), они должны игнорировать и исходящие сообщения-запросы. Последствия отправки сообщения `diameter` скрыты внутри `Gear`. Поскольку всему остальному приложению отправка этого сообщения не нужна, тестам заниматься им не стоит.

Работа определенного в классе `Gear` метода `gear_inches` зависит от результата, возвращаемого методом `diameter`, но тестирование с целью проверки правильности `diameter` принадлежит классу `Wheel`, но никак не классу `Gear`. Дублировать в `Gear` подобные тесты излишне, поскольку это неизбежно повысит расходы на сопровождение кода. Единственной обязанностью `Gear` является проверка правильности работы `gear_inches`; это может быть реализовано путем тестирования того факта, что `gear_inches` всегда возвращает приемлемые результаты.

Проверка сообщений-команд

Иногда факт отправки сообщения *играет весьма важную роль*, поскольку от того, что в результате этого произойдет, зависят другие части приложения. В таком случае тестируемый объект отвечает за отправку сообщения, и тесты должны подтвердить факт этой отправки.

Чтобы проиллюстрировать решение этой задачи, понадобится новый пример. Представьте себе игру, участники которой устраивают гонки на виртуальных велосипедах. У этих велосипедов, конечно же, имеются цепные передачи (gears). Класс `Gear` теперь отвечает за оповещение приложения о факте переключения игроком передачи, чтобы приложение могло изменить поведение велосипеда.

В следующем примере кода класс `Gear` отвечает этим новым требованиям за счет добавления `observer` (наблюдателя). Когда игрок переключает передачу, выполняется либо метод `set_cog`, либо метод `set_chainring`. Эти методы сохраняют новое значение величины задней или передней звездочки цепной передачи, а потом в связи с произошедшими изменениями вызывают определенный в классе `Gear` метод `changed` (строка 20). Затем этот метод отправляет сообщение `changed` в адрес `observer`, передавая с ним текущее значение `chainring` и `cog`.

```
1  class Gear
2    attr_reader :chainring, :cog, :wheel, :observer
3    def initialize(args)
4       # ...
5       @observer = args[:observer]
6    end
7
8     # ...
9
10   def set_cog(new_cog)
11     @cog = new_cog
12     changed
13   end
14
15   def set_chainring(new_chainring)
16     @chainring = new_chainring
17     changed
18   end
19
20   def changed
```

```
21      observer.changed(chainring, cog)
22    end
23  # ...
24  end
```

У класса `Gear` имеется новая обязанность: он должен оповестить `observer` об изменении `cog` или `chainring`. Эта обязанность важна не менее прежней, связанной с вычислением передаточного отношения в дюймах (gear inches). Когда игрок меняет передачу, приложение будет корректно работать только при условии отправки классом `Gear` сообщения `changed` в адрес `observer`. Тесты должны убедиться в том, что сообщение было отправлено (и выполнить свою задачу без каких-либо утверждений относительно результата, возвращаемого имеющимся в `observer` методом `changed`). Обязанность по выдаче утверждений о результате выполнения метода `changed` возлагается на тесты его владельца `observer` точно так же, как исключительная ответственность за выдачу утверждения о результате выполнения метода `diameter` возлагалась на тесты его владельца — класса `Wheel`. Обязанность по тестированию значения, возвращаемого сообщением, возлагается на его получателя. Совершение этого действия где-нибудь в другом месте приведет к появлению продублированного кода и к увеличению затрат.

Чтобы избежать дублирования кода, нужен способ подтверждения факта отправки кодом `Gear`-объекта сообщения `changed` в адрес `observer`, не заставляющего вас проверять возвращаемое после отправки значение. К счастью, все решается довольно просто. Вам нужен *имитатор* (mock). Имитаторы тестируют поведение, а не состояние. Вместо проверки того, что будет возвращено в результате отправки сообщения, имитаторы содержат предположение, что сообщение будет отправлено.

В показанном ниже тесте проверяется выполнение кодом `Gear` своих обязанностей, и делается это без привязки к подробностям поведения `observer`. В тесте создается имитатор (строка 4), который внедряется вместо `observer` (строка 8). Каждый тестовый метод предписывает имитатору ожидать получения сообщения `changed` (строки 12 и 18), а затем убеждается, что он его дождался (строки 14 и 20).

```
1  class GearTest < MiniTest::Unit::TestCase
2
3    def setup
4      @observer = MiniTest::Mock.new
5      @gear = Gear.new(
```

```
 6                chainring: 52,
 7                cog:       11,
 8                observer:  @observer)
 9    end
10
11    def test_notifies_observers_when_cogs_change
12      @observer.expect(:changed, true, [52, 27])
13      @gear.set_cog(27)
14      @observer.verify
15    end
16
17    def test_notifies_observers_when_chainrings_change
18      @observer.expect(:changed, true, [42, 11])
19      @gear.set_chainring(42)
20      @observer.verify
21    end
22  end
```

Это классическое использование схемы имитатора. В показанном выше тесте уведомления наблюдателей при изменении значения `cog` имеют вид `notifies_observers_when_cogs_change`. В строке 12 имитатору указывается, какое именно сообщение следует ожидать, в строке 13 запускается поведение, заставляющее это ожидание сбыться, в строке 14 от имитатора требуется проверка факта отправки сообщения. Тест проходит, только если отправка `set_cog` в адрес `gear` вызывает нечто, заставляющее `observer` получить сообщение `changed` с заданными аргументами.

Заметьте, что единственное действие имитатора в отношении сообщения свелось к тому, что он запомнил его получение. Если тестируемый объект зависит от получаемого им обратно результата после получения `observer` сообщения `changed`, имитатор может быть настроен на возвращение соответствующего значения. Но это возвращаемое значение к делу не относится. Имитаторы предназначены для проверки факта отправки сообщений, результаты возвращаются ими только при необходимости получения работоспособных тестов.

Факт сохранения работоспособности `Gear` даже после имитации работы принадлежащего `observer` метода `changed`, при которой не делается практически ничего, доказывает, что классу `Gear` все равно, чем на самом деле занимается этот метод. Единственной обязанностью `Gear` является отправка сообщения, и этот тест сам себя ограничивает лишь доказательством факта, что `Gear` с этой обязанностью справляется.

В качественно спроектированном приложении тестирование исходящих сообщений не доставляет особых хлопот. При наличии заранее введенных зависимостей их можно легко заменить имитаторами. Установка ожиданий для этих имитаторов позволяет проверить факт выполнения тестируемым объектом своих обязанностей без каких-либо утверждений, принадлежащих другим компонентам кода.

Тестирование неявных типов

В разделе «Тестирование входящих сообщений» уже заходил разговор о тестировании ролей, но до предоставления подходящего решения дело не дошло. Теперь настало время вернуться к этой теме и изучить вопрос тестирования неявных типов. В данном разделе показано, как создаются тесты, которые могут совместно использоваться исполнителями ролей. Затем мы вернемся к исходной проблеме и применим совместно используемые тесты.

Тестирование ролей

Код для первого примера взят из неявного типа `Preparer`, который был рассмотрен в главе 5. Первые несколько примеров кода повторяют часть материала, изложенного в главе 5, и если память у вас еще свежа, можете сразу переходить к первому тесту.

Вспомним, что собой представляют исходные классы `Mechanic`, `TripCoordinator` и `Driver`.

```
class Mechanic
  def prepare_bicycle(bicycle)
     #...
   end
end

class TripCoordinator
  def buy_food(customers)
     #...
  end
end

```

```
class Driver
  def gas_up(vehicle)
    #...
  end
  def fill_water_tank(vehicle)
    #...
  end
end
```

Каждый из этих классов обладает продуманным открытым интерфейсом, в то время как `Trip`-объект, использующий эти интерфейсы для подготовки путешествия, вынужден проверять класс каждого объекта для определения того, какое именно сообщение ему отправлять.

```
class Trip
  attr_reader :bicycles, :customers, :vehicle

  def prepare(preparers)
     preparers.each {|preparer|
      case preparer
      when Mechanic
         preparer.prepare_bicycles(bicycles)
      when TripCoordinator
        preparer.buy_food(customers)
      when Driver
        preparer.gas_up(vehicle)
        preparer.fill_water_tank(vehicle)
      end
    }
  end
end
```

Показанная выше инструкция `case` связывает `prepare` с тремя существующими конкретными классами. Представьте себе попытки тестирования метода `prepare` или последствия добавления к этому смешанному коду новой разновидности `preparer`. Этот метод сложно тестировать и накладно сопровождать.

Если вам повстречается код, использующий такой антишаблон, но не имеющий тестов, то перед созданием тестов присмотритесь к возможности его реструктуризации в более грамотную конструкцию. Внесение изменения при от-

сутствии тестов представляет немалую опасность, но это нагромождение кода имеет настолько хрупкую природу, что его предварительная реструктуризация может оказаться наиболее экономичной стратегией. Реструктуризация, устраняющая проблему, не представляет особой сложности и существенно упрощает все последующие изменения.

Первая стадия реструктуризации касается принятия решения о конструкции интерфейса `Preparer` и его реализации в каждом исполнителе роли. Если открытым интерфейсом `Preparer` станет `prepare_trip`, то следующие изменения позволят исполнять роль объектам классов `Mechanic`, `TripCoordinator` и `Driver`.

```
class Mechanic
   def prepare_trip(trip)
     trip.bicycles.each {|bicycle|
     prepare_bicycle(bicycle)}
   end

   # ...
end

class TripCoordinator
  def prepare_trip(trip)
    buy_food(trip.customers)
  end

  # ...
end

class Driver
  def prepare_trip(trip)
    vehicle = trip.vehicle
    gas_up(vehicle)
    fill_water_tank(vehicle)
  end
  # ...
end
```

Теперь, когда существуют объекты с неявным типом `Preparer`, определяемый в `Trip` метод `prepare` может быть существенно упрощен. Следующая реструктуризация вносит в принадлежащий `Trip` метод `prepare` изменения для сотрудничества

с объектами неявного типа `Preparer`, упраздняя отправку уникальных сообщений каждому конкретному классу.

```
1  class Trip
2    attr_reader :bicycles, :customers, :vehicle
3
4    def prepare(preparers)
5      preparers.each {|preparer|
6      preparer.prepare_trip(self)}
7    end
8  end
```

После этой реструктуризации можно переходить к написанию тестов. В показанный выше код заложено взаимодействие между объектами неявного типа `Preparer` и объектами класса `Trip`, который теперь может считаться исполнителем роли `Preparable` (пригодным к подготовке). В тестах должно быть задокументировано существование роли `Preparer`, подтверждено, что каждый ее исполнитель ведет себя правильно, и показано, что `Trip` взаимодействует с ними надлежащим образом.

Поскольку действия в качестве `Preparer`-объектов характерны для нескольких различных классов, тест роли должен быть создан единожды и совместно использоваться каждым исполнителем. MiniTest представляет собой среду тестирования, не требующую особых формальностей, и поддерживает совместно используемые тесты наипростейшим из возможных способов посредством модулей Ruby. Рассмотрим модуль, тестирующий и документирующий интерфейс `Preparer`.

```
1  module PreparerInterfaceTest
2    def test_implements_the_preparer_interface
3      assert_respond_to(@object, :prepare_trip)
4    end
5  end
```

Модуль убеждается в том, что `@object` реагирует на `prepare_trip`. Показанный ниже тест использует этот модуль, чтобы убедиться, что `Mechanic`-объект является также объектом неявного типа `Preparer`. Он включает модуль (строка 2) и предоставляет при выполнении метода `setup` объект типа `Mechanic`, используя для этого переменную `@object` (строка 5).

```
1  class MechanicTest < MiniTest::Unit::TestCase
2    include PreparerInterfaceTest
3
```

```
4    def setup
5      @mechanic = @object = Mechanic.new
6    end
7
8    # other tests which rely on @mechanic
9  end
```

Тесты `TripCoordinator` и `Driver` следуют той же схеме. В них также включается модуль (строки 2 и 10 во фрагменте кода ниже), в их методах `setup` инициализируется переменная `@object` (строки 5 и 13).

```
1  class TripCoordinatorTest < MiniTest::Unit::TestCase
2    include PreparerInterfaceTest
3
4    def setup
5       @trip_coordinator = @object = TripCoordinator.new
6    end
7  end
8
9  class DriverTest < MiniTest::Unit::TestCase
10   include PreparerInterfaceTest
11
12   def setup
13     @driver = @object = Driver.new
14   end
15 end
```

Выполнение этих трех тестов приводит к вполне приемлемому результату.

```
1  DriverTest
2     PASS test_implements_the_preparer_interface
3
4  MechanicTest
5     PASS test_implements_the_preparer_interface
6
7  TripCoordinatorTest
8     PASS test_implements_the_preparer_interface
```

Определение `PreparerInterfaceTest` (теста интерфейса подготовителя путешествия) в виде модуля позволяет написать тест единожды, а затем повторно использовать в каждом объекте, играющем данную роль. Модуль служит в качестве теста и в качестве документации. Он поднимает степень видимости роли и упрощает подтверждение того факта, что любой вновь созданный `Preparer`-объект успешно справляется со своими обязанностями.

Метод `test_implements_the_preparer_interface` тестирует входящее сообщение и в качестве такового относится к тестам объектов-получателей; именно поэтому модуль включен в тесты классов `Mechanic`, `TripCoordinator` и `Driver`. Но входящие сообщения идут рука об руку с исходящими сообщениями, и вам нужно тестировать обе части этого уравнения. Вы уже убедились, что все получатели надлежащим образом реализуют метод `prepare_trip`, теперь же вам следует убедиться, что `Trip` правильно отправляет соответствующее сообщение.

Как вы уже знаете, убедиться в отправке исходящего сообщения можно путем установки ожиданий в имитаторе. В следующем тесте создается имитатор (строка 4), ему предписывается ожидать сообщение `prepare_trip` (строка 6), инициируется принадлежащий `Trip` метод `prepare` (строка 8), а затем проверяется, получил ли имитатор надлежащее сообщение (строка 9).

```
1 class TripTest < MiniTest::Unit::TestCase
2
3   def test_requests_trip_preparation
4     @preparer = MiniTest::Mock.new
5     @trip = Trip.new
6     @preparer.expect(:prepare_trip, nil, [@trip])
7
8     @trip.prepare([@preparer])
9     @preparer.verify
10  end
11 end
```

Тест `test_requests_trip_preparation` находится непосредственно в `TripTest`. Объекты класса `Trip` единственные в приложении, кто исполняет роль `Preparable`, поэтому нет никаких других объектов, с которыми пришлось бы совместно использовать данный тест. Если появятся другие объекты, исполняющие роль `Preparable`, то нужно будет выделить тест в модуль, который будет совместно применяться всеми объектами, исполняющими данную роль.

Выполнение теста позволяет убедиться, что `Trip` сотрудничает с объектами, играющими роль `Preparer`, используя для этого надлежащий интерфейс.

```
1 TripTest
2   PASS test_requests_trip_preparation
```

Тестирование роли `Preparer` завершено, можно вернуться к использованию дублеров для исполнения ролей в тестах.

Ролевые тесты для проверки дублеров

После того как вы узнали о способах написания повторно используемых тестов, позволяющих убедиться в том, что объект правильно играет свою роль, этим же приемом можно воспользоваться для устранения хрупкости, вызванной применением заглушек.

Ранее в разделе «Тестирование входящих сообщений» поднималась проблема «витания в облаках». Заключительный тест раздела выдавал ложноположительный результат, из-за которого удавалось проходить заведомо провальный тест (причиной был тестовый дублер, который служил заглушкой устаревшего метода). Вспомним, как выглядит тест, прохождение которого не должно было состояться.

```
class DiameterDouble

  def diameter # Этот интерфейс уже изменился на 'width',
    10         # но данный дублер, а также объект класса Gear
  end          # по-прежнему используют 'diameter'.
end

class GearTest < MiniTest::Unit::TestCase
  def test_calculates_gear_inches
   gear = Gear.new(
             chainring: 52,
             cog:       11,
             wheel:     DiameterDouble.new)

   assert_in_delta(47.27,
                   gear.gear_inches,
                   0.01)
  end
end
```

Теперь при наличии этого теста у вас собрались все пазлы, необходимые для решения проблемы хрупкости. Вы знаете, как можно организовать совместное использование тестов исполнителями ролей, вы узнали о наличии двух исполнителей роли `Diameterizable`, и у вас есть тест, которым может воспользоваться любой объект, чтобы убедиться в правильности исполнения своей роли.

Первый шаг к решению этой проблемы — извлечение теста `test_implements_the_diameterizable_interface` из `Wheel` в его собственный модуль.

```
1  module DiameterizableInterfaceTest
2    def test_implements_the_diameterizable_interface
3      assert_respond_to(@object, :width)
4    end
5  end
```

Как только этот модуль появится, возвращение извлеченного поведения обратно в `WheelTest` сведется к простому включению модуля (строка 2) и к инициализации переменной `@object` значением `Wheel`-объекта (строка 5).

```
1  class WheelTest < MiniTest::Unit::TestCase
2    include DiameterizableInterfaceTest
3
4    def setup
5      @wheel = @object = Wheel.new(26, 1.5)
6    end
7
8    def test_calculates_diameter
9      # ...
10   end
11 end
```

Теперь `WheelTest` работает точно так же, как и до извлечения, в чем можно убедиться, выполнив тест.

```
1 WheelTest
2   PASS test_implements_the_diameterizable_interface
3   PASS test_calculates_diameter
```

Хорошо, конечно, что тест `WheelTest` по-прежнему проходит, но эта реорганизация преследует более обширные цели, чем простое перестроение кода. Теперь, когда есть независимый модуль, позволяющий убедиться в том, что роль `Diameterizable` исполняется правильно, этот модуль можно использовать, чтобы не дать тестовым дублерам молча устареть.

Показанный ниже тест `GearTest` был обновлен с целью использования этого нового модуля. В строках с 9-й по 15-ю определен новый тестовый класс `DiameterDoubleTest`. Он не имеет к `Gear` никакого отношения, его назначение заключается в предотвращении хрупкости теста путем обеспечения постоянной правдоподобности дублера.

```
class DiameterDouble
   def diameter
     10
   end
end

# Проверка соблюдения тестовым дублером интерфейса,
# ожидаемого этим тестом.
class DiameterDoubleTest < MiniTest::Unit::TestCase
  include DiameterizableInterfaceTest

  def setup
    @object = DiameterDouble.new
  end
end

class GearTest < MiniTest::Unit::TestCase
  def test_calculates_gear_inches
    gear = Gear.new(
              chainring: 52,
              cog:       11,
              wheel:     DiameterDouble.new)

    assert_in_delta(47.27,
                    gear.gear_inches,
                    0.01)
  end
end
```

Тот факт, что и `DiameterDouble`, и `Gear` содержат неправильный код, позволял предыдущим версиям тестов успешно завершаться. Теперь, когда дублер тестируется на предмет четкого выполнения своей роли, при запуске теста наконец-то выдается ошибка.

```
DiameterDoubleTest
  FAIL test_implements_the_diameterizable_interface
        Expected #<DiameterDouble:...> (DiameterDouble)
          to respond to #width.
GearTest
  PASS test_calculates_gear_inches
```

Тест GearTest по-прежнему выполняется без ошибок, но это не создает никаких проблем, поскольку теперь DiameterDoubleTest информирует вас, что в DiameterDouble используется неверный код. Это заставляет внести в DiameterDouble коррективы, касающиеся реализации width, показанные в строке 2.

```
1  class DiameterDouble
2    def width
3      10
4    end
5  end
```

После этого изменения перезапущенный тест выдает ошибку в GearTest.

```
1  DiameterDoubleTest
2     PASS test_implements_the_diameterizable_interface
3
4  GearTest
5    ERROR test_calculates_gear_inches
6          undefined method 'diameter'
7            for #<DiameterDouble:0x0000010090a7f8>
8              gear_test.rb:35:in 'gear_inches'
9              gear_test.rb:86:in 'test_calculates_gear_inches'
10
```

Теперь проходит DiameterDoubleTest и не проходит GearTest. Такой сбой напрямую указывает на проблемную строку кода в Gear. И наконец, тесты сообщают о необходимости внесения изменений в принадлежащий Gear метод gear_inches, чтобы вместо diameter отправлялось, как в следующем примере, сообщение width.

```
1  class Gear
2
3    def gear_inches
4      # и наконец, вместо 'diameter' отправляется 'width'
5      ratio * wheel.width
6    end
7
8    # ...
9  end
```

После внесения этого последнего изменения приложение обретает правильный код и все тесты проходят без ошибок.

```
1 DiameterDoubleTest
2    PASS test_implements_the_diameterizable_interface
```

```
3
4 GearTest
5   PASS test_calculates_gear_inches
```

Однако польза не только в том, что этот тест прошел, а и в том, что он будет проходить (или не проходить) в соответствии с обстоятельствами, что бы ни случилось с интерфейсом `Diameterizable`. Когда тестовые дублеры рассматриваются в таком же качестве, как и другие исполнители роли, и тестируются на предмет корректности, это позволяет обойтись без теста на хрупкость и использовать заглушку, не опасаясь последствий.

Стремление протестировать неявные типы создает потребность в применении совместно используемых ролевых тестов. Как только у вас появляется возможность оценить обстановку с точки зрения исполняемых ролей, ею можно воспользоваться во многих ситуациях. С точки зрения тестируемого объекта любой другой объект исполняет роль, и обращение с объектами как с представителями исполняемых ими ролей ослабляет связанность и повышает гибкость не только приложения, но и тестов.

Тестирование унаследованного кода

Мы добрались до решения последней непростой задачи — до тестирования унаследованного кода. Этот раздел во многом похож на предыдущие — мы детальнее рассмотрим ранее встречавшийся пример (финальная иерархия `Bicycle` из главы 6) и перейдем к его тестированию. Несмотря на то что в конечном итоге иерархия оказалась непригодной для наследования, положенный в ее основу код вполне сгодится.

Определение унаследованного интерфейса

Рассмотрим класс `Bicycle` в том виде, в каком он встречался в главе 6.

```
1 class Bicycle
2   attr_reader :size, :chain, :tire_size
3
4   def initialize(args={})
5     @size = args[:size]
6     @chain = args[:chain] || default_chain
7     @tire_size = args[:tire_size] || default_tire_size
8     post_initialize(args)
```

```
  end

  def spares
    { tire_size: tire_size,
      chain: chain}.merge(local_spares)
  end

  def default_tire_size
    raise NotImplementedError
  end

  # может быть переопределен в подклассах
  def post_initialize(args)
    nil
  end

  def local_spares
    {}
  end

  def default_chain
    '10-speed'
  end
end
```

А вот код для дорожного велосипеда `RoadBike`, одного из подклассов `Bicycle`.

```
class RoadBike < Bicycle
  attr_reader :tape_color

  def post_initialize(args)
     @tape_color = args[:tape_color]
  end

  def local_spares
     {tape_color: tape_color}
  end

  def default_tire_size
    '23'
  end
end
```

Первоначальная задача тестирования — подтверждение факта, что все объекты в иерархии соблюдают свои контракты. Принцип подстановки Лисков объявляет, что подтипы должны подходить в качестве подстановки для своих родительских типов. Нарушение принципа Лисков делает объекты ненадежными (они начинают вести себя неожиданным образом). Проще всего убедиться в том, что каждый объект иерархии придерживается принципа Лисков, — это написать совместно используемый тест для общего контракта и включить этот тест в каждый объект.

Контракт воплощается в общем интерфейсе. Следующий тест дает четкую формулировку интерфейса и тем самым определяет, что означает быть `Bicycle` (велосипедом в общем смысле).

```
module BicycleInterfaceTest
  def test_responds_to_default_tire_size
     assert_respond_to(@object, :default_tire_size)
  end

  def test_responds_to_default_chain
     assert_respond_to(@object, :default_chain)
  end

  def test_responds_to_chain
    assert_respond_to(@object, :chain)
  end

  def test_responds_to_size
    assert_respond_to(@object, :size)
  end

  def test_responds_to_tire_size
    assert_respond_to(@object, :tire_size)
  end

  def test_responds_to_spares
    assert_respond_to(@object, :spares)
  end
end
```

Любой объект, который проходит тест `BicycleInterfaceTest`, может считаться действующим в качестве `Bicycle`. Все классы в иерархии `Bicycle` должны откликаться на этот интерфейс и проходить этот тест. В следующем примере

тест интерфейса включен в абстрактный родительский класс `BicycleTest` (строка 2) и в конкретный подкласс `RoadBikeTest` (строка 10).

```
1  class BicycleTest < MiniTest::Unit::TestCase
2    include BicycleInterfaceTest
3
4    def setup
5      @bike = @object = Bicycle.new({tire_size: 0})
6    end
7  end
8
9  class RoadBikeTest < MiniTest::Unit::TestCase
10   include BicycleInterfaceTest
11
12   def setup
13     @bike = @object = RoadBike.new
14   end
15 end
```

При выполнении теста выдаются следующие результаты.

```
1  BicycleTest
2    PASS test_responds_to_default_chain
3    PASS test_responds_to_size
4    PASS test_responds_to_tire_size
5    PASS test_responds_to_chain
6    PASS test_responds_to_spares
7    PASS test_responds_to_default_tire_size
8
9  RoadBikeTest
10   PASS test_responds_to_chain
11   PASS test_responds_to_tire_size
12   PASS test_responds_to_default_chain
13   PASS test_responds_to_spares
14   PASS test_responds_to_default_tire_size
15   PASS test_responds_to_size
```

ПРИМЕЧАНИЕ

Пусть вас не смущает, что части BicycleTest и RoadBikeTest выполняются не по порядку, поскольку произвольный порядок запуска является особенностью среды тестирования MiniTest.

Тест `BicycleInterfaceTest` будет работать для любой разновидности `Bicycle` и может быть легко включен в любой новый подкласс. Он документирует интерфейс и предотвращает любую случайную регрессию.

Определение обязанностей подкласса

Кроме того, что все элементы иерархии `Bicycle` используют общий интерфейс, родительский класс `Bicycle` предъявляет к своим подклассам вполне определенные требования.

Подтверждение поведения подклассов

При наличии множества подклассов все они должны совместно использовать общий тест для подтверждения своего соответствия требованиям. Документирование требований для подклассов осуществляется в следующем тесте.

```
module BicycleSubclassTest
  def test_responds_to_post_initialize
     assert_respond_to(@object, :post_initialize)
  end

  def test_responds_to_local_spares
     assert_respond_to(@object, :local_spares)
  end

  def test_responds_to_default_tire_size
    assert_respond_to(@object, :default_tire_size)
  end
end
```

В этом тесте систематизируются требования для подклассов иерархии `Bicycle`. Фактически он не заставляет подклассы реализовывать эти методы, `post_initialize` и `local_spares` могут свободно унаследовать любой подкласс. Этот тест просто проверяет, что в подклассах нет ничего из ряда вон выходящего, чтобы помешало бы им реагировать на соответствующие сообщения. Единственный метод, который должен быть реализован подклассами, — это `default_tire_size`. Имеющаяся в родительском классе реализация `default_tire_size` приводит к ошибке, и если в подклассе не реализована своя собственная специализированная версия этого метода, то тест пройден не будет.

Подкласс `RoadBike` действует точно так же, как и `Bicycle`, поэтому его тест уже включает `BicycleInterfaceTest`. Показанный ниже тест был изменен с целью включения в него нового теста `BicycleSubclassTest`, поскольку `RoadBike` должен также действовать в качестве подкласса `Bicycle`.

```
class RoadBikeTest < MiniTest::Unit::TestCase
   include BicycleInterfaceTest
   include BicycleSubclassTest

   def setup
     @bike = @object = RoadBike.new
   end
end
```

Выполнение измененного теста приводит к следующим расширенным результатам.

```
 RoadBikeTest
    PASS test_responds_to_default_tire_size
    PASS test_responds_to_spares
    PASS test_responds_to_chain
    PASS test_responds_to_post_initialize
    PASS test_responds_to_local_spares
    PASS test_responds_to_size
    PASS test_responds_to_tire_size
    PASS test_responds_to_default_chain
```

Эти два модуля должны совместно использоваться каждым подклассом `Bicycle`, поскольку каждый подкласс обязан действовать и как `Bicycle`, и как подкласс `Bicycle`. Хотя с момента вашего знакомства с подклассом `MountainBike` (горный велосипед) прошло уже немало времени, вы можете по достоинству оценить появившуюся возможность убедиться в том, что объекты `MountainBike` достойно представляют иерархию, просто добавив к их тесту два модуля.

```
class MountainBikeTest < MiniTest::Unit::TestCase
   include BicycleInterfaceTest
   include BicycleSubclassTest

   def setup
     @bike = @object = MountainBike.new
   end
end
```

Сочетание `BicycleInterfaceTest` и `BicycleSubclassTest` полностью решает проблему тестирования общего поведения подклассов. Эти тесты убедят, что подклассы не отклоняются от стандарта, и позволят новичкам создавать новые подклассы в условиях полной безопасности. Программистам, недавно подключившимся к работе с этим кодом, не нужно рыскать по родительским классам в поисках требований, они могут просто включить эти тесты при написании новых подклассов.

Подтверждение навязывания требований родительским классом

Если в подклассе не реализуется метод `default_tire_size`, класс `Bicycle` должен выдавать ошибку. Хотя это требование применяется к подклассам, фактическое поведение, принуждающее к его выполнению, принадлежит классу `Bicycle`. Поэтому соответствующий тест помещается, как показано в строке 8, непосредственно в `BicycleTest`.

```
1  class BicycleTest < MiniTest::Unit::TestCase
2    include BicycleInterfaceTest
3
4    def setup
5      @bike = @object = Bicycle.new({tire_size: 0})
6    end
7
8    def test_forces_subclasses_to_implement_default_tire_size
9      assert_raises(NotImplementedError) {@bike.default_tire_size}
10   end
11 end
```

Заметьте, что в строке 5 теста `BicycleTest` предоставляется размер шин (tire size), хотя это делается дополнительно во время создания объекта `Bicycle`. Причина станет ясна, если вернуться к принадлежащему `Bicycle` методу `initialize`. Метод `initialize` ожидает для `tire_size` либо получения входящего значения, либо возможности извлечения этого значения путем последующей отправки сообщения `default_tire_size`. Если аргумент `tire_size` из строки 5 удалить, то тест прекратит выполнение в своем методе `setup` при создании объекта `Bicycle`. Без этого аргумента `Bicycle` не сможет успешно преодолеть инициализацию объекта.

Аргумент `tire_size` необходим, потому что `Bicycle` является абстрактным классом, не ожидающим получения сообщения `new`. У `Bicycle` нет дружелюбного

протокола создания. Он ему и не нужен, поскольку само приложение никогда не создает экземпляры класса `Bicycle`. Но тот факт, что приложение не создает новые объекты `Bicycle`, еще не означает, что это никогда не случается. Конечно же, случается. В строке 5 показанного выше теста `BicycleTest` происходит явное создание нового экземпляра этого абстрактного класса.

При тестировании абстрактных классов подобная проблема возникает довольно часто. Тесту `BicycleTest` необходим объект, в отношении которого нужно запускать тесты, и наиболее очевидным кандидатом является экземпляр класса `Bicycle`. Однако создать новый экземпляр абстрактного класса нелегко, а иногда вообще невозможно. Этому тесту повезло: протокол создания экземпляра `Bicycle` позволяет создать конкретный экземпляр `Bicycle` путем передачи аргумента `tire_size`, но создать тестируемый объект не всегда легко, и может возникнуть необходимость применения более тонкой стратегии. К счастью, есть несложный способ обхода этой общей проблемы, который рассмотрен далее в разделе «Тестирование поведения абстрактного родительского класса».

В нашем случае вполне хватает предоставления аргумента `tire_size`. Запуск `BicycleTest` теперь выдает результат, который больше похож на результат тестирования абстрактного родительского класса.

```
1 BicycleTest
2    PASS test_responds_to_default_tire_size
3    PASS test_responds_to_size
4    PASS test_responds_to_default_chain
5    PASS test_responds_to_tire_size
6    PASS test_responds_to_chain
7    PASS test_responds_to_spares
8    PASS test_forces_subclasses_to_implement_default_tire_size
```

Тестирование уникального поведения

До сих пор тесты наследования сводились к тестированию общих качеств. Большинство создаваемых тестов подходили для совместного использования, поэтому помещались в модули (`BicycleInterfaceTest` и `BicycleSubclassTest`), но один тест (`forces_subclasses_to_implement_default_tire_size`) был помещен непосредственно в тест `BicycleTest`.

Мы разобрались с общим поведением, но до сих пор не созданы тесты по конкретике (ни для той, что предоставляется конкретными подклассами, ни для той, что определяется в абстрактных родительских классах). В следующем раз-

деле основное внимание уделяется первой разновидности этих тестов и протестирована конкретика, предоставляемая отдельно взятыми подклассами. Далее внимание направлено на тестирование уникальным для класса `Bicycle` поведения.

Тестирование конкретного поведения подкласса

Настало время еще раз напомнить о намерениях создать абсолютный минимум тестов. Вернемся к классу `RoadBike`. Основная часть его поведения уже проверяется с помощью совместно используемых модулей. Осталось только протестировать конкретику, предоставляемую `RoadBike`.

Важно, чтобы это было сделано без внедрения в тест знания о родительском классе. Например, в `RoadBike` реализуется метод `local_spares`, а также реакция на сообщение `spares`. Тест `RoadBikeTest` должен проследить за работой `local_spares`, сохраняя при этом намеренную неосведомленность о существовании метода `spares`. Тест `BicycleInterfaceTest` уже позволяет убедиться, что `RoadBike` правильно реагирует на `spares`, поэтому ссылаться на этот метод непосредственно в данном тесте излишне.

Однако метод `local_spares` вполне определенно является обязанностью подкласса `RoadBike`. В строке 9 именно эта конкретика тестируется непосредственно в `RoadBikeTest`.

```
1 class RoadBikeTest < MiniTest::Unit::TestCase
2    include BicycleInterfaceTest
3    include BicycleSubclassTest
4
5    def setup
6      @bike = @object = RoadBike.new(tape_color: 'red')
7    end
8
9    def test_puts_tape_color_in_local_spares
10    assert_equal 'red', @bike.local_spares[:tape_color]
11  end
12 end
```

Теперь запуск `RoadBikeTest` показывает, что класс соответствует своим общим обязанностям, а также предоставляет свою собственную конкретику.

```
1 RoadBikeTest
2    PASS test_responds_to_default_chain
3    PASS test_responds_to_default_tire_size
```

```
 4    PASS test_puts_tape_color_in_local_spares
 5    PASS test_responds_to_spares
 6    PASS test_responds_to_size
 7    PASS test_responds_to_local_spares
 8    PASS test_responds_to_post_initialize
 9    PASS test_responds_to_tire_size
10   PASS test_responds_to_chain
```

Тестирование поведения абстрактного родительского класса

Теперь после тестирования конкретики подкласса настал черед завершить тестирование родительского класса. Перемещение фокуса вверх по иерархии к `Bicycle` возвращает нас к ранее встречавшейся проблеме. `Bicycle` является абстрактным родительским классом. Мало того, что создание экземпляра `Bicycle` — непростая задача, дело еще и в том, что у экземпляра может не иметься всего того поведения, которое необходимо для проведения теста.

К счастью, решение может быть принято с учетом уже приобретенного вами опыта. Поскольку в `Bicycle` для получения конкретики использовались шаблонные методы, можно смоделировать в виде заглушки то поведение, которое обычно поставляется подклассами. Еще лучше, помня о принципах подстановки Лисков, воспользоваться возможностью изготовления тестируемого экземпляра `Bicycle` путем создания нового подкласса, применяемого исключительно для этого теста.

Показанный ниже тест соответствует этой стратегии. В строке 1 определяется новый класс `StubbedBike`, который является подклассом `Bicycle`. В тесте создается экземпляр этого класса (строка 15), который используется для проверки факта включения локальных запчастей `local_spares` подклассов в запчасти `spares` (строка 23).

Иногда все же удобнее создавать экземпляр абстрактного класса `Bicycle`, даже притом что для этого требуется передача аргумента `tire_size`, как показано в строке 14. Экземпляр `Bicycle` продолжает использоваться в тесте (строка 18), чтобы убедиться в принуждении родительским классом подклассов к реализации метода `default_tire_size`.

Две разновидности экземпляров `Bicycle` мирно сосуществуют в одном тесте.

```
1  class StubbedBike < Bicycle
2    def default_tire_size
3      0
4    end
```

```
    def local_spares
      {saddle: 'painful'}
    end
  end

class BicycleTest < MiniTest::Unit::TestCase
  include BicycleInterfaceTest

  def setup
    @bike = @object = Bicycle.new({tire_size: 0})
    @stubbed_bike = StubbedBike.new
  end

  def test_forces_subclasses_to_implement_default_tire_size
    assert_raises(NotImplementedError) {
    @bike.default_tire_size}
  end

  def test_includes_local_spares_in_spares
    assert_equal @stubbed_bike.spares,
                 { tire_size: 0,
                   chain:     '10-speed',
                   saddle:    'painful'}
  end
end
```

Идея создания подкласса для предоставления заглушки может пригодиться во многих ситуациях. При условии, что ваш новый подкласс не нарушает принципа Лисков, этот прием можно использовать в любом тесте.

Теперь выполнение `BicycleTest` доказывает включение в список запчастей `spares` вкладов со стороны подклассов.

```
BicycleTest
   PASS test_responds_to_spares
   PASS test_responds_to_tire_size
   PASS test_responds_to_default_chain
   PASS test_responds_to_default_tire_size
   PASS test_forces_subclasses_to_implement_default_tire_size
   PASS test_responds_to_chain
   PASS test_includes_local_spares_in_spares
   PASS test_responds_to_size
```

И последний штрих: если вас волнует, что `StubbedBike` может утратить свою актуальность и позволить тесту `BicycleTest` успешно пройти, когда он должен дать сбой, то решение, как говорится, под рукой. У вас уже есть тест `BicycleSubclassTest`. Для получения гарантий о сохраняющемся надлежащем поведении `DiameterDouble` вы использовали тест `DiameterizableInterfaceTest`, точно так же для получения гарантий сохранения актуальности `StubbedBike` вы можете воспользоваться `BicycleSubclassTest`. Добавьте к `BicycleTest` следующий код.

```
# Подтверждение факта соблюдения тестовым дубликатом интерфейса,
# ожидаемого этим тестом.
class StubbedBikeTest < MiniTest::Unit::TestCase
  include BicycleSubclassTest

  def setup
    @object = StubbedBike.new
  end
end
```

После внесения этих изменений выполнение `BicycleTest` даст следующий дополнительный результат.

```
StubbedBikeTest
  PASS test_responds_to_default_tire_size
  PASS test_responds_to_local_spares
  PASS test_responds_to_post_initialize
```

Грамотно спроектированная иерархия наследования легко поддается тестированию. Для всего интерфейса нужно создать один совместно используемый тест, а также еще один тест для обязанностей подклассов. Нужно также тщательно изолировать обязанности друг от друга. Особую внимательность следует проявить при тестировании конкретики подклассов, чтобы не дать сведениям о родительском классе просочиться вниз, в тесты подклассов.

При тестировании абстрактных подклассов могут возникнуть трудности, поэтому при создании тестов нужно воспользоваться принципом подстановки Лисков. Если вы применили принцип Лисков и создали новый подкласс, используемый исключительно для тестирования, рассмотрите возможность его применения для тестирования обязанности вашего подкласса, позволяющего убедиться в том, что она случайно не утратила свою актуальность.

Выводы

Тестирование обязательно должно проводиться. Грамотно спроектированные приложения обладают высокой абстрагированностью и постоянно находятся в процессе развития, поэтому без тестов в них станет невозможно не только разбираться, но и безопасно вносить какие-либо изменения. Самые качественные тесты отличаются слабой связанностью с основным кодом, тестирование всех компонентов проводится в них только один раз и в нужном месте. Тесты приносят пользу, не увеличивая при этом расходов.

Изучение качественно спроектированного приложения с тщательно созданным набором тестов и его расширение принесет вам истинное удовольствие. Такое приложение можно приспособить к любому новому обстоятельству, оно способно отвечать даже самым неожиданным требованиям.

Заключение

Обязанности, зависимости, интерфейсы, неявные типы, наследования, совместно используемое поведение, составление композиций и тестирование — все это вы уже изучили. Вы окунулись с головой в мир объектов, и если эта книга достигла своей цели, то теперь ваше представление об объектах отличается от того, каким оно было вначале.

В главе 1 утверждалось, что объектно-ориентированное проектирование предусматривает управление зависимостями. Справедливость этого утверждения бесспорна, но это не единственная истина, касающаяся проектирования.

Важный момент — наличие способа, позволяющего считать все объекты одинаковыми независимо от того, что они собой представляют, будь то все приложение целиком, его основные подсистемы, отдельные классы или простые методы. Отдельно взятый объект никогда не стоит особняком, приложение состоит из объектов, связанных друг с другом. Все определяется не тем, чем именно занимаются объекты, а передающимися между ними сообщениями. Объектно-ориентированное проектирование — это своеобразный фрактал в виде определяемой основной задачи и расширяемых способов связи объектов друг с другом (и при каждом уровне увеличения эта задача выглядит одинаково).

Из этой книги вы узнали немало правил, касающихся написания кода (управления зависимостями и создания интерфейсов). Теперь, когда вы их усвоили, можете приспособить под собственные нужды. Противоречия, свойственные проектированию, напоминают о том, что нужно предусмотреть и случаи нарушения этих правил. Опытный проектировщик умеет нарушать эти правила с пользой для дела.

В основу проектирования положены инструментальные средства, и по мере освоения вами приемов проектирования они естественным образом превратятся в подручный инструментарий, позволяющий создавать легко изменяемые приложения, полностью отвечающие своему предназначению и приносящие вам истинную радость. Конечно, ваши приложения не будут идеальными, но это не должно вас останавливать. Достичь совершенства крайне сложно и даже, наверное, невозможно, но это не повод не стремиться к его достижению. Проявляйте упорство. Набирайтесь опыта. Экспериментируйте. Развивайте воображение. Старайтесь выкладываться на все сто, а все остальное приложится.

Сэнди Метц

Ruby. Объектно-ориентированное проектирование

Серия «Библиотека программиста»

Перевел с английского *Н. Вильчинский*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*О. Андросик*
Художественный редактор	*С. Заматевская*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*Г. Блинов*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.
Подписано в печать 01.11.16. Формат 70x100/16. Бумага писчая. Усл. п. л. 24,510. Тираж 1000. Заказ 0000
Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*