ATDD. Разработка программного обеспечения через приемочные тесты

Применяя методику разработки через приемочные тесты (ATDD), заказчики, разработчики и тестировщики получают возможность совместно сформулировать тестопригодные требования, что позволяет создавать высококачественное программное обеспечение в сжатые сроки. Однако на практике многие не понимают, в чем истинный смысл ATDD. Это первое практическое руководство начального уровня по внедрению и успешному применению этой методики.

Автор шаг за шагом демонстрирует читателю, как получить от специалистов в предметной области правильную постановку задачи и затем реализовать полностью автоматизированные функциональные тесты, точно отражающие бизнес-требования и понятные всем заинтересованным сторонам. При этом сама процедура создания тестов способствует эффективной разработке продукта.

На двух сквозных примерах Гэртнер показывает, как применять ATDD, используя различные языки и каркасы. В результате проработки примеров порождаются различные артефакты: классы для автоматизации тестирования, определения шагов и полные реализации. На этих вполне реалистичных примерах автор иллюстрирует фундаментальные принципы ATDD, показывает место ATDD в общем процессе разработки, делится своим обширным опытом и предостерегает против типичных ошибок. Вот некоторые из рассматриваемых тем:

- мыслительный процесс, ведущий к успешному внедрению ATDD;
- применение системы Cucumber для описания программы в понятных заказчику терминах;
- тестирование веб-страниц с помощью инструментов ATDD;
- написанная на Java система FitNesse — каркас приемочного тестирования на основе вики;
- эффективное описание примеров с помощью методики разработки на основе поведения (BDD);
- рабочие встречи — новаторский способ совместной выработки спецификаций;
- внедрение дружественных к пользователю и поощряющих сотрудничество методов автоматизации тестирования;
- рациональное тестирование, прислушивание к результатам тестов и их рефакторинг для повышения качества.

Эта книга будет полезна тестировщикам, разработчикам, бизнес-аналитикам и руководителям проектов. Она позволит заложить прочный фундамент для получения первых результатов от внедрения ATDD уже сейчас и поможет добиться еще большего в будущем, по мере накопления опыта.

Интернет-магазин:
www.dmkpress.com
Книга – почтой:
e-mail: orders@alians-kniga.ru
Оптовая продажа:
«Альянс-книга»
Тел./факс: (499) 782-3889
e-mail: books@alians-kniga.ru

www.дмк.рф

Addison-Wesley
Pearson Education

ISBN 978-5-97060-418-2

ATDD. Разработка через приемочные тесты

The Addison-Wesley Signature Series

A Kent Beck Signature Book

ATDD

Разработка программного обеспечения через приемочные тесты

Маркус Гэртнер

Практический подход

Маркус Гэртнер

ATDD – разработка программного обеспечения через приемочные тесты

ATDD by Example

A Practical Guide to Acceptance Test-Driven Development

Markus Gärtner

Addison-Wesley

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

ATDD – разработка программного обеспечения через приемочные тесты

Маркус Гэртнер

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2016

УДК 004.054ATDD
ББК 32.973-018
Г21

Г21 Маркус Гэртнер
ATDD – разработка программного обеспечения через приемочные тесты. Пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2016. – 232 с.: ил.

ISBN 978-5-97060-418-2

Применяя методику разработки через приемочные тесты (ATDD), заказчики, разработчики и тестировщики получают возможность совместно сформулировать тестопригодные требования, что позволяет создавать высококачественное программное обеспечение в сжатые сроки. Однако на практике многие не понимают, в чем истинный смысл ATDD. Настоящая книга – первое практическое руководство начального уровня по внедрению и успешному применению этой методики.

На примерах автор показывает, как применять ATDD, используя различные языки и каркасы. В результате проработки примеров порождаются различные артефакты: классы для автоматизации тестирования, определения шагов и полные реализации. На этих вполне реалистичных примерах автор иллюстрирует фундаментальные принципы ATDD, показывает место ATDD в общем процессе разработки, делится своим обширным опытом и предостерегает против типичных ошибок.

Эта книга будет полезна тестировщикам, разработчикам, бизнес-аналитикам и руководителям проектов. Она позволит заложить прочный фундамент для получения первых результатов от внедрения ATDD уже сейчас и поможет добиться еще большего в будущем, по мере накопления опыта.

ISBN 978-0-321-78415-5 (англ.)
ISBN 978-5-97060-418-2

Мой жене Дженнифер,
моему пасынку Леону
и нашей дочери Катрин
за то, что они не обижались, когда я
проводил с ними слишком мало времени,
работая над этой книгой

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ Кента Бека

Существует любопытная симметрия между тем, как в этой книге представлена методология разработки через приемочные тесты (ATDD – Acceptance Test-Driven Development), и тем, как с помощью этой методологии разрабатывается программное обеспечение. Умение выбрать конкретные примеры поведения программы, которые выявляли бы правильное поведение системы в целом, – это особое искусство, как и искусство подобрать такие примеры применения методики, подобной ATDD, которая дала бы читателю возможность освоить ее. Маркус проделал блистательную работу по выбору и представлению примеров.

Читая эту книгу, придется вникать в код. Продвигаясь вперед, вы поймете, как нужно изменить взгляд на проблему, чтобы успешно применять ATDD. В двух словах эта смена мировоззрения заключается в том, чтобы вместо «Вот эту функцию я хочу включить» говорить «А как мы будем ее тестировать? Вот пример». Изучая примеры, вы снова и снова будете встречаться с этим переосмыслением в разных контекстах.

Что мне нравится в построении изложения вокруг примеров кода, так это излучаемая автором уверенность в вашей способности учиться. Это не просто «12 простых правил тестирования веб-приложения», отпечатанных на тонкой папиросной бумаге, которая расползается при первом соприкосновении с влажной реальностью. Вы узнаете о конкретных решениях, принимаемых в конкретных контекстах, о решениях, с которыми вы можете (и обязательно будете, если хотите извлечь из книги максимум пользы) не соглашаться, спорить и делать по-своему.

Ближе к концу книги формулируются общие выводы, подводящие итог принципам, продемонстрированным на примерах. Если вы из тех, кому легче усвоить материал, понимая общие идеи, то начать лучше с этого места. Но в любом случае эффект от прочтения этой

книги прямо будет пропорционален усилиям, которые вы приложите для осмысления примеров.

Одна из слабых сторон первоначального варианта методики разработки через тестирование (TDD) заключается в том, что она может превратиться в технику, которую программист использует для нужд собственно программирования. Некоторые программисты смотрят на TDD более широко, легко переходя в тестах от одного уровня абстрагирования к другому. Но в ATDD такой двусмысленности нет – эта методика предназначена для общения с людьми, которые с языками программирования не знакомы вовсе. Качество отношений с заказчиком – и лежащего в их основе взаимопонимания – способствует повышению эффективности процесса разработки ПО. ATDD может стать шагом в направлении более ясного выражения мыслей, а эта книга – обстоятельное и доступное введение в тему.

– Кент Бек

ПРЕДИСЛОВИЕ
Дэйла Эмери

Не счесть программных проектов, не отвечающих ожиданиям заказчика. На протяжении многих лет я десятки раз слышал, как заказчик объясняет причины провала тем, что *разработчики не обращали внимания на наши пожелания*. А сотни разработчиков, напротив, считают: *«Заказчик не говорит, чего он хочет. По большей части он вообще не знает, чего хочет»*.

Я видел достаточно проектов, чтобы прийти к другому выводу – сформулировать, что должна делать программная система, трудно. Для этого нужно предельно точно говорить и внимательно слушать, а в обычных взаимоотношениях между людьми так бывает редко, да и не очень-то необходимо. Писать хорошие программы трудно. Качественно тестировать программы трудно. Но труднее всего четко объяснить, чего мы хотим от системы.

Методика разработки через приемочные тесты (ATDD) помогает справиться с этой проблемой. С ее помощью вся команда совместно добивается ясности и точного понимания еще до начала разработки. В основе ATDD лежат два приема. Прежде чем приступать к реализации какой-либо функции, члены команды формулируют конкретные примеры ее практического применения. Затем эти примеры переводятся на язык автоматизированных приемочных тестов. Примеры и тесты становятся важной частью точного и разделяемого всеми сторонами описания того, что понимать под словом «сделано» для каждой функции.

Какова цена общего понимания? На семинаре по ATDD один разработчик объяснил это такими словами: «После того как мы начали совместно работать над примерами, мне стало *небезразлично*, что мы делаем. Я наконец-то стал понимать, что и зачем мы создаем. И что еще более важно, я был уверен, что вся команда одинаково понимает, чего мы пытаемся достичь. У нас появилась общая цель – мы стали одной командой».

ATDD не только помогает понять, когда разработку функции следует считать законченной, но и когда приступать к тестированию этой функции (и всех предыдущих). Примеры служат верстовыми столбами на пути к завершению. А поскольку каждый пример описывает ситуацию, значимую для заказчика, то мы можем быть уверены, что не только продвигаемся вперед, но и движемся в нужном направлении.

Ну ладно, я упомянул о нескольких важных особенностях и преимуществах ATDD. Это-то было просто. А вот вопрос посложнее: как всё это организовать в реальном проекте? Ответ оставляю Маркусу Гэртнеру. В этой книге Мартин закатывает рукава и не только рассказывает, но и *показывает*, как ATDD работает на практике. Он позволяет постоять у него за спиной и увидеть, как тестировщики, программисты и бизнес-аналитики думают, применяя принципы и приемы ATDD.

Хочу предупредить об одном подводном камне. В первых главах – когда мы следим, как бизнес-аналитик Билл, тестировщик Тони и программисты Филлис и Алекс описывают и реализуют небольшую программную систему, – материал может показаться чрезмерно упрощенным, даже примитивным. Не поддавайтесь первому впечатлению. В этих главах происходит *много* интересного. Это весьма квалифицированная команда, и некоторые аспекты ее работы на первый взгляд неочевидны. Например, обратите внимание, что при обсуждении требований нет никаких упоминаний о технологии. Члены команды говорят исключительно о бизнес-функциях системы. И заметьте, что когда Тони и Алекс автоматизируют первые тесты, Тони небезуспешно пользуется *отсутствием* у себя опыта программирования. Столкнувшись с непонятной технической деталью, он просит Алекса объяснить ее, а затем они вместе исправляют код, так чтобы он говорил сам за себя. И обратите внимание, как часто Алекс настаивает на сохранении тестов в системе управления версиями, – но только после того как код заработает. Если вы новичок в ATDD, то эти моменты могут показаться неочевидными, но они – неотъемлемая составная часть успеха.

По счастью, чтобы осознать эти тонкие нюансы, нужно всего лишь продолжать чтение. Маркус часто делает паузы, чтобы объяснить, что и почему делает команда. В конце каждой главы он резюмирует, как команда работала совместно, о чем думали ее члены и какие приемы применяли. А в заключительной части книги Маркус сводит все воедино, подробно описывая принципы, на которых зиждется ATDD.

Эта книга – прекрасное введение в методику разработки через приемочные тесты. Заодно она позволяет по-новому взглянуть на ATDD людям, которые уже применяет ее на практике. И наконец, она заслуживает многократного прочтения. Итак, читайте, практикуйтесь и снова читайте. Каждый раз вы будете находить что-то новое и полезное.

– Дэйл Эмери

ВСТУПЛЕНИЕ

Эта книга представляет собой введение в методику, которая получила название «разработка через приемочные тесты», или ATDD (Acceptance Test-Driven Development). Впервые встретив этот термин в 2008 году, я счел его искусственным и излишним. Мне казалось, что это избыточное понятие, потому что тогда я как раз изучал разработку через тестирование и считал, что этого вполне достаточно. В конце концов, зачем бы мне могло понадобиться тестировать программу на соответствие приемочным критериям?

«Каждому воздастся по заслугам»[1] [Wei86]. И вот четыре года спустя я пишу книгу, посвященную именно этой методике. В 2009 году я познакомился с Гойко Аджичем (Gojko Adzic), который только что закончил писать книгу «Bridging the Communication Gap» [Adz09]. Он подарил мне экземпляр, и я тут же, на обратном пути из Лондона, начал читать ее. А когда закончил, имел ясное понимание того, что такое ATDD и почему мы не должны употреблять этот термин.

Но отчего же тогда на обложке книги, которую вы держите в руках, значатся слова «ATDD – разработка через приемочные тесты»?[2]

О названии

Методика ATDD родилась не вчера. Она известна под разными названиями. Вот их неполный перечень:

- разработка через приемочные тесты (Acceptance Test-Driven Development);

1 В оригинале «Time wounds all heels» – парафраз известного изречения «Time heals all wounds» (время лечит все раны). Так называется рассказ американского фантаста Роберта Блоха, написанный в 1942 году и опубликованный в сборнике «Lost in Space and Time with Lefty Feep» 1987 года. Джон Леннон произнес эту фразу в ответ на попытки ФБР депортировать его из США, имея в виду, что время расставит все по своим местам и его притеснителям воздастся по делам их. – *Прим. перев.*

2 Или почему в файле, представляющем электронное издание, встречается именно такая комбинация нулей и единиц?

- разработка на основе поведения (Behavior-Driven Development – BDD);
- спецификирование через пример (Specification by Example);
- гибкое приемочное тестирование (Agile Acceptance Testing);
- тестирование по историям (Story Testing).

На мой взгляд, все эти названия несовершенны. Фраза «разработка через приемочные тесты» подразумевает, что итерацию можно считать законченной, когда все приемочные тесты проходят. Но это не так, потому что ни один набор тестов не дает полного покрытия. Всегда что-то остается непротестированным. Невозможность протестировать программу полностью – хорошо известный феномен. Как говорит Майкл Болтон (Michael Bolton), мы можем быть уверены лишь в одном – если приемочные тесты не проходят, то дело не закончено.

Но я решил привести не аргументы в пользу того или другого названия, а собственно перечень вариантов и дать читателю возможность самому решить, какое название лучше отвечает его представлениям. В конце концов, название не важно, если сама методика решает поставленную задачу. В области разработки программного обеспечения полно неудачных терминов, и такое положение, вероятно, сохранится и в будущем. Программная инженерия, автоматизация тестирования, разработка через тестирование – все эти слова так или иначе вводят в заблуждение. При абстрагировании никогда не следует путать название с предметом. Специалисты знают о недостатках, присущих именованию любого подхода.

Но почему похожие методики называются по-разному? Потому что способы их практического применения могут очень сильно различаться. Имея за плечами опыт консультирования многочисленных коллективов по ATDD, я могу сказать, что общее у них только одно: каждая команда чем-то отличается от всех остальных. Подход, успешно работающий в одной группе внутри одной компании, может привести к полному провалу в другой. Вы никогда не задумывались о смысле слов «зависит от обстоятельств», произносимых консультантом? Он имеет в виду именно это явление.

Работая над книгой «Specification by Example» [Adz11], Гойко Аджич побеседовал более чем с пятьюдесятью коллективами, применяющими ATDD в том или ином виде. Те, что добились успеха, неизменно начинали с базовой методики, а затем пересматривали ее и вносили коррективы с учетом конкретных условий. Начинать с простого процесса и адаптировать его к обстоятельствам, столкнув-

шись с проблемами, – самый гибкий (agile) способ внедрения любой методики. Применяя ATDD, памятуйте о том, что первая попытка вряд ли решит все ваши проблемы. Со временем, накопив опыт, вы сможете видоизменить процесс с учетом конкретных особенностей работы своей команды.

Зачем нужна еще одна книга по ATDD?

Гойко описывает много примеров успешного внедрения ATDD, но я обнаружил существенный пробел во всех имеющихся на сегодняшний день книгах по ATDD. Между опытными командами, давно практикующими тот или иной подход или методику, и командами, только приступающими к ее освоению, имеется огромная разница.

Знакомясь с литературой по ATDD, я наткнулся на несколько книг, в которых ATDD объясняется на продвинутом уровне со ссылками на принципы. Опытному читателю не составит труда применить принципы к конкретному случаю. Но для новичка это совсем не так. Начинающему, чтобы понять, что к чему, нужны более конкретные указания. А уже потом, накопив опыт, он сможет выйти за рамки жестких ограничений, налагаемых методикой.

Новичку проще освоить материал, следуя точному рецепту, но это вовсе не означает, что эта книга – сборник рецептов по ATDD. На рассматриваемых примерах я предлагаю два работоспособных подхода к ATDD и показываю, как думают участники процесса. Начинающий может воспользоваться этими примерами, приступая к внедрению ATDD в своем коллективе. По ходу изложения я буду давать ссылки на более углубленные источники.

Основная идея заимствована из книги Кента Бека «Test-Driven Development: By Example»[3] [Bec02]. Бек приводит два примера разработки через тестирование и в конце объясняет некоторые лежащие в их основе принципы. Книга задумана как вводное описание методики TDD и дает начинающему достаточно материала, чтобы приступить к практическому применению, – в предположении, что TDD можно научиться путем осмысления и практики. В какой-то степени это относится и к данной книге.

3 Кент Бек «Экстремальное программирование: разработка через тестирование». Питер, 2003. *Прим. перев.*

Терминология

В этой книге я пользуюсь терминами из области гибкой разработки ПО. Поскольку не все знают о том, что такое гибкая разработка, я считаю уместным дать краткое описание терминов.

Владелец продукта (product owner). В гибкой методике Scrum определены три роли: команда разработчиков, скрам-мастер (ScrumMaster) и владелец продукта. Владелец продукта отвечает за успех продукта, создаваемого командой. Он устанавливает приоритеты реализации различных функций, обсуждая их со всеми заинтересованными сторонами. Он же играет в команде роль представителя заказчика и в этом качестве принимает решения о деталях – обязательно обговаривая их с другими заинтересованными сторонами.

Итерация, или **забег** (sprint). Основой гибкой разработки является повторяющийся цикл, состоящий из итераций, или – в терминологии Scrum – забегов. Это короткие отрезки, в ходе которых команда реализует одно расширение функциональности, потенциально допускающее включение в готовый продукт. Продолжительность типичной итерации составляет от одной до четырех недель.

Пользовательская история (user story). Это ограниченный набор функций, которые, по мнению команды, можно без напряжения реализовать в ходе одной итерации. Это крохотные срезы функциональности продукта. Обычно команда стремится реализовать за одну итерацию несколько пользовательских историй. За определение историй отвечает представитель заказчика или владелец продукта.

Доска задач (taskboard). В большинстве команд, практикующих гибкую разработку, работа планируется на доске, видной каждому сотруднику. Используются карточки, на которых написано, кто что делает. Доска задач обычно разбита не несколько столбцов – как минимум, «Предстоит сделать», «Делается», «Сделано». На доске отражается текущее состояние работы над продуктом.

Карточка истории (story card). Пользовательские истории обычно записываются на бумажных карточках. По ходу итерации карточки вывешиваются на доску задач.

Ежедневная оперативка (standup meeting, daily Scrum). Не реже раза в день члены команды рассказывают о текущем состоянии дел. Команда собирается на 15 минут и обсуждает, что нужно сделать для завершения оставшихся задач, запланированных на текущую итерацию.

Очередь работ для продукта (product backlog), **очередь работ для забега** (sprint backlog). В методологии Scrum владелец продукта организует еще не реализованные истории в виде очереди работ для продукта. Владелец отвечает за обновление очереди при появлении новых требований. Планируя следующую итерацию, члены команды отбирают задачи для включения в очередь работ для забега. Отобранные из очереди работ для продукта истории автоматически включаются в очередь работ для забега. Чаще всего очередь работ для забега вывешивается на доске задач по завершении планерки.

Рефакторинг. Так называется изменение структуры исходного кода без изменения решаемой им задачи. Я обычно провожу рефакторинг перед внесением изменений. Это позволяет упростить реализацию предстоящих изменений.

Разработка через тестирование (Test-Driven Development – TDD). Методика разработки через тестирование предполагает, что сначала пишется один тест, который не проходит, затем пишется код, необходимый и достаточный для того чтобы тест прошел (и при этом не «сломались» ранее написанные тесты), и после этого код подвергается рефакторингу в преддверии следующего крохотного шажка. TDD – это подход к проектированию, он позволяет улучшить качество кода, потому что по определению создается код, допускающий тестирование.

Непрерывная интеграция, НИ (continuous integration – CI). Это процедура частого включения изменений в исходный код. Сервер сборки собирает всю ветвь, прогоняет все автономные и приемочные тесты, после чего рассылает информацию о новой сборке всем членам команды. В основе НИ лежит какая-то технология автоматизированной сборки, а сама процедура позволяет команде узнать о проблемах в текущей ветви на ранней стадии, а не за час до передачи новой версии заказчику.

Как читать эту книгу

В этой книге описываются как конкретные практические приемы, так и принципы, которые я считаю полезными. Читать книгу можно поразному – всё зависит от уровня вашей подготовки.

Можно читать книгу от корки до корки. Вы узнаете о системе Cucumber, разработке на основе поведения и о том, как тестировать с помощью инструментария ATDD веб-страницы. В нашем первом

примере речь пойдет о команде, где существует четкая граница между программистами и тестировщиками. Вы увидите, что сотрудничество – одно из ключевых условий успеха.

Во второй части мы с вами составим пару. Объединившись в пару, мы сможем компенсировать отсутствие знаний о тестировании или программировании. Мы напишем код приложения, применяя ATDD на практике. Мы познакомимся с основанным на вики каркасом приемочного тестирования FitNesse. Примеры из второй части написаны на Java.

В третьей части приведены рекомендации по внедрению методики. Я даю ссылки на материал для дальнейшего чтения, а также советую, как приступать к делу, и делюсь своими наблюдениями об успешном и не очень успешном опыте применения ATDD в других командах.

В приложениях вы найдете описание как обоих использованных в этой книге инструментов, так и еще одного – достаточно подробные, чтобы приступить к работе. Если раньше вы не встречались с Cucumber или FitNesse, то можете начать с чтения этих приложений.

Опытный читатель может пропустить первые две части и сразу перейти к принципам, изложенным в третьей части. Возможно, впоследствии вы захотите ввести своих коллег в курс дела – для этого могут пригодиться примеры из первой и второй части.

Можно поступить и по-другому – познакомиться с первыми двумя примерами и сразу же попробовать внедрить простой процесс. Зайдя в тупик, можете обратиться к третьей части – хотя я бы не рекомендовал читать книгу в таком порядке.

Если в вашей команде методика ATDD уже внедрена, то, быть может, вам будет интересно более глубоко изучить часть II, где я объясняю, как управлять разработкой предметного кода, отталкиваясь от примеров.

Это лишь некоторые варианты чтения книги, которые пришли мне в голову. Если вы похожи на меня, то, наверное, подумываете о том, чтобы набрать приведенный в примерах код и «поиграться» с ним. Я подготовил репозиторий на сайте github, содержащий код обоих примеров. Это позволит провести приемочное тестирование «вживую». Если вы где-то застрянете, то сможете заодно использовать этот материал как подсказку. Примеры из первой части находятся по адресу http://github.com/mgaertne/airport, а из второй – по адресу http://github.com/mgaertne/trafficlights.

БЛАГОДАРНОСТИ

Такая книга не могла бы состояться без поддержки многих помощников. Прежде всего, я хотел бы поблагодарить Дэйла Эмери, высказавшего очень полезные замечания касательно моего слога. Поскольку английский язык для меня не родной, комментарии Дэйла пришлись весьма кстати.

Отдельное спасибо Кенту Беку. В августе 2010 я говорил с ним по поводу написания книги по ATDD по образцу его книги «TDD by Example». Он же привел меня в издательство Addison-Wesley и представил Кристоферу Гузиковски (Christopher Guzikowski), который оказал мне всемерную поддержку в издании этой книги.

Многие люди принимали участие в рецензировании ранних вариантов текста. Я благодарен Лайзе Криспин (Lisa Crispin), Мэтту Хойсеру (Matt Heusser), Элизабет Хендриксон (Elisabeth Hendrickson), Бретту Шухерту (Brett Schuchert), Гойко Аджичу (Gojko Adzic), Джорджу Динвидди (George Dinwiddie), Кэвину Боди (Kevin Bodie), Олафу Левитцу (Olaf Lewitz), Мануэлю Кюбльбёку (Manuel Küblböck), Андреасу Хавенштайну (Andreas Havenstein), Себастьяну Санитцу (Sebastian Sanitz), Майку Мерчу (Meike Mertsch), Грегору Грамлиху (Gregor Gramlich) и Стефану Кэмперу (Stephan Kämper).

И не в последнюю очередь я хочу поблагодарить свою жену Дженнифер и наших детей Катрин и Леона за поддержку во время работы над этой книгой. Надеюсь в будущем компенсировать то время, в течение которого вы были вынуждены обходиться без мужа и папы.

ОБ АВТОРЕ

Маркус Гэртнер работает тестировщиком, инструктором, тренером и консультантом в компании it-agile GmbH, базирующейся в Гамбурге. Ученик Джерри Вайнберга, Маркус в 2011 году основал германскую рабочую и исследовательскую группу по гибким методикам тестирования. Он также является сооснователем Европейского отделения организации Weekend Testing. Он инструктор и обладатель черного пояса в школе тестирования программного обеспечения Miagi-Do, является участником сообщества пишущих авторов Agile Alliance FTT-Patterns, а также движения Software Craftsmanship. Маркус регулярно выступает на конференциях по гибкой разработке и тестированию по всему миру и активно пишет статьи о тестировании, преимущественно в контексте гибкой разработки. Его личный блог находится по адресу shino.de/blog. Он ведет заказные курсы по ATDD и контекстному тестированию. Он преподавал ATDD тестировщикам без технического образования, а также нескольким программистам.

ЧАСТЬ I.

Парковка в аэропорту

В этой части мы будем рассматривать онлайновое приложение. Автоматизированное тестирование веб-страниц — вещь, которая сегодня неплохо реализуется с помощью графического интерфейса пользователя. Но у такого подхода есть и недостатки. Впрочем, многие команды, которые имеют дело с онлайновыми приложениями, найдут в этой части советы о том, как управлять своими тестами.

В качестве приложения мы возьмем калькулятор стоимости парковки в международном аэропорту. В аэропорту есть несколько парковок с разными тарифами за время пребывания автомобиля.

Бизнес-правила вычисления стоимости парковки довольно сложны, поэтому последняя попытка команды разработать онлайновое приложение закончилась неудачей. Поскольку члены команды полагают, что неправильно поняли требования, они решили изменить подход и обсудить приложение с заказчиком на рабочей встрече по выработке спецификаций (технического задания). Тестировщики, программисты и заказчики собираются рассмотреть примеры, описывающие бизнес-правила для калькулятора стоимости парковки.

Пока программисты будут работать, тестировщик займется автоматизированием примеров с помощью комбинации Cucumber, Ruby и Selenium. В какой-то момент ему может понадобиться помощь программиста, но не буду раскрывать все секреты во введении.

Кстати, если вам интересно, в каком редакторе тестировщик Тони набирает спецификации примеров на Cucumber, сообщу, что это, естественно, emacs, а не vi.

ГЛАВА 1. Рабочая встреча по калькулятору стоимости парковки

Какое-то время назад компания Major International Airport Corp. решила расширить свое присутствие в Интернете. В частности, она хочет, чтобы сайт предоставлял потенциальным пассажирам возможность заранее рассчитать стоимость парковки. Пассажиру нужно будет только заполнить форму, а калькулятор вычислит стоимость парковки на выбранной стоянке за все время поездки.

Major International Airport Corp. уже пыталась разработать такую форму, но отзывы пассажиров были настолько негативными, что администрация решила переписать все заново.

Памятуя о неудаче, команда разработчиков в составе старшего программиста Филлис, программиста Алекса и тестировщика Тони решила подойти к задаче по-другому. В прошлый раз требования, похоже, постоянно изменялись, из-за чего приходилось накладывать заплату на заплату, а закончилось всё признанием, что изначально создавалось не то, что нужно.

Вместо того чтобы повторять весь процесс заново, команда решает созвать рабочее совещание и сформулировать на нем бизнес-правила калькулятора стоимости парковки. Для подготовки нового технического задания Филлис и Тони пригласили руководителя отдела парковок Билла, который знает о расчете стоимости всё.

Парковка с доставкой в назначенное место

Филлис Итак, давайте обсудим требования к расчету стоимости парковки. Билл, что скажешь?

Билл У нас есть парковки трех типов. На одних почасовая оплата, на других – посуточная, на третьих установлен суточный или недельный максимум.

Филлис А как вы отличаете одну парковку от другой? У них есть какие-то названия?

Билл Парковка с доставкой в назначенное место, краткосрочная и стандартная. При утрате жетона взимается штраф 10 долларов.

Филлис Давайте остановимся на этих трех типах. Чем они отличаются?

Билл На парковке с доставкой в назначенное место пассажир оставляет свою машину и получает квитанцию с указанием места доставки.

Филлис А как насчет стоимости?

Билл Парковка с доставкой стоит 18 долларов в сутки. Если машина пробыла на стоянке не более пяти часов, предоставляется скидка 6 долларов.

Тони Минутку, Билл. Ты хочешь сказать, что даже оставив машину на 30 минут, я должен буду заплатить 12 долларов – столько же, сколько за три или за пять часов? Но за пять часов и одну минуту с меня уже возьмут 18 долларов, как и за 12 и за все 24 часа?

Билл Именно так.

Тони А за 24 часа и одну минуту? 30 или 36 долларов?

Билл 36, разумеется.

Филлис Как насчет недельных максимумов? Они на парковке с доставкой применяются?

Билл Нет, о парковке с доставкой я рассказал всё.

Тони Ладно, дайте-ка я запишу это в виде примеров.

Тони рисует табл. 1.1, в которой представлены обсужденные примеры, и называет ее «Парковка с доставкой в назначенное место».

Филлис Эти примеры покрывают всё, что имеет отношение к парковке с доставкой?

Билл Да, всё, о чем мы говорили, отражено точно.

Таблица 1.1. Первый вариант примеров для парковки с доставкой в назначенное место

Время парковки	Стоимость
30 минут	$12,00
3 часа	$12,00
5 часов	$12,00
5 часов 1 минута	$18,00
12 часов	$18,00
24 часа	$18,00
1 сутки 1 минута	$36,00
3 суток	$54,00
1 неделя	$126,00

Краткосрочная парковка

Филлис Ну а для других парковок как рассчитывается стоимость? Ты говорил о трех типах.

Билл Еще мы предлагаем краткосрочные парковочные места для водителей, которые подвозят и встречают пассажиров.

Филлис И сколько это стоит?

Билл За первый час мы берем 2 доллара. И по одному доллару за каждые последующие полчаса.

Тони А есть какие-нибудь ограничения, например, максимальное время парковки?

Билл Нет. Но за сутки мы берем не более 24 долларов.

Филлис Таким образом, максимальная плата за одни сутки оставляет 24 доллара?

Билл Точно.

Тони А по истечении первых суток начисляется 2 доллара за первый час следующих или стоимость рассчитывается за каждые полчаса?

Билл Хороший вопрос. За сутки и полчаса начисляется 25 долларов.

Филлис А как насчет недельного максимума? Существует такой?

Билл Нет. На краткосрочной парковке люди не оставляют машину на неделю, слишком дорого получается. Третий вариант куда привлекательнее.

Тони Прекрасно, что ты думаешь об этой табличке?

Тони рисует табл. 1.2 и передает ее Биллу и Филлис.

Таблица 1.2. Первый вариант примеров для краткосрочной парковки

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
1 час 30 минут	$3,00
2 часа	$4,00
3 часа 30 минут	$7,00
12 часов	$24,00
12 часов 30 минут	$24,00
1 сутки 30 минут	$25,00
1 сутки 1 час	$26,00

Филлис Всё, как я говорил, вопросов нет.

Экономичная и длительная парковка

Филлис Ну а что с третьим типом парковки?

Билл Есть еще экономичная парковка. Она расположена довольно далеко от аэропорта, поэтому и стоит дешевле. Между парковкой и терминалом ходит автобус.

Филлис И насколько она дешевле?

Билл Тут правила посложнее. Во-первых, парковка стоит 2 доллара в час.

Тони В любой день? Или для выходных предусмотрен другой тариф?

Билл Нет, день недели не имеет значения.

Тони Значит, что первые 30 минут, что первые 60 минут на экономичной парковке стоят ровно 2 доллара?

Билл Так и есть.

Филлис Вроде бы ничего сложного. Три часа, надо полагать, стоят 6 долларов, десять часов – 20 долларов.

Филлис И да, и нет. Максимальная плата за сутки составляет 9 долларов. Это означает, что часы с первого по четвертый оплачиваются по 2 доллара. За пятый час взимается еще один доллар, за последующие часы плата не начисляется вплоть до следующего дня.

Тони Стало быть, полчаса и час стоят 2 доллара, три часа – 6 долларов, четыре часа – 8 долларов, пять часов – 9 долларов, шесть часов – 9 долларов, 24 часа – 9 долларов...

Билл Да, всё так.

Тони А что происходит, начиная со вторых суток? Прибавляем 2 доллара или 9 долларов за каждые сутки?

Билл Нет, снова начисляем по 2 доллара за каждый час, пока не дойдем до суточного максимума 9 долларов.

Тони Таким образом, плата за сутки и полчаса составит 11 долларов, а за сутки и пять часов – 18 долларов. И точно так же на третий, четвертый, пятый день и так далее?

Билл Да, но есть еще одно ограничение. Недельный максимум составляет 54 доллара. Иначе говоря, седьмой день обходится бесплатно.

Тони Отлично, всё понял. Дай-ка подведу итог. Вот какую табличку я нарисовал, пока мы говорили.

Тони показывает Биллу и Филлис табл. 1.3, которую назвал «Экономичная парковка».

Таблица 1.3. Первый вариант примеров для экономичной парковки

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
3 часа	$6,00
4 часа	$8,00
5 часов	$9,00
24 часа	$9,00
1 сутки 1 час	$11,00
1 сутки 3 часа	$15,00
1 сутки 5 часов	$18,00
3 суток	$27,00
6 суток	$54,00
7 суток	$54,00
1 неделя и 2 дня	$72,00
3 недели	$162,00

Билл Да, всё как будто правильно.

Филлис Минутку, Билл. А как насчет шести дней и одного часа? Получится 56 или 54 доллара?

Билл Будет 54 доллара, потому что седьмые сутки бесплатны. Но, может быть, имеет смысл включить и этот пример.

Тони Уже включил.

Филлис Отлично, это всё о стоимости парковки?

Билл Нет, есть два варианта экономичной парковки. Длительная стоянка в гараже оплачивается из расчета 2 доллара в час, но не более 12 долларов в сутки, и седьмые сутки тоже бесплатны. Длительная стоянка под открытым небом стоит 2 доллара в час, но не более 10 долларов в сутки, седьмые сутки опять-таки бесплатны.

Тони Вот эти две таблицы правильны?

Тони рисует табл. 1.4 и 1.5 и показывает их Биллу и Филлис.

Таблица 1.4. Первый вариант примеров для длительной парковки в гараже

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
3 часа	$6,00
6 часов	$12,00
7 часов	$12,00
24 часа	$12,00
1 сутки 1 час	$14,00
1 сутки 3 часа	$18,00
1 сутки 7 часов	$24,00
3 суток	$36,00
6 суток	$72,00
6 суток 1 час	$72,00
7 суток	$72,00
1 неделя и 2 дня	$96,00
3 недели	$216,00

Таблица 1.5. Первый вариант примеров для длительной парковки под открытым небом

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
3 часа	$6,00

Время парковки	Стоимость
4 часа	$8,00
5 часов	$10,00
6 часов	$10,00
24 часа	$10,00
1 сутки 1 час	$12,00
1 сутки 3 часа	$16,00
1 сутки 6 часов	$20,00
3 суток	$30,00
6 суток	$60,00
6 суток 1 час	$60,00
7 суток	$60,00
1 неделя и 2 дня	$80,00
3 недели	$180,00

Тони Да, вроде всё нормально.

Филлис Еще вопрос. Возьмем пример с 24-часовой парковкой. Что произойдет, если я приеду в 11 вечера и оставлю машину на сутки до 11 часов вечера следующего дня? Получится один и второй день, каждый по 10 долларов, итого 20 долларов?

Билл Нет, мы начислим только 10 долларов, так как общее время стоянки составило 24 часа.

Тони Точно так же обрабатываются несколько дней пребывания на любой парковке?

Билл Точно так же. Граница суток значения не имеет, только разбиение общего времени стоянки на 24-часовые отрезки.

Существенные примеры

Тони Мы почти закончили. Осталось сделать еще один шаг. Думаю, все мы понимаем требования, но хотелось бы сократить количество примеров, оставив только самую суть бизнес-правил. Давайте посмотрим на таблицы еще разок и решим, что можно и нужно убрать.

Билл Хорошо, пойдем назад. Я бы хотел убрать несколько примеров для длительной стоянки под открытым небом.

Билл вычеркивает несколько примеров из таблицы, относящейся к длительной стоянке под открытым небом. Результат показан в табл. 1.6.

Таблица 1.6. Примеры для длительной парковки под открытым небом после того, как Билл вычеркнул избыточные

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
~~3 часа~~	~~$6,00~~
~~4 часа~~	~~$8,00~~
5 часов	$10,00
24 часа	$10,00
1 сутки 1 час	$12,00
1 сутки 3 часа	$16,00
1 сутки 6 часов	$20,00
3 суток	$30,00
6 суток	$60,00
6 суток 1 час	$60,00
7 суток	$60,00
1 неделя и 2 дня	$80,00
3 недели	$180,00

Филлис А что с примером, где трое суток? У нас уже есть пример для одних и шести суток. Может, этот тоже вычеркнуть?

Тони Да, пожалуй. Билл, что ты думаешь?

Билл Давай, вычеркивай. В остальных и так все отражено. Думаю, вполне можем без него обойтись.

В табл. 1.7 показаны окончательные примеры.

Таблица 1.7. Примеры для длительной парковки под открытым небом после того, как Билл вычеркнул избыточные

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
~~3 часа~~	~~$6,00~~
~~4 часа~~	~~$8,00~~
5 часов	$10,00
6 часов	$10,00
24 часа	$10,00
1 сутки 1 час	$12,00
1 сутки 3 часа	$16,00

Время парковки	Стоимость
1 сутки 6 часов	$20,00
~~3 суток~~	~~$30,00~~
6 суток	$60,00
6 суток 1 час	$60,00
7 суток	$60,00
1 неделя и 2 дня	$80,00
3 недели	$180,00

Билл Что касается длительной парковки в гараже, я думаю, будет безопасно убрать пример, где трое суток.

После вычеркивания нескольких примеров из табл. 1.4 получается табл. 1.8.

Таблица 1.8. Примеры для длительной парковки в гараже после удаления избыточных

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
3 часа	$6,00
6 часов	$12,00
7 часов	$12,00
24 часа	$12,00
1 сутки 1 час	$14,00
1 сутки 3 часа	$18,00
1 сутки 7 часов	$24,00
~~3 суток~~	~~$36,00~~
6 суток	$72,00
6 суток 1 час	$72,00
7 суток	$72,00
1 неделя и 2 дня	$96,00
3 недели	$216,00

Билл Гм, для экономичной парковки избавимся от примера, где три часа, потому что у нас уже есть четыре часа.

Тони И пример, где трое суток, нам тоже не нужен.

Билл Действительно не нужен.

Билл еще сокращает примеры для экономичной парковки, оставляя только показанные в табл. 1.9.

Таблица 1.9. Примеры для экономичной парковки после удаления избыточных

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
~~3 часа~~	~~$6,00~~
4 часа	$8,00
5 часов	$9,00
24 часа	$9,00
1 сутки 1 час	$11,00
1 сутки 3 часа	$15,00
1 сутки 5 часов	$18,00
~~3 суток~~	~~$27,00~~
6 суток	$54,00
6 суток 1 час	$54,00
7 суток	$54,00
1 неделя и 2 дня	$72,00
3 недели	$162,00

Билл Отлично, для краткосрочной парковки можно убрать примеры, где час и тридцать минут, два часа и двенадцать часов и тридцать минут.

Тони Постой, Билл. Я думаю двенадцать часов и тридцать минут надо оставить. Он отражает суточный максимум 24 доллара.

Билл Да, ты прав. Возвращаем.

Билл вычеркивает лишние примеры для краткосрочной парковки, оставляя те, что показаны в табл. 1.10.

Таблица 1.10. Примеры для краткосрочной парковки после удаления избыточных

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
~~1 час 30 минут~~	~~$3,00~~
~~2 часа~~	~~$4,00~~
3 часа 30 минут	$7,00
12 часов	$24,00

Время парковки	Стоимость
12 часов 30 минут	$24,00
1 сутки 30 минут	$25,00
1 сутки 1 час	$26,00

Билл Ну и напоследок посмотрим на примеры для парковки с доставкой в назначенное клиентом место. Не вижу, что здесь можно было бы удалить.

Тони Согласен. Тут уже представлены все существенные бизнес-правила в том виде, как ты объяснил.

Филлис Вот и хорошо, теперь у нас, похоже, есть всё для создания пользовательских историй о парковках. Билл, Тони, спасибо.

Резюме

В этой главе мы рассмотрели процедуру согласования требований к программе на совместной встрече специалистов в предметной области, программистов и тестировщиков. Поначалу вклад Тони состоял всего из нескольких идей, но, изложив примеры в наглядном виде, он помог достичь общего понимания. Обладая специальными познаниями в области тестирования, Тони сумел принять полезное участие в обсуждении, переведя его в конкретную плоскость – и всё благодаря нарисованным им таблицам со сценариями парковки.

После того как Тони показал первую таблицу с примерами, обсуждение требований стало носить более предметный характер. Программист Филлис нашла ошибку в примерах для экономичной парковки. Она попросила уточнить, как обрабатывается случай шести часов и одной минуты. Отталкиваясь от примеров, все трое обсуждали ожидаемое поведение с точки зрения бизнеса.

Билл также смог выразить свои мысли более понятно, ясно видя, согласен он или не согласен с примерами. В случае парковки с доставкой в указанной место Билл сразу сказал, сколько будет стоить парковка в течение 24 часов и одной минуты, еще даже не видя первую нарисованную Тони таблицу. В тот момент, когда был представлен первый пример, совместная деятельность в рамках рабочей встречи по выработке спецификаций стала набирать обороты.

В ходе встречи каждый мог принять участие в обмене мнениями. Филлис изложила свое видение задачи, а Тони привнес осмысление

граничных случаев. Филлис нашла ошибку в примерах для экономичной парковки еще до написания первой строки кода. Устранение этого дефекта свелось к нескольким словам, хотя впоследствии могло бы стоить целого лишнего цикла разработки.

Тони осмыслил исходную спецификацию, услышанную от Билла. Он составил на ее основе примеры и исследовал граничные случаи, например 24 часа и одна минута, прямо на месте получив от Билла правильный ответ. Допустим, что на этой итерации какой-то вопрос остался нерассмотренным, и это было бы обнаружено разработчиками позже. В этот момент Билл мог бы находиться в командировке, и задать ему вопрос было бы невозможно. Тогда команде пришлось бы придумать собственную интерпретацию. Если бы она оказалась неверной, то ошибка могла бы остаться незамеченной до презентации заказчику или – хуже того – обнаружиться спустя несколько месяцев после сдачи в эксплуатацию.

А так Билл, специалист в предметной области, принял все существенные решения по поводу программы, определив, что выкинуть, а что оставить. Говоря, что стоимость пребывания на парковке с доставкой в течение 24 часов и одной минуты составляет 36 долларов, он подтвердил, что этот аспект ему очевиден, и тем не менее вопрос Тони высветил неявные допущения. Благодаря разносторонности участников команда очень легко смогла выработать общее понимание цели реализации.

В таблицах 1.11, 1.12, 1.13, 1.14 и 1.15 сведены окончательные результаты обсуждения. Команде не потребуется много времени, чтобы реализовать и автоматизировать эти тесты. При гибкой разработке это может произойти как на следующей итерации, так и на итерации в течение трех последующих месяцев. Поскольку участники команды обсудили требования и зафиксировали свое понимание в виде существенных примеров, то как реализовать историю в соответствии с согласованной спецификацией, будет ясно и по прошествии времени.

Таблица 1.11. Окончательные примеры для парковки с доставкой в назначенное место

Время парковки	Стоимость
30 минут	$12,00
3 часа	$12,00
5 часов	$12,00
5 часов 1 минута	$18,00
12 часов	$18,00
24 часа	$18,00
1 сутки 1 минута	$36,00
3 суток	$54,00
1 неделя	$126,00

Таблица 1.12. Окончательные примеры для краткосрочной парковки

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
3 часа 30 минут	$7,00
12 часов	$24,00
12 часов 30 минут	$24,00
1 сутки 30 минут	$25,00
1 сутки 1 час	$26,00

Таблица 1.13. Окончательные примеры для экономичной парковки

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
4 часа	$8,00
5 часов	$9,00
24 часа	$9,00
1 сутки 1 час	$11,00
1 сутки 3 часа	$15,00
1 сутки 5 часов	$18,00
6 суток	$54,00
6 суток 1 час	$54,00
7 суток	$54,00
1 неделя и 2 дня	$72,00
3 недели	$162,00

Таблица 1.14. Окончательные примеры для длительной парковки в гараже

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
3 часа	$6,00
6 часов	$12,00
7 часов	$12,00
24 часа	$12,00
1 сутки 1 час	$14,00
1 сутки 3 часа	$18,00
1 сутки 7 часов	$24,00
6 суток	$72,00
6 суток 1 час	$72,00
7 суток	$72,00
1 неделя и 2 дня	$96,00
3 недели	$216,00

Таблица 1.15. Окончательные примеры для длительной парковки под открытым небом

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
5 часов	$10,00
6 часов	$10,00
24 часа	$10,00
1 сутки 1 час	$12,00
1 сутки 3 часа	$16,00
1 сутки 6 часов	$20,00
6 суток	$60,00
6 суток 1 час	$60,00
7 суток	$60,00
1 неделя и 2 дня	$80,00
3 недели	$180,00

ГЛАВА 2. Автоматизация тестов для парковки с доставкой в указанное место

Команда решает начать с примеров для парковки с доставкой в указанное место, представленных в табл. 1.11. Для автоматизации тестов решено выбрать программу Cucumber[1], в которой для связывания данных примеров и тестируемой системы используется язык Ruby. Каждый набор тестов в Cucumber называется функционалом (feature). Каждый функционал оформляется в виде отдельного текстового файла.

Чтобы автоматизировать тест в Cucumber, нам нужны три вещи: функционал, содержащий тестовые данные, определение шагов теста для взаимодействия с тестируемым приложением и параметры среды.

Тони представляет себе общую архитектуру такой, как на рис. 2.1.

Верхний блок – это примеры, выявленные в ходе рабочей встречи. Тони собирается подать их на вход Cucumber. Для тестирования приложения Cucumber необходим некий связующий код, состоящий из определений шагов, вспомогательного кода и сторонних библиотек, например Selenium, которая управляет работой браузера.

Тони планирует создать библиотеку вспомогательного кода для тестирования калькулятора стоимости парковки и использовать ее в связующем коде для определения шагов.

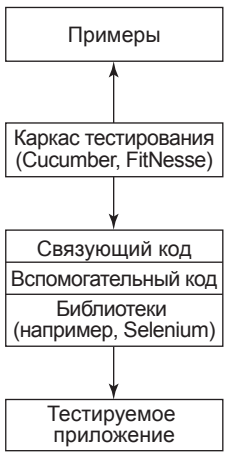


Рис. 2.1. Архитектура автоматизации тестирования, поддерживающая ATDD

1 http://cukes.info

Для автоматизации будет использоваться библиотека Selenium[2], которая позволяет строить взаимодействие с браузером на основе кода автоматизации. С ее помощью код автоматизации может взаимодействовать с веб-страницами и проверять правильность значений, например рассчитанной стоимости парковки. Команда настроила систему непрерывной интеграции с сервером Selenium без монитора[3], на котором во время сборки будут прогоняться тесты.

Первый пример

Тони решает начать с примера 30-минутной стоянки. Он описывает функциональность парковки с доставкой в файле `Valet.feature`. Первый тест следует таблице, составленной в ходе рабочей встречи (листинг 2.1).

Листинг 2.1. Первый тест для парковки с доставкой в указанное место

```
1  Feature: функциональность парковки с доставкой в указанное место
2    Калькулятор стоимости парковки вычисляет стоимость стоянки на
     парковке с доставкой в указанное место.
3
4  Scenario: Calculate Valet Parking Cost for half an hour
5    When I park my car in the Valet Parking Lot for 30 minutes
6    Then I will have to pay $ 12.00
```

Таким образом, у нас есть первый тест для стоянки на парковке с доставкой в течение 30 минут. Как было установлено на рабочей встрече, ожидаемая стоимость составляет 12,00 долларов. В первой строчке листинга 12.1 указывается имя тестируемой функции. Во второй строчке эта функция описывается на естественном языке. Cucumber напечатает это описание на консоли. Обычно Тони в этом описании сообщает автору будущего теста, что он хотел протестировать, памятуя о том, что через несколько месяцев этим автором может оказаться он сам.

В строке 4 Тони назвал свой тест «Расчет стоимости 30-минутной стоянки на парковке с доставкой». Название отражает смысл теста. В строках 5 и 6 используются ключевые слова *When* и *Then*, выражающие два этапа одного теста.

После слова When описываются действия, которые необходимо предпринять для тестирования конкретной функциональности системы. Это может быть вызов какой-то функции или нажатие кнопки.

2 http://seleniumhq.org

3 Сервер Selenium без монитора запускает браузер на виртуальном сервере. С его помощью можно прогнать все тесты на машине без монитора.

Название парковки и время стоянки – параметры ключевого слова When. Cucumber выделит их и подаст на вход системы, чтобы приложение могло вычислить стоимость парковки.

После ключевого слова Then располагаются постусловия, которые должны выполняться после прогона теста. Здесь описывается ожидаемый результат работы приложения. В данном случае Тони включил проверку рассчитанной стоимости парковки.

Теперь Тони сохраняет файл `Valet.feature` и прогоняет его через Cucumber, выполняя команду `cucumber Valet.feature`. В ответ он получает распечатку, показанную в листинге 2.2. В ней приведен только что введенный Тони сценарий. В строке 8 говорится, сколько сценариев пытался прогнать Cucumber (один) и сколько из них оказались не определены (один). В строке 9 Cucumber сообщает, что нашел два неопределенных шага. Начиная со строки 12, Cucumber дает рекомендации о том, как реализовать отсутствующие шаги[4].

Листинг 2.2. Вывод, полученный в результате прогона первого теста для парковки с доставкой в указанное место

```
1  Feature: функциональность парковки с доставкой в указанное место
2    Калькулятор стоимости парковки вычисляет стоимость стоянки на
     парковке с доставкой в указанное место.
3
4  Scenario: Calculate Valet Parking Cost
       # Valet.feature:4
5    When I park my car in the Valet Parking Lot for 30 minutes
       # Valet.feature:5
6    Then I will have to pay $ 12.00
       # Valet.feature:6
7
8  1 scenario (1 undefined)
9  2 steps (2 undefined)
10 0m0.001s
11
12 Определения для неопределенных шагов можно реализовать с помощью
     следующих фрагментов:
13
14 When /^I park my car in the Valet Parking Lot for (\d+)
       minutes$/ do |arg1|
15   pending # оформите нужное вам утверждение в виде регулярного
       выражения, как показано выше
16 end
17
18 Then /^I will have to pay \$ (\d+)\.(\d+)$/ do |arg1, arg2|
```

4 Cucumber печатает рекомендации на английском языке. Для удобства читателя они частично переведены. – *Прим. перев.*

```
19  pending # оформите нужное вам утверждение в виде регулярного
      выражения, как показано выше
20 end
21
22 Если вы хотите записать фрагменты на другом языке программирования,
     подготовьте файл
23 с подходящим расширением, в котором cucumber будет искать
     определения шагов.
```

По совету Cucumber Тони создает файл с определениями шагов, скопировав предложенные фрагменты из оболочки в новый файл с именем `Valet steps.rb`. Чтобы отделить тестовые данные от связующего кода, который выдает команды тестируемой системе, Тони помещает вспомогательный код в файл, находящийся в новой папке `step_definitions`.

Тони немного изменяет предложенные заглушки методов. Это окажется полезно, когда впоследствии он расширит первый пример, добавив другие значения времени стоянки. Результат приведен в листинге 2.3.

Cucumber умеет выделять переменные из текста. Тони воспользовался этим, чтобы задать время и стоимость. Cucumber распознает ключевое слово `pending` и сообщает, что этот тест отложен и, вероятно, продолжает разрабатываться. При повторном прогоне теста, дополненного определениями шагов, Тони получает результат, показанный в листинге 2.4.

Листинг 2.3. Начальный вариант определения шагов для первого теста

```
1  When /^I park my car in the Valet Parking Lot for (.*)$/ do
       |duration|
2   pending
3  end
4
5  Then /^I will have to pay (.*)$/ do |price|
6   pending
7  end
```

Листинг 2.4. Результат прогона первого теста для парковки с доставкой после добавления определений шагов

```
1  Feature: функциональность парковки с доставкой в указанное место
2   Калькулятор стоимости парковки вычисляет стоимость стоянки на
     парковке с доставкой в указанное место.
3
4  Scenario: Calculate Valet Parking Cost
     # Valet.feature:4
5   When I park my car in the Valet Parking Lot for 30 minutes
```

```
      # step_definitions/Valet_steps.rb:1
6       TODO (Cucumber::Pending)
7       ./step_definitions/Valet_steps.rb:2:in '/^I park my car
   in the Valet Parking Lot for (.*)$/'
8       Valet.feature:5:in 'When I park my car in the Valet
   Parking Lot for 30 minutes'
9     Then I will have to pay $ 12.00
        # step_definitions/Valet_steps.rb:5
10
11 1 scenario (1 pending)
12 2 steps (1 skipped, 1 pending)
13 0m0.002s
```

Чтобы подвергнуть систему тестированию, Тони должен еще настроить драйвер веб-браузера, Selenium. В состав Selenium входит серверный компонент и клиентская библиотека, к которой вспомогательный код может обращаться для управления браузером и навигации по веб-страницам. Тони помещает вспомогательный код в файл `env.rb`, чтобы не смешивать его со связующим кодом, тестирующим систему. Этот файл создается в подпапке `etc`. Структура папок показана на рис. 2.2, а сам код – в листинге 2.5.

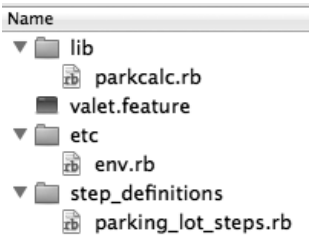


Рис. 2.2. Структура папок с указанием всех необходимых файлов

Листинг 2.5. Вспомогательный код для настройки клиента Selenium

```
1  require 'rubygems'
2  gem 'selenium-client'
3  require 'selenium/client'
4  require 'spec/expectations'
5  require 'lib/parkcalc'
6
7  # до всего остального
8  selenium_driver = Selenium::Client::Driver.new \
9    :host => 'localhost',
10   :port => 4444,
11   :browser => '*firefox',
12   :url => 'http://www.shino.de/parkcalc',
13   :timeout_in_second => 60
14  selenium_driver.start_new_browser_session
15 $parkcalc = ParkCalcPage.new(selenium_driver)
16
17 # после всего остального
18 at_exit do
```

```
19   selenium_driver.close_current_browser_session
20 end
```

Здесь мы загружаем библиотеку Selenium, запускаем браузер и открываем страницу ParkCalc. По завершении всех тестов окно браузера закрывается.

В файле `env.rb` затребуется также библиотека калькулятора стоимости парковки `lib/parkcalc`. Тони будет дописывать в нее код по мере разработки тестов. Первоначальное содержимое этого файла показано в листинге 2.6.

Листинг 2.6. Первая версия класса ParkCalcPage

```
1  class ParkCalcPage
2    attr :page
3
4    def initialize(page_handle)
5      @page = page_handle
6      @page.open '/parkcalc'
7    end
8  end
```

В инициализаторе мы сохраняем переданный описатель страницы (`page_handle`) в локальном атрибуте `page` и открываем страницу `/parkcalc`. Таким образом, в первом варианте реализации мы просто открываем веб-форму в браузере, который был настроен в файле `env.rb`.

Тони разрабатывает первый тест по шагам. На первом шаге необходимо реализовать ввод времени стоянки в веб-интерфейсе. Тони еще не знает о конкретных деталях реализации, но предварительные нарисованные от руки эскизы верстки он видел. В условии `When I park my car in the Valet Parking Lot for <duration>` содержатся два шага. Во-первых, необходимо выбрать парковку, а, во-вторых, ввести значение, сопоставляемое со временем стоянки. Тони формулирует пожелание, которое считает разумным, и прописывает в файле `Valet steps.rb` тот API, который хотел бы видеть в этом случае (см. листинг 2.7).

Листинг 2.7. Первоначальная реализация ключевого слова When, основанная на предположениях о структуре формы

```
1  When /^I park my car in the Valet Parking Lot for (.*)$/ do
       |duration|
2    $parkcalc.select('Valet Parking')
3    $parkcalc.enter_parking_duration(duration)
4    pending
5 end
```

Первый шаг описан в строке 2 листинга 2.7. Тони решил, что название парковки будет задаваться. В зависимости от особенностей реализации это может быть ввод строки в текстовое поле, выбор парковки из комбинированного списка или из раскрывающегося меню. Поскольку конкретного механизма Тони еще не знает, он откладывает решение до реализации класса ParkCalcPage.

Второй шаг, представленный строкой 3, подразумевает, что каким-то образом должно быть введено время стоянки. Тут тоже есть разные варианты, например: ввести значение в текстовое поле или задать время начала и окончания в двух полях даты, а затем вычислить продолжительность. Поскольку вопрос о пользовательском интерфейсе пока открыт, Тони откладывает решение о том, как это будет реализовано в классе ParkCalcPage.

Чтобы показать, что определение шага теста еще не закончено, Тони оставляет в конце ключевое слово `pending`. Это напоминание тому, кто будет реализовывать условие, о необходимости изменить определение шага после реализации двух других ключевых слов.

Далее Тони информирует будущего разработчика класса ParkCalcPage о принятых им решениях касательно интерфейса. Для этого он включает в класс пустые реализации двух методов, которые должны присутствовать в классе (листинг 2.8).

Листинг 2.8. Пустые реализации методов, добавленные в класс ParkCalcPage

```
1  class ParkCalcPage
2    attr :page
3
4    def initialize(page_handle)
5      @page = page_handle
6      @page.open '/parkcalc'
7    end
8
9    def select(parking_lot)
10   end
11
12   def enter_parking_duration(duration)
13   end
14
15 end
```

Добавлены методы `select(parking_lot)` и `enter_parking_duration(duration)`. Первый в будущем будет выбирать парковку, а второй отвечает за получение от пользовательского интерфейса времени стоянки.

Теперь Тони может сосредоточиться на реализации шага верификации. Как и ранее, Тони делает предположение о том, как будет выражена верификация стоимости парковки в окончательном варианте. В листинге 2.9 показаны изменения в файле `step_definitions/Valet_steps.rb`, а в листинге 2.10 – модифицированный класс ParkCalcPage.

Листинг 2.9. Первоначальная, основанная на предположении реализация верификации

```
1  Then /^I will have to pay (.*)$/ do |price|
2    $parkcalc.parking_costs.should == price
3    pending
4  end
```

Листинг 2.10. Пустая реализация шага вычисления стоимости парковки

```
1  def parking_costs
2    return nil
3  end
```

Тони закончил первый тест в той части, которая отнесена к его компетенции. Теперь надо решить, что делать дальше. Можно было бы включить остальные примеры для парковки с доставкой в указанное место, согласованные на рабочей встрече. С другой стороны, можно составить пару с разработчиком и вместе автоматизировать примеры данной функциональности. Третий вариант – заняться остальными видами парковок и закончить первый тест в Cucumber. Тони решает объединиться с разработчиком, чтобы реализовать первый функционал парковки с доставкой и автоматизировать первый тест. Тогда он сможет узнать мнение коллег о проделанной работе и позже перейти к разработке других тестов. У этого подхода есть и еще одно достоинство – к началу разработки не накопится слишком много безуспешных тестов.

Парная разработка первого теста

Чтобы реализовать и автоматизировать первый тест, Тони объединяется в пару с Алексом. Алекс уже сверстал начерно веб-сайт и рассказывает о своих идеях Тони.

Алекс Привет, Тони, хочешь посмотреть, как продвигаются дела с калькулятором стоимости парковки?

Тони Вообще-то я собирался в паре с тобой поработать над автоматизацией первого теста.

Алекс Вот и славненько. С чего ты начал?

Тони С парковки с доставкой. Я подготовил первый тест для 30-минутной стоянки и перед тем, как подойти к тебе, записал его в систему управления версиями.

Алекс Прекрасно, теперь глянь, к чему я пришел.

Алекс показывает Тони эскиз разработанной им веб-страницы (рис. 2.3).

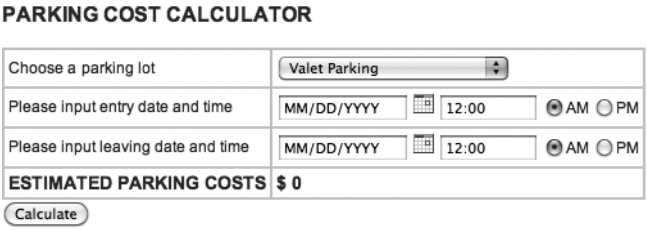


Рис. 2.3. Макет страницы калькулятора стоимости парковки, подготовленный Алексом

Инициализация

Алекс Я решил, что тип парковки будет выбираться из раскрывающегося списка. Дату можно ввести прямо в текстовое поле или выбрать из календаря. Время прибытия и убытия вводится в виде текста, а с помощью переключателя задается признак «до или после полудня». Расчетная стоимость парковки показывается после нажатия кнопки Calculate.

Тони Выглядит неплохо. У меня тут несколько шагов помечены как отложенные, и теперь самое время их подцепить. Взгляни.

Алекс Ну вроде ничего сложного. Давай для начала выберем парковку из списка. Я назвал список «ParkingLot». Поэтому выбор значения из списка – это один шаг, вот такой.

Алекс реализует метод `select` в файле `lib/parkcalc.rb`, как показано в листинге 2.11.

Листинг 2.11. Выбор парковки из выпадающего списка

```
1  def select(parking_lot)
2    @page.select 'ParkingLot', parking_lot
3  end
```

Тони Ну что ж, всё интуитивно понятно. Выбираю переданный параметр из элемента с идентификатором «ParkingLot». Чудненько. А как насчет ввода времени стоянки?

Алекс Дай подумать. А воспользуемся-ка мы для этой цели хешем Ruby. Потом можно будет обобщить это решение для ввода даты и времени прибытия и убытия и, стало быть, смоделировать все необходимые тебе интервалы. Если я правильно помню примеры, которые вы утвердили на совещании, то интервалов понадобится много.

Тони И как мы это сделаем?

Алекс Идея в том, чтобы по переданной в эту функцию строке продолжительности стоянки найти фактические значения даты и времени (вместе с признаком AM/PM) прибытия и убытия и передать их веб-странице. В совокупности это будут шесть значений в моей форме. Но давай для начала определим хеш.

Алекс помещает определение хеша продолжительностей `durationMap` в начало класса ParkCalcPage (листинг 2.12).

Тони Два знака @ говорят, что `durationMap` – переменная класса. Верно?

Листинг 2.12. Хеш отображает продолжительность стоянки на фактические дату и время

```
1  @@durationMap = {
2    '30 minutes' => ['05/04/2010', '12:00', 'AM', '05/04/2010'
     , '12:30', 'AM']
3  }
```

Алекс Верно. Теперь мы воспользуемся этим хешем, чтобы получить интересующие нас шесть значений. Давай покажу, как извлекаются данные из хеша.

Алекс начинает писать функцию `enter_parking_duration` (листинг 2.13).

Листинг 2.13. Получение из `durationMap` шести параметров для формы

```
1  def enter_parking_duration(duration)
2    startingDate, startingTime, startingTimeAMPM, leavingDate,
       leavingTime, leavingTimeAMPM = @@durationMap[duration]
3  end
```

Алекс А теперь подсунем эти шесть значений форме. Начнем с даты и времени прибытия.

Алекс дописывает код ввода продолжительности стоянки (листинг 2.14).

Листинг 2.14. Дата и время прибытия помещены в нужные поля формы

```
1  def enter_parking_duration(duration)
2    startingDate, startingTime, startingTimeAMPM, leavingDate,
       leavingTime, leavingTimeAMPM = @@durationMap[duration]
3    @page.type 'StartingDate', startingDate
4    @page.type 'StartingTime', startingTime
5    @page.click "//input[@name='StartingTimeAMPM' and @value
     ='%s']" % startingTimeAMPM
6  end
```

Тони Объясни, будь добр. Что-то я не понимаю последнюю строчку.

Алекс Объясняю. Сначала мы получаем из хеша шесть параметров, соответствующих переданному ключу – продолжительности стоянки. А затем помещаем их в поля даты и времени прибытия.

Тони Да, это мне понятно. Но что это за абракадабра в последней строчке?

Алекс Так я нахожу значения переключателей. Это запрос на языке xpath, описывающий, где находится элемент «переключатель» и каково его значение. Он говорит драйверу, что нужно щелкнуть по элементу ввода, для которого атрибут name равен «StartingTimeAMPM», а значение совпадает с переданным.

Тони Я бы поместил это куда-нибудь еще. Мне кажется, что это слишком техническая деталь, которой не место в более абстрактном методе.

Алекс Пожалуй, ты прав. Я помечу это у себя, а пока давай закончим с этой функцией. Мы еще не заполнили дату и время убытия. Тут всё так же, как с прибытием. Но сначала посмотрим, работает ли то, что мы уже написали.

Алекс запускает тест, и оба смотрят, как появляется окно браузера, открывается страница калькулятора и устанавливается тип парковки и дата и время прибытия. Затем окно браузера закрывается и печатается результат.

Тони Всё нормально. Продолжим, нам еще нужно заполнить дату и время убытия.

Алекс Само собой. Код аналогичен тому, что уже написан для даты и времени прибытия. Просто скопируем его и заменим

переменные. А когда убедимся, что все работает как надо, подчистим.

Алекс добавляет в метод код заполнения даты и времени убытия (листинг 2.15).

Листинг 2.15. В ранее написанную функцию добавляется код заполнения даты и времени убытия с парковки

```
1  def enter_parking_duration(duration)
2    startingDate, startingTime, startingTimeAMPM, leavingDate,
       leavingTime, leavingTimeAMPM = @@durationMap[duration]
3    @page.type 'StartingDate', startingDate
4    @page.type 'StartingTime', startingTime
5    @page.click "//input[@name='StartingTimeAMPM' and @value
     ='%s']" % startingTimeAMPM
6
7    @page.type 'LeavingDate', leavingDate
8    @page.type 'LeavingTime', leavingTime
9    @page.click "//input[@name='LeavingTimeAMPM' and @value='%
     s']" % leavingTimeAMPM
10 end
```

Тони и Алекс прогоняют эти шаги и убеждаются, что дата и время убытия заполнены правильно.

Алекс Отлично, работает. Теперь немного подчистим. Оба блока очень похожи. Давай заведем для них отдельный метод.

Алекс и Тони создают новый метод, который умеет заполнять дату и время как прибытия, так и убытия, а затем заменяют прежние блоки обращениями к новому методу. После каждого изменения они проверяют, что тест по-прежнему работает правильно. В результате получается код, приведенный в листинге 2.16.

Листинг 2.16. Отдельный метод для заполнения полей даты и времени

```
1  def enter_parking_duration(duration)
2    startingDate, startingTime, startingTimeAMPM, leavingDate,
       leavingTime, leavingTimeAMPM = @@durationMap[duration]
3    fill_in_date_and_time_for 'Starting', startingDate,
       startingTime, startingTimeAMPM
4    fill_in_date_and_time_for 'Leaving', leavingDate,
       leavingTime, leavingTimeAMPM
5  end
6
7  def fill_in_date_and_time_for(formPrefix, date, time, ampm)
8    @page.type "%sDate" % formPrefix, date
9    @page.type "%sTime" % formPrefix, time
10   @page.click "//input[@name='%sTimeAMPM' and @value='%s']"
```

```
      % [ formPrefix, ampm ]
11 end
```

Алекс Так, теперь давай вынесем этот малопонятный фрагмент с выражением на xpath.

Тони Объявим константное выражение?

Алекс Именно это я и собирался сделать. И заодно я хочу поместить в переменные другие константные строки, чтобы в будущем их было проще изменять. Давай по одной. Сначала избавимся от xpath. Переменную нужно как-то назвать. Есть идеи?

Тони Как насчет `amPMRadioButtonTemplate`?

Алекс По мне так нормально. Тогда переменные для времени и даты назовем `timeTemplate` и `dateTemplate`?

Тони Годится. И давай еще префиксы добавим – `startingPrefix` и `leavingPrefix`.

Алекс Договорились. И заведем для идентификатора парковки отдельную константу `lotIdentifier`.

Тони Вот теперь совсем хорошо.

В листинге 2.17 приведена окончательная версия класса ParkCalcPage после того как Алекс с Тони вынесли константы. Теперь все шаги инициализации готовы.

Листинг 2.17. Окончательная версия ParkCalcPage с шагами инициализации

```
1  class ParkCalcPage
2
3    @@lotIdentifier = 'ParkingLot'
4    @@startingPrefix = 'Starting'
5    @@leavingPrefix = 'Leaving'
6    @@dateTemplate = "%sDate"
7    @@timeTemplate = "%sTime"
8    @@amPMRadioButtonTemplate = "//input[@name='%sTimeAMPM' and
       @value='%s']"
9
10   @@durationMap = {
11     '30 minutes' => ['05/04/2010', '12:00', 'AM', '05/04/2010'
       , '12:30', 'AM']
12 }
13
14   attr :page
15
16   def initialize(page_handle)
17     @page = page_handle
18     @page.open '/parkcalc'
19   end
```

```
20
21   def select(parking_lot)
22     @page.select @@lotIdentifier, parking_lot
23   end
24
25   def enter_parking_duration(duration)
26     startingDate, startingTime, startingTimeAMPM, leavingDate,
         leavingTime, leavingTimeAMPM = @@durationMap[duration]
27   fill_in_date_and_time_for @@startingPrefix, startingDate,
         startingTime, startingTimeAMPM
28   fill_in_date_and_time_for @@leavingPrefix, leavingDate,
         leavingTime, leavingTimeAMPM
29   end
30
31   def fill_in_date_and_time_for(formPrefix, date, time, ampm)
32     @page.type @@dateTemplate % formPrefix, date
33     @page.type @@timeTemplate % formPrefix, time
34     @page.click @@amPMRadioButtonTemplate % [ formPrefix,
         ampm ]
35   end
36
37 end
```

Проверка результатов

Тони Давай проверим, что получилось. Мы еще не нажимали кнопку Calculate и надо придумать, как получить от страницы вычисленную стоимость парковки.

Алекс Само собой. Давай уберем из определения шага слово `pending`, чтобы выполнялось определение Then.

Тони Ой, совсем забыл про это. Этак я бы полдня потратил на поиск причины.

Алекс Ну так для того мы и работаем в паре, правда?

Тони Заодно давай уж уберем `pending` из проверки, которую реализуем чуть позже.

Алекс Ты прав. Теперь посмотрим, как проверить правильность расчета стоимости. Сначала нужно нажать кнопку Calculate. Я хочу добавить это действие в функцию `parking_costs` как первый шаг перед возвратом стоимости. Потом нужно будет дождаться загрузки страницы с результатом. А потом мы просто найдем и вернем элемент, содержащий значение стоимости.

Алекс реализует функцию `parking_costs`, как показано в листинге 2.18.

Листинг 2.18. Первоначальная версия кода проверки

```
1  def parking_costs
2    @page.click 'Submit'
3    @page.wait_for_page_to_load 10000
4    cost_element = @page.get_text "//tr[td/div[@class='SubHead']
       = 'estimated Parking costs']/td/span/b"
5    return cost_element
6  end
```

Тони А что здесь означает константа 10000?

Алекс А это величина таймаута. Драйвер будет ждать загрузки новой страницы 10 секунд, а если она не загрузится, то будет считаться, что тест не прошел.

Тони И давай снова уберем абракадабру. Предлагаю завести для xpath константу.

Алекс Но сначала посмотрим, правильно ли работает. Давай прогоним тест.

Алекс и Тони запускают тест и наблюдают, как вызывается калькулятор стоимости парковки, а на консоли появляется зеленая полоса, означающая, что все прошло хорошо (см. листинг 2.19).

Листинг 2.19. На консоли печатается результат первого теста для парковки с доставкой

```
1  Feature: функциональность парковки с доставкой в указанное место
2    Калькулятор стоимости парковки вычисляет стоимость стоянки на
     парковке с доставкой в указанное место.
3
4  Scenario: Calculate Valet Parking Cost
     # Valet.feature:4
5  When I park my car in the Valet Parking Lot for 30 minutes
     # step_definitions/Valet_steps.rb:1
6  Then I will have to pay $ 12.00
     # step_definitions/Valet_steps.rb:6
7
8  1 scenario (1 passed)
9  2 steps (2 passed)
10 0m0.324s
```

Тони Отлично, тест проходит. И пока мы ничего не сломали, давай сохраним его в СУВ, чтобы в случае чего можно было откатить последующие изменения. Так, на всякий случай.

Алекс Хорошая мысль.

Алекс и Тони сохраняют всё сделанное в системе управления версиями.

Алекс Вернемся к твоему предложению. Идея неплохая, но я бы хотел сначала разбить эту функцию на две. Первая будет нажимать кнопку Submit и ждать загрузки страницы. А вторая – искать и возвращать вычисленную стоимость. Сейчас покажу.

Алекс выделяет из первоначальной функции `parking_costs` два метода (листинг 2.20).

Листинг 2.20. Проверка после выделения двух функций для выполнения отдельных шагов

```
1  def parking_costs
2    calculate_parking_costs
3    get_parking_costs_from_page
4  end
5
6  def calculate_parking_costs
7    @page.click 'Submit'
8    @page.wait_for_page_to_load 10000
9  end
10
11 def get_parking_costs_from_page
12   @page.get_text "//tr[td/div[@class='SubHead'] = 'estimated
       Parking costs']/td/span/b"
13 end
```

Тони Теперь заведем константу для выражения xpath, которое описывает путь к элементу, где хранится стоимость.

Алекс И заодно запишем имя кнопки Calculate в осмысленную переменную.

Тони Теперь последний прогон, и мы сможем записать плоды наших трудов в репозиторий исходного кода.

Тони и Алекс наблюдают за выполнением теста. Он по-прежнему проходит. Они записывают файлы в репозиторий. В листинге 2.21 приведен окончательный вариант файла `step_definitions/Valet_steps.rb`, а в листинге 2.22 – окончательный код в файле `lib/parkcalc.rb`.

Листинг 2.21. Окончательный вариант шагов первого теста для парковки с доставкой

```
1  When /^I park my car in the Valet Parking Lot for (.*)$/ do
       |duration|
2    $parkcalc.select('Valet Parking')
3    $parkcalc.enter_parking_duration(duration)
4  end
```

```
5
6  Then /^I will have to pay (.*)$/ do |price|
7    $parkcalc.parking_costs.should == price
8  end
```

Листинг 2.22. Окончательный вариант класса ParkCalcPage для первого теста

```
1  class ParkCalcPage
2
3    @@lotIdentifier = 'ParkingLot'
4    @@startingPrefix = 'Starting'
5    @@leavingPrefix = 'Leaving'
6    @@dateTemplate = "%sDate"
7    @@timeTemplate = "%sTime"
8    @@amPMRadioButtonTemplate = "//input[@name='%sTimeAMPM' and
       @value='%s']"
9
10   @@calculateButtonIdentifier = 'Submit'
11   @@costElementLocation = "//tr[td/div[@class='SubHead']
       = 'estimated Parking costs']/td/span/b"
12
13   @@durationMap = {
       14 '30 minutes' => ['05/04/2010', '12:00', 'AM', '05/04/2010'
       , '12:30', 'AM']
15   }
16
17   attr :page
18
19   def initialize(page_handle)
20     @page = page_handle
21     @page.open '/parkcalc'
22   end
23
24   def select(parking_lot)
25     @page.select @@lotIdentifier, parking_lot
26   end
27
28   def enter_parking_duration(duration)
29     startingDate, startingTime, startingTimeAMPM, leavingDate,
         leavingTime, leavingTimeAMPM = @@durationMap[duration]
30     fill_in_date_and_time_for @@startingPrefix, startingDate,
         startingTime, startingTimeAMPM
31     fill_in_date_and_time_for @@leavingPrefix, leavingDate,
         leavingTime, leavingTimeAMPM
32   end
33
34   def fill_in_date_and_time_for(formPrefix, date, time, ampm)
35     @page.type @@dateTemplate % formPrefix, date
36     @page.type @@timeTemplate % formPrefix, time
```

```
37    @page.click @@amPMRadioButtonTemplate % [ formPrefix,
        ampm ]
38   end
39
40   def parking_costs
41    calculate_parking_costs
42    get_parking_costs_from_page
43   end
44
45   def calculate_parking_costs
46    @page.click @@calculateButtonIdentifier
47    @page.wait_for_page_to_load 10000
48   end
49
50   def get_parking_costs_from_page
51    @page.get_text @@costElementLocation
52   end
53 end
```

Табличные тесты

Автоматизировав первый тест, Тони без труда может повторно использовать уже написанные шаги для автоматизации остальных примеров, рассмотренных в ходе рабочей встречи. Для начала он указывает, что файл `Valet.feature` следует рассматривать как шаблон сценария, а затем организует примеры в виде таблицы. Для этого он заменяет продолжительность 30 минут маркером `<parking duration>`, а ожидаемую стоимость – маркером `<parking costs>`. Затем Тони помещает в таблицу под шаблоном сценария конкретные значения, взятые из примеров. Столбцы таблицы помечаются именами маркером. Результат показан в листинге 2.23.

Листинг 2.23. Первый тест, преобразованный в табличный формат

```
1  Feature: функциональность парковки с доставкой в указанное место
2    Калькулятор стоимости парковки вычисляет стоимость стоянки на
     парковке с доставкой в указанное место.
3
4  Scenario Outline: Calculate Valet Parking Cost
5  When I park my car in the Valet Parking Lot for <parking
     duration>
6  Then I will have to pay <parking costs>
7
8  Examples:
9  | parking duration | parking costs |
10 | 30 minutes       | $ 12.00       |
```

Здесь Тони начал преобразовывать примеры с рабочей встречи в табличный формат – в самом буквальном смысле. Тони прогоняет тест, чтобы убедиться, что всё по-прежнему работает правильно. Результат показан в листинге 2.24.

Листинг 2.24. Вывод на консоль результатов первого теста в табличном формате

```
1  Feature: функциональность парковки с доставкой в указанное место
2    Калькулятор стоимости парковки вычисляет стоимость стоянки на
     парковке с доставкой в указанное место.
3
4  Scenario Outline: Calculate Valet Parking Cost
     # Valet.feature:4
5  When I park my car in the Valet Parking Lot for <parking
     duration>  # step_definitions/Valet_steps.rb:1
6  Then I will have to pay <parking costs>
              # step_definitions/Valet_steps.rb:6
7
8  Examples:
9  | parking duration | parking costs |
10 | 30 minutes       | $ 12.00       |
11
12 1 scenario (1 passed)
13 2 steps (2 passed)
14 0m0.316s
```

Далее он вводит данные из остальных примеров. В конечном итоге все выработанные на рабочей встрече примеры будут представлены в виде табличного теста (листинг 2.25).

Листинг 2.25. Все примеры с рабочей встречи в виде таблицы

```
1  Feature: функциональность парковки с доставкой в указанное место
2    Калькулятор стоимости парковки вычисляет стоимость стоянки на
     парковке с доставкой в указанное место.
3
4  Scenario Outline: Calculate Valet Parking Cost
5  When I park my car in the Valet Parking Lot for <parking
     duration>
6  Then I will have to pay <parking costs>
7
8  Examples:
9  | parking duration | parking costs |
10 | 30 minutes       | $ 12.00       |
11 | 3 hours          | $ 12.00       |
12 | 5 hours          | $ 12.00       |
13 | 5 hours 1 minute | $ 18.00       |
14 | 12 hours         | $ 18.00       |
```

```
15 | 24 hours          | $ 18.00       |
16 | 1 day 1 minute    | $ 36.00       |
17 | 3 days            | $ 54.00       |
18 | 1 week            | $ 126.00      |
```

Чтобы выполнить все эти тесты, Тони должен записать в хеш `durationMap` в классе ParkCalcPage соответствующие значения (листинг 2.26).

Листинг 2.26. Класс ParkCalcPage, в котором хеш durationMap дополнен значениями для всех тестов парковки с доставкой в указанное место

```
1  class ParkCalcPage
2
3  ...
4
5  @@durationMap = {
6    '30 minutes' => ['05/04/2010', '12:00', 'AM', '05/04/2010'
       ,'12:30', 'AM'],
7    '3 hours' => ['05/04/2010', '12:00', 'AM', '05/04/2010',
       '03:00', 'AM'],
8    '5 hours' => ['05/04/2010', '12:00', 'AM', '05/04/2010',
       '05:00', 'AM'],
9    '5 hours 1 minute' => ['05/04/2010', '12:00', 'AM',
       '05/04/2010', '05:01', 'AM'],
10   '12 hours' => ['05/04/2010', '12:00', 'AM', '05/04/2010',
       '12:00', 'PM'],
11   '24 hours' => ['05/04/2010', '12:00', 'AM', '05/05/2010',
       '12:00', 'AM'],
12   '1 day 1 minute' => ['05/04/2010', '12:00', 'AM',
       '05/05/2010', '12:01', 'AM'],
13   '3 days' => ['05/04/2010', '12:00', 'AM', '05/07/2010',
       '12:00', 'AM'],
14   '1 week' => ['05/04/2010', '12:00', 'AM', '05/11/2010',
       '12:00', 'AM']
15 }
16
17 ...
```

Тони прогоняет тесты и видит, что все они проходят. Алекс, похоже, реализовал всю функциональность парковки с доставкой. Критерий приемки на обратной стороне карточки с историей содержит ответы на все вопросы, необходимые Алексу для реализации. Последний шаг Тони – запись всех измененных файлов в репозиторий исходного кода, после чего он помечает на доске задач, что история для парковки с доставкой в указанное место автоматизирована и тесты проходят. В конце рабочего дня Алекс и Тони отмечают успех, хлопая друг друга по ладони.

Резюме

На этом автоматизация примеров для парковки с доставкой в указанное место завершается. Мы видели, как Тони работает с Cucumber. Первый пример сначала был написан на естественном языке и сохранен в текстовом файле. Потом он приступил к автоматизации примера и занимался этим, пока не почувствовал, что достиг предела своей компетенции. Зайдя в тупик, Тони объединился в пару с Алексом – программистом, отвечающим за автоматизацию тестирования.

Алекс и Тони реализовали драйвер ParkCalcPage, который заполняет в веб-форме различные поля: тип парковки, а также даты и время прибытия и убытия машины. После нажатия кнопки Calculate функция возвращает вычисленную стоимость парковки, которую каркас тестирования сравнивает с ожидаемой.

Работая в паре, Алекс и Тони дополняют друг друга. Тестировщик Тони может критически осмыслить тестовый код, тогда как программист Алекс помогает Тони преодолеть технические сложности автоматизации. К тому же, вместе они помогают друг другу взглянуть на проблему под другим углом зрения. Наконец, код, написанный совместно Алексом и Тони, по определению уже подвергнут критическому анализу. Иметь напарника, который смотрит на создаваемый код, вообще очень ценно при коллективной разработке ПО – и к коллективной автоматизации тестов это тоже относится.

Преобразовав первый тест в шаблон сценария, Тони смог сразу же автоматизировать остальные тесты для парковки с доставкой. Специалист в предметной области Билл сможет найти в результатах тестов примеры, которые он вместе с Филлис и Тони подготовил в ходе рабочей встречи.

ГЛАВА 3. Автоматизация тестов для остальных типов парковок

Закончив тесты для парковки с доставкой в указанное место, Тони был уверен, что сможет автоматизировать примеры для остальных парковок за несколько часов. Так он и поступил, начав с примеров для краткосрочной парковки, затем для экономичной, долгосрочной в гараже и долгосрочной под открытым небом.

Краткосрочная парковка

В табл. 3.1 показаны примеры для краткосрочной парковки, утвержденные на рабочей встрече.

Таблица 3.1. Примеры для краткосрочной парковки

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
3 часа 30 минут	$7,00
12 часов	$24,00
12 часов 30 минут	$24,00
1 сутки 30 минут	$25,00
1 сутки 1 час	$26,00

Тони начинает автоматизировать их, взяв за основу примеры для парковки с доставкой в указанное место. Он создает файл `Short-Term.feature`, приведенный в листинге 3.1.

Листинг 3.1. Автоматизированные примеры для краткосрочной парковки

```
1  Feature: функциональность краткосрочной парковки
2    Калькулятор стоимости парковки вычисляет стоимость стоянки на
     краткосрочной парковке.
3
4  Scenario Outline: Calculate Short-Term Parking Cost
5  When I park my car in the Short-Term Parking Lot for <parking
     duration>
6  Then I will have to pay <parking costs>
7
8  Examples:
9  | parking duration    | parking costs |
10 | 30 minutes          | $ 2.00        |
11 | 1 hour              | $ 2.00        |
12 | 1 hour 30 minutes   | $ 3.00        |
13 | 2 hours             | $ 4.00        |
14 | 3 hours 30 minutes  | $ 7.00        |
15 | 12 hours 30 minutes | $ 24.00       |
16 | 1 day 30 minutes    | $ 25.00       |
17 | 1 day 1 hour        | $ 26.00       |
```

Но сразу же выясняется, что строка 5 не работает – ни одно из имеющихся определений шагов не соответствует части «Short-Term Parking». Тони смотрит на написанные ранее определения и понимает, что может повторно воспользоваться ими – нужно только добавить образец для сравнения с новым типом парковки. Поэтому он переименовывает файл `step_definitions/valet_steps.rb` в `step_definitions/parking_lot_steps.rb` и добавляет параметр `parking_lot` в определение ключевого слова When. Результат показан в листинге 3.2.

Листинг 3.2. Обобщенные определения шагов

```
1  When /^I park my car in the (.*) Lot for (.*)$/ do |
     parking_lot, duration|
2    $parkcalc.select(parking_lot)
3    $parkcalc.enter_parking_duration(duration)
4  end
5
6  Then /^I will have to pay (.*)$/ do |price|
7    $parkcalc.parking_costs.should == price
8  end
```

И чтобы довершить дело с тестами для краткосрочной парковки, Тони добавляет в определение хеша `durationMap` в классе ParkCalcPage продолжительности 1 hour, 1 hour 30 minutes, 2 hours, 3 hours

30 minutes, 12 hours 30 minutes, 1 day 30 minutes и 1 day 1 hour. Окончательный вариант хеша, находящегося в файле `lib/parkcalc.rb`, представлен в листинге 3.3.

Листинг 3.3. Хеш durationMap в классе ParkCalcPage с примерами для парковки с доставкой и для краткосрочной парковки

```
1  class ParkCalcPage
2
3  ...
4  @@durationMap = {
5    '30 minutes' => ['05/04/2010', '12:00', 'AM', '05/04/2010'
       , '12:30', 'AM'],
6    '1 hour' => ['05/04/2010', '12:00', 'AM', '05/04/2010',
       '01:00', 'AM'],
7    '1 hour 30 minutes' => ['05/04/2010', '12:00', 'AM',
       '05/04/2010', '01:30', 'AM'],
8    '2 hours' => ['05/04/2010', '12:00', 'AM', '05/04/2010',
       '02:00', 'AM'],
9    '3 hours' => ['05/04/2010', '12:00', 'AM', '05/04/2010',
       '03:00', 'AM'],
10   '3 hours 30 minutes' => ['05/04/2010', '12:00', 'AM',
       '05/04/2010', '03:30', 'AM'],
11   '5 hours' => ['05/04/2010', '12:00', 'AM', '05/04/2010',
       '05:00', 'AM'],
12   '5 hours 1 minute' => ['05/04/2010', '12:00', 'AM',
       '05/04/2010', '05:01', 'AM'],
13   '12 hours' => ['05/04/2010', '12:00', 'AM', '05/04/2010',
       '12:00', 'PM'],
14   '12 hours 30 minutes' => ['05/04/2010', '12:00', 'AM',
       '05/04/2010', '12:30', 'PM'],
15   '24 hours' => ['05/04/2010', '12:00', 'AM', '05/05/2010',
       '12:00', 'AM'],
16   '1 day 1 minute' => ['05/04/2010', '12:00', 'AM',
       '05/05/2010', '12:01', 'AM'],
17   '1 day 30 minutes' => ['05/04/2010', '12:00', 'AM',
       '05/05/2010', '12:30', 'AM'],
18   '1 day 1 hour' => ['05/04/2010', '12:00', 'AM',
       '05/05/2010', '01:00', 'AM'],
19   '3 days' => ['05/04/2010', '12:00', 'AM', '05/07/2010',
       '12:00', 'AM'],
20   '1 week' => ['05/04/2010', '12:00', 'AM', '05/11/2010',
       '12:00', 'AM']
21 }
22 ...
```

Тони прогоняет все тесты для краткосрочной парковки и убеждается, что они проходят. После этого он прогоняет уже полный набор тестов, включая и написанные ранее для парковки с доставкой в указанное место. Делает он это, потому что не уверен, как будут два частичных

набора взаимодействовать друг с другом. Удостоверившись, что оба набора проходят, Тони спокойно сохраняет их в системе управления версиями и поздравляет себя с тем, что сэкономил компании кучу денег, поскольку повторно воспользовался уже написанным кодом автоматизации, а не стал заново изобретать велосипед.

Экономичная парковка

Далее Тони принимается за примеры для экономичной парковки, приведенные в табл. 3.2. Он создает файл `Economy.feature`, в котором описывает функциональность экономичной парковки (см. листинг 3.4).

Таблица 3.2. Примеры для экономичной парковки

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
4 часа	$8,00
5 часов	$9,00
24 часа	$9,00
1 сутки 1 час	$11,00
1 сутки 3 часа	$15,00
1 сутки 5 часов	$18,00
6 суток	$54,00
6 суток 1 час	$54,00
7 суток	$54,00
1 неделя и 2 дня	$72,00
3 недели	$162,00

Листинг 3.4. Автоматизированные примеры для экономичной парковки

```
1  Feature: функциональность экономичной парковки
2    Калькулятор стоимости парковки вычисляет стоимость стоянки на
     экономичной парковке.
3
4  Scenario Outline: Calculate Economy Parking Cost
5  When I park my car in the Economy Parking Lot for <parking
     duration>
6  Then I will have to pay <parking costs>
7
8  Examples:
```

```
9  | parking duration | parking costs |
10 | 30 minutes       | $ 2.00        |
11 | 1 hour           | $ 2.00        |
12 | 4 hours          | $ 8.00        |
13 | 5 hours          | $ 9.00        |
14 | 6 hours          | $ 9.00        |
15 | 24 hours         | $ 9.00        |
16 | 1 day, 1 hour    | $ 11.00       |
17 | 1 day, 3 hours   | $ 15.00       |
18 | 1 day, 5 hours   | $ 18.00       |
19 | 6 days           | $ 54.00       |
20 | 6 days, 1 hour   | $ 54.00       |
21 | 7 days           | $ 54.00       |
22 | 1 week, 2 days   | $ 72.00       |
23 | 3 weeks          | $ 162.00      |
```

После чего Тони добавляет новые значения в хеш `durationMap` в классе ParkCalcPage. Оставляю эту модификацию амбициозному читателю в качестве упражнения. Перед тем как сохранить тесты для экономичной парковки в системе управления версиями, Тони еще раз прогоняет весь набор.

Те же действия Тони повторяет для обоих вариантов длительной парковки. Оставляю это прилежному читателю. Полный текст примеров можно найти в репозитории github для этой части книги[1].

Резюме

Тони сумел быстро автоматизировать остальные примеры. Поскольку Алекс и Тони применяли модульный подход, автоматизировать оставшиеся четыре типа парковок оказалось несложно даже для Тони – программиста не очень опытного. Получившиеся в результате автоматизированные тесты выражают согласованную с заказчиком спецификацию калькулятора стоимости парковки. В будущем их можно будет адаптировать, если логика расчета изменится.

Тони также обратил внимание, что может повторно использовать большую часть кода автоматизации, написанного при разработке тестов для парковки с доставкой. Вместо того чтобы начинать все заново, он просто включил в справочную таблицу дополнительные продолжительности стоянки, изменил выбранный тип парковки и аннотировал примеры.

1 http://github.com/mgaertne/airport

ГЛАВА 4. Предполагать и сотрудничать

После этого краткого описания итерации вернемся назад и поразмышляем. Была разработана функциональность калькулятора стоимости парковки в аэропорту. Перед тем как приступать к итерации, команда обсудила требования к создаваемому приложению. Для этого был выбран формат рабочей встречи по выработке спецификаций [Adz09, Adz11]. В ходе беседы были выявлены различные типы парковок и подготовлены примеры, отражающие начисление платы за разное время стоянки на разных парковках.

Уяснив примеры, команда приступила к работе над функциональностью. Члены команды параллельно занимались кодированием и тестированием. Тестировщик подготовил первый пример к автоматизации, начал работать с каркасом и продолжал, пока не зашел в тупик. Вероятно, вы помните, что Тони начал с простого примера. Это существенный момент, поскольку при таком подходе вы с самого начала получите правильную реализацию, не увязнув в трясине технических деталей и граничных случаев. Работа над первым примером проясняет некоторые особенности интерфейса конечного пользователя. Отталкиваясь от этой точки, можно развивать примеры во многих направлениях. Какой именно простой случай рассмотреть первым, не так важно; главное применять гибкие методики проектирования и объектно-ориентированный подход. В данном примере в коде автоматизации не было ничего особенно интересного, но в нем имеются кое-какие точки развития. Например, сейчас дата во всех тестах одинакова. В будущем вы, возможно, захотите варьировать ее, например, поручив вычисление продолжительности вспомогательному классу `DurationFactory`, который в частности вычисляет случайную дату.

Важный – быть может, даже самый важный – момент наступил, когда Тони отправился за помощью к программисту. Сотрудничество тестировщика и программиста во имя достижения общей цели

автоматизации играет в методике ATDD особую роль. Хотя задача автоматизации тестов была поручена Тони, Алекс с готовностью оказал ему помощь в работе над первым тестом. Тони узнал от Алекса об общих принципах проектирования и о том, как сделать код удобным для чтения и сопровождения. Со временем Тони ближе познакомился с кодированием автоматизации, и это позволило ему успешно справиться с задачей.

Рассмотрим подробнее каждый из трех встретившихся нам элементов: рабочая встреча по выработке спецификаций, выдвижение пожеланий и сотрудничество.

Рабочие встречи по выработке спецификаций

На рабочей встрече по выработке спецификаций команда обсуждает истории для предстоящих итераций. Поначалу такие рабочие встречи показались мне вариантом методики «водопада» в применении к требованиям. Программисты и тестировщики встречаются со специалистом в предметной области для формулирования требований. Однако гибкие команды могут извлечь из рабочих встреч и другие преимущества.

Вовлечение всех сторон позволяет выработать общий язык для обсуждения проекта. Эрик Эванс (Eric Evans) называет этот язык общеупотребительным (ubiquitous language) [Eva03]. Когда программисты, тестировщики и специалисты в предметной области садятся вместе, чтобы прийти к общему пониманию проекта, они могут устранить многие причины недоразумений, прежде чем те приведут к провалу.

Рабочая встреча как раз и есть инструмент достижения взаимопонимания. Но чтобы такая встреча принесла пользу всем участникам, необходимо помнить о некоторых моментах.

Прежде всего, не следует впустую тратить время представителей бизнеса. Пригласив на встречу эксперта, присутствующие должны ценить его время. В роли представителя бизнеса может выступать владелец продукта, реальный пользователь или специалист в предметной области. Если команда начнет обсуждать на рабочей встрече последние технологии, то в следующий раз представитель бизнеса, скорее всего, не придет. И, значит, ценнейший источник информации о приложении будет потерян.

Заранее отбирайте истории из очереди работ. Если вы знаете, какие истории с наибольшей вероятностью будут реализованы в ближайшем будущем, включите их в список. Если список историй получится слишком большим для обсуждения в отведенное для встречи время, сократите его.

Если порядок обработки истории кажется вам очевидным, можете заранее подготовить данные и представить их на встрече. Представитель бизнеса оценит вашу вовлеченность в проект и изложение простых историй. Подготовив примеры, вы сможете заинтересовать представителя бизнеса и наглядно продемонстрировать ему преимущества рабочих встреч.

Во время рабочей встречи очень важно задавать уточняющие вопросы. Во время подготовки к встрече можно проработать истории внутри команды и составить список открытых вопросов. Со временем вы научитесь задавать уточняющие вопросы более спонтанно, но поначалу лучше заручиться поддержкой команды.

И наконец, я считаю, что на любом совещании или рабочей встрече важнейшим элементом является наглядность. Вместо абстрактного обсуждения покажите, как вы понимаете проблему, и, сверяясь со своими заметками, спросите, согласны ли прочие с вашим видением. Изложить свое понимание можно на магнитно-маркерной доске или на листах бумаги, которые раздаются участникам. Если на встрече присутствует много людей, я бы рекомендовал доску, а когда собираются всего три человека, как в нашем первом примере, листка бумаги вполне достаточно.

Если заказчик находится в другой стране или в другом часовом поясе, то можно попробовать мультимедийные технологии. Интерактивные чаты и средства совместного доступа к экрану позволяют плодотворно сотрудничать, даже не присутствуя на совещании физически. Однако нужно подготовить эти инструменты до начала встречи.

Выдвижение пожеланий

Для успешного внедрения методики разработки через приемочные тесты необходимо уметь выдвигать разумные пожелания. В рассмотренном выше примере Тони реализовал тесты, не имея предварительной информации о деталях устройства калькулятора стоимости парковки.

Для этого Тони сформулировал пожелание, позволившее ему автоматизировать примеры, обсуждавшиеся на рабочей встрече. Тони

обошелся без рассмотрения имеющегося пользовательского интерфейса. Вместо этого он выдвинул пожелание о том, как должен был бы, по его мнению, выглядеть интерфейс. Примеры четко показывали, что для расчета стоимости парковки следует учитывать продолжительность стоянки. Как подтвердил специалист в предметной области, даты прибытия и убытия машины роли не играют. Поэтому Тони не загромождал примеры ненужными деталями.

Вместо того чтобы программировать с использованием реального пользовательского интерфейса, абстрагируйтесь и рассматривайте выраженные в примерах сценарии. Следуя примеру Тони, считайте, что подходящий для тестов интерфейс уже имеется. Дэйл Эмери рекомендует писать тесты так, как будто нужный интерфейс уже существует. Для автоматизации примеров применяйте такой интерфейс, который был бы наиболее удобен для восприятия. При попытке протестировать уже готовое приложение может оказаться, что для его автоматизации придется написать много кода. Если же вы будете «прислушиваться к своим тестам» [FP09], то, возможно, выяснится, что приложению нужен другой интерфейс – по крайней мере, для автоматизации тестов.

Выдвижение пожеланий особенно действенно, если это можно сделать до начала кодирования. В тот момент, когда начинается реализация промышленного кода, может выясниться, каким интерфейсом должно обладать приложение, чтобы его можно было считать тестопригодным. В нашем случае мы видели, что Алекс и Тони работали параллельно. Спроектированный Алексом интерфейс оказался достаточен для рассмотренных примеров, но из-за отсутствия явной продолжительности стоянки пришлось писать дополнительный код автоматизации.

В данном случае преобразование даты и времени прибытия и убытия в продолжительность стоянки производится легко. Возможно, вы обратили внимание, что во всех примерах дата прибытия одна и та же. Но как правило тестировщики и программисты недовольны, когда видят такие зашитые в код значения. Конечно, сгенерировать продолжительность стоянки динамически во время исполнения тестов несложно, но общий объем и сложность вспомогательного кода все же возрастают. Как разработчик ПО, я считаю разумным писать автономные тесты для этого сложного кода, применяя к реализации вспомогательного кода методику разработки через тестирование.

Необходимость преобразования даты и времени прибытия и убытия в продолжительность стоянки – раннее предупреждение о том,

что, возможно, не всё в порядке. Быть может, пользовательский интерфейс спроектирован неправильно. Но на месте пассажира я, наверное, действительно захотел бы ввести дату и время отлета и прилета. Так что с точки зрения потенциальных пользователей интерфейс, вероятно, хороший.

Другая возможная причина – отсутствие должного разделения обязанностей. Сейчас калькулятор сначала вычисляет продолжительность стоянки, а затем – стоимость парковки. Код расчета стоимости можно вынести и тестировать его вообще без привлечения пользовательского интерфейса.

Так или иначе, тесты сами дают рекомендации относительно структуры пользовательского интерфейса. Это касается как автономных, так и приемочных тестов. Если тестировщики и программисты работают независимо друг от друга, то появление проблем с тестопригодностью интерфейса более вероятно, чем при совместной работе.

Сотрудничество

В примере парковочного калькулятора для компании Major International Airport Corp. мы встречались с сотрудничеством на разных уровнях. Тестировщик Тони участвовал в рабочей встрече вместе со специалистом в предметной области Биллом и программистом Филлис. Затем, на этапе автоматизации утвержденных примеров Тони работал совместно с Алексом.

Сотрудничество – еще одно ключевое условие успешного применения методики ATDD. Подумайте, что произошло бы, если бы Тони работал над примерами в одиночестве. Вероятно, он обнаружил бы много ошибок в программе. Об этих дефектах он сообщал бы программистам, и в результате все только раздражались бы. А когда продукт в конце концов был бы выпущен, заказчик оказался бы недоволен неправильной интерпретацией граничных случаев.

Если эта картина вам знакома, то подумайте о том, чтобы начинать проект с рабочей встречи. На ней можно устранить большую часть причин недопонимания между программистами и тестировщиками. А оставшиеся вопросы будут разрешены еще до того, как команда приступит к реализации. Поскольку примеры выражают требования к продукту, команда точно знает, когда реализацию можно считать законченной. Между тестировщиками и программистами существует взаимообмен. Рано или поздно программисты поймут, что только выигрывают, прогоняя автоматизированные примеры перед тем, как

сохранять код в системе управления версиями. И итог закономерен – проект сдан вовремя и без проблем.

Большинству команд, только начинающих практиковать разработку через приемочные тесты, всё это может показаться сказкой. Но существует много примеров успешной поставки программных продуктов, разработанных с применением ATDD в сочетании с другими гибкими методиками, в частности рефакторингом, разработкой через тестирование (TDD), непрерывной интеграцией и концепцией единой команды. Комбинация передового опыта, с одной стороны, и коллективной работы, с другой, оказывается волшебным средством.

Я обращаюсь к идее сотрудничества и в ходе автоматизации тестов. В конце концов, автоматизация тестов – это тоже разработка ПО, поэтому я стремлюсь применять все приемы и методы, которые применяются при разработке самого программного продукта. Как правило, я отношусь к реализации вспомогательного кода для тестов даже более внимательно, чем к продуктовому коду. Это означает, что я применяю разработку через тестирование, рефакторинг и непрерывную интеграцию и к вспомогательному коду тоже.

Тони начал работать с Cucumber еще до написания вспомогательного кода. Но очевидно, что его знаний не хватало, чтобы написать весь код автоматизации тестов самостоятельно. Обнаружив, что зашел в тупик, Тони остановился и обратился к члену команды, который, обладая опытом программирования, мог бы выручить его. Многие команды, не знакомые с ATDD, путают сотрудничество с требованием, чтобы каждый тестировщик умел программировать. Впрочем, тестировщику становится легче жить, когда он может работать над тестами и их автоматизацией независимо от программистов. Поэтому со временем тестировщики осваивают различные трюки, полезные для автоматизации тестов, но обязательным предварительным условием это не является. Это, скорее, результат и побочный эффект, достигаемый в перспективе.

Когда-то я преподавал эту методику тестировщикам в компании, поставляющей медицинское оборудование. Тестировщики раньше работали медсестрами и не имели никакого технического образования. До какого-то момента они тестировали приложение вручную. Программисты стремились увеличить долю автоматизированного тестирования, но у них не было необходимых знаний в предметной области. Сошлись на том, что тестировщики будут описывать суть примеров, а программисты – писать большую часть вспомогательного кода, необходимого для их автоматизации.

Отсутствие опыта программирования не означает, что от методики следует отказаться. Кроме ручки и бумаги, Тони вообще ничего не нужно, чтобы записать пример и использовать его как средство обмена информацией. На самом деле, большинство команд должны начинать именно с этого, не пытаясь автоматизировать примеры вовсе. Расширение обмена информацией само по себе повышает качество процесса разработки. Полной отдачи вы так не получите, но сотрудничество и улучшившийся обмен информацией – уже неплохое начало. Особенно это полезно, когда приходится иметь дело с унаследованным кодом, который (пока) не подготовлен к автоматизированному тестированию.

Резюме

Рабочие встречи по выработке спецификаций, выдвижение пожеланий и сотрудничество – важнейшие инструменты в концертном ансамбле тестирования. Прежде всего, чтобы точно знать, то ли, что нужно, пытается создается команда, необходимо побеседовать с заказчиком. Работая плечом к плечу для определения критериев приемки, команда формирует единое понимание задачи.

Отталкиваясь от целей, сформулированных представителем бизнеса, вы выдвигаете пожелания к интерфейсу приложения. Затем, имея в виду этот тестопригодный интерфейс, вы создаете автоматизированные тесты. В результате приложение по определению оказывается тестопригодным, и вместе с тем есть уверенность, что тесты не слишком тесно связаны с фактической реализацией пользовательского интерфейса.

Наконец, не стоит забывать, что в процессе разработки программного обеспечения мы не одиноки, то есть, зайдя в тупик, можем обратиться за помощью к другим членам команды. Это особенно верно для команд, которые только осваивают гибкие методики и состоят из специалистов разного профиля. Чтобы подняться на более высокий уровень, необходимо учиться у коллег, перенимая их специальные знания. Со временем вы сможете заменять друг друга на время болезни или отпуска.

ЧАСТЬ II.

Система управления светофорами

Рабочие встречи по выработке спецификаций, выдвижение пожеланий и сотрудничество могут сослужить хорошую службу в начале. Но этим методика ATDD не ограничивается. Многочисленные команды экспериментировали с различными подходами к решению задач, с которыми сталкивались, освоив начала ATDD. Подборку этих решений можно найти в книге «Specification by Example» [Adz11].

Вместо того чтобы демонстрировать решение, найденное какой-то одной командой, я хотел бы поговорить об ограничениях, которые некоторые команды налагали в ходе применения ATDD. Слово «driven» в названии «Acceptance Tests **Driven** Development» означает, что в основу разработки кода приложения можно положить примеры. В этой части я, базируясь на уроках, которые извлек в процессе разработки через тестирование, и на собственном опыте, хочу вместе с тобой, любезный читатель, пройти по всем этапам создания системы с применением методик ATDD и TDD и показать, как изменяется ее дизайн.

Мы будем рассматривать систему управления светофорами. В процессе разработки система будет эволюционировать. И вместе с ней будут эволюционировать тесты. Подчас мы будем исследовать фрагменты вспомогательного кода, чтобы лучше понять альтернативные способы проектирования кода продукта. Для автоматизации тестов мы будем использовать написанный на Java API с применением каркаса FitNesse в сочетании со SLiM.

Поскольку я планирую объединиться с тобой в пару, мы отойдем от повествовательного стиля, принятого в части I. Мы будем вместе работать над примерами, но начать должны с рабочей встречи по выработке спецификаций. Я родом из Германии, где правила работы светофоров отличаются от действующих в других странах, поэтому сначала мы должны прийти к общему пониманию предметной области. Приступим.

ГЛАВА 5. Приступая к работе

В этой главе мы ближе познакомимся с некоторыми предпосылками. В разных странах светофоры работают по-разному. Поэтому сначала я познакомлю вас с требованиями.

В этой части мы будем иметь дело с другим каркасом тестирования, FitNesse, поэтому уместно будет краткое введение. Если вы хотите узнать о нем подробнее, обратитесь к приложению B.

В FitNesse связующий и вспомогательный код пишется на другом языке программирования. В этой части мы будем также работать над продуктовым кодом. Но не пугайтесь, мы же пара. Я расскажу об этом языке достаточно, чтобы вы могли начать.

Мы увидим также, как ATDD дополняется TDD. Мы начнем с тестирования бизнес-функций. Затем поговорим о классах с точки зрения разработчика и о том, как тесты влияют на структуру исходного кода продукта. На самом деле, мы будем использовать приемочные тесты для раскрытия предметной области, что позволит нам писать классы основной программы, лучше понимая назначение приложения и результирующую модель.

Светофоры

В большинстве крупных городов, где я бывал, светофоры с точки зрения пешеходов являются скорее рекомендациями, нежели частью системы правил дорожного движения. Изучая законодательство, касающееся систем светофоров, я столкнулся с широким разнообразием хорошо документированных требований и законов. Поскольку законы в разных странах различаются, я буду говорить о тех, к которым привык, – немецких. Вы живете не в Германии? Не страшно, ниже приведены основные положения о светофорах, действующие в Германии.

Для пешеходов предусмотрен светофор с красным и зеленым цветом и изображением фигурки человека, который либо идет, либо сто-

ит. В столице Берлине встречаются также светофорные человечки, которых мы называем Ampelmdnnchen. По форме они отличаются от светофорных человечков в других частях страны. Да, согласен, я отвлекся.

Для водителей транспортных средств предназначены светофоры с тремя цветами: красным, желтым и зеленым. Смысл красного и зеленого света такой же, как для пешеходов. Если горит только желтый свет, значит, светофор скоро переключится с зеленого на красный. Это предупредительный сигнал для водителей. Вообще-то, проезд на желтый свет запрещен, но немногие водители соблюдают это правило. Если красный и желтый горят одновременно, то светофор скоро переключится на зеленый. Движение в этот момент запрещено, но водитель должен быть готов тронуться, потому что эта фаза длится всего лишь долю секунды.

В законе определено одно непререкаемое правило: светофоры на перекрестке не могут гореть зеленым светом одновременно в обоих направлениях. Если такое происходит, светофорная система должна быть немедленно выключена. В этом случае система переключается в режим мигающего желтого света, уведомляя водителей о необходимости следовать указаниям дорожных знаков, когда они принимают решение, кто должен проехать первым. Наша простенькая система управления светофорами тоже должна обрабатывать такие ошибки. Лично я не хочу попасть в аварию только потому, что светофор показывал зеленый свет в обоих направлениях.

Существуют различные модификации светофоров. Например, иногда, чтобы зажегся зеленый свет, пешеход должен нажать кнопку. Тогда контроллер сначала включит красный свет для транспортных средств, а потом зеленый для пешеходов – иногда в сочетании с зеленым для машин в попутном направлении, если светофор установлен на перекрестке. Но по большей части светофоры для пешеходов синхронизированы со светофорами для машин и не имеют средств взаимодействия с пешеходами.

На некоторых пешеходных переходах светофор показывает зеленого шагающего человечка и одновременно издает звуковой сигнал для людей с ослабленным зрением. Незадолго до включения красного света подается звуковой сигнал, означающий, что пешеходы должны поторопиться. На последних пяти секундах зеленой фазы непрерывный сигнал становится прерывистым.

Еще одно дополнение к базовому сценарию – светофоры со специальными сигналами, например, на перекрестке двух больших

улиц. Машины, поворачивающие налево, должны дождаться зеленой стрелки, которая загорается с другой периодичностью, чтобы не задерживать основной поток надолго. Это повышает эффективность управления потоками транспорта на оживленных перекрестках. Но вместе с тем может приводить к другим проблемам, особенно если светофор оборудован также стрелкой для машин, поворачивающих направо (рис. 5.1).

Рис. 5.1. Светофор, остро нуждающийся в рефакторинге

Еще одно дополнение – петли индуктивности, подающие светофору сигнал включить зеленый свет, когда подъезжает машина и горит красный. Петля распознает приближение машины, и система управления светофором планирует для нее зеленую фазу. Другое применение петель индуктивности – измерять скорость движения и притормаживать лихачей, включая красный свет, когда машина едет слишком быстро. Если всего этого недостаточно, упомяну еще о специальных радиосигналах, которые посылают светофорной системе машины скорой помощи и полиции, а также автобусы, чтобы включить для себя зеленый свет.

На первый взгляд, система управления светофором представляется простой, но при ближайшем рассмотрении оказывается весьма

сложно устроенной. Если наша система применяется для управления светофорами в большом городе, то, вероятно, мэр или начальник отдела общественных сооружений захочет получать статистику работы. Располагая такой статистикой, можно оптимизировать транспортные потоки в дневное и ночное время. Разрабатывая первый вариант системы, мы будем иметь в виду такие точки возможного развития. Но, конечно, перед тем как включать дополнительные функции, необходимо реализовать базовые требования.

Созданием систем управления транспортом сейчас занимаются многие крупные компании. Они собирают разнообразные данные для оптимизации транспортных потоков в городах. Задача, которую пытаются решить эти компании, чрезвычайно динамична и очень сложна. А система управления светофорами – лишь одна часть системы в целом. Итак, займемся построением первого элемента для решения этой задачи.

FitNesse

В этой части книги мы будем пользоваться системой FitNesse для составления спецификации первых шагов системы управления светофорами. FitNesse – это вики-система, в которой тесты определяются иерархически с помощью вики-страниц. Поскольку FitNesse представляет собой вики-систему, то в нее можно заносить и дополнительную информацию. По мере роста числа тестов будет расти и объем сопроводительной документации.

Так как FitNesse – вики-сервер, то тесты записываются в нотации вики. В руководстве пользователя FitNesse[1] приведена подробная информация о синтаксисе языка разметки. Мы рассмотрим основные конструкции, которые понадобятся при чтении этой главы, так что не расстраивайтесь, если вы не знакомы ни с FitNesse, ни с вики-системами вообще.

FitNesse позволяет исполнять вики-страницы в ходе тестирования системы. Первоначально FitNesse представляла собой расширение каркаса Framework for Integrated Tests (FIT), но, начиная с 2008 года, она поддерживает собственную систему тестирования, которая называется SLiM – Simple List Invocation Method, –и, в отличие от FIT, не требует, чтобы код, применяемый для автоматизации тестов, удовлетворял условиям лицензии GPLv2. Да, отсюда следует, что мы будем определять тесты светофоров в нотации SLiM, а не в стиле FIT.

1 http://fitnesse.org/.FitNesse.UserGuide

В этом примере у нас будет две таблицы: таблица решений и таблица сценария. Таблица решений служит для того, чтобы описать данные, подаваемые на вход тестируемой системы, выполнить над ними некоторую операцию и сравнить полученные результаты с ожидаемыми. Входные и выходные значения задаются в строках таблицы. Нотация похожа на применяемую в Cucumber, поэтому сложностей с ее пониманием возникнуть не должно. Вот, например, как записываются в виде таблицы решений примеры для парковки с доставкой в указанное место из первой части книги.

Таблица 5.1. Тесты для парковки с доставкой в указанное место, записанные в виде таблицы решений SLiM

```
 1 !/Parking costs for/Valet Parking  /
 2 /parking duration  /parking costs? /
 3 /30 minutes        /$ 12.00        /
 4 /3 hours           /$ 12.00        /
 5 /5 hours           /$ 12.00        /
 6 /5 hours 1 minute  /$ 18.00        /
 7 /12 hours          /$ 18.00        /
 8 /24 hours          /$ 18.00        /
 9 /1 day 1 minute    /$ 36.00        /
10 /3 days            /$ 54.00        /
11 /1 week            /$ 126.00       /
```

В таблице сценария инкапсулирован типичный технологический процесс. Можно считать, что это метод или функция, принимающая параметры. Мы будем описывать в виде сценариев различные встречающиеся технологические процессы. Таблицы сценариев позволяют абстрагировать низкоуровневые детали и сделать примеры более удобными для восприятия.

Вспомогательный код

В этой части книги мы будем писать как вспомогательный код, так и продуктовый код. Отличие от примера с парковками только в том, что код будет разрабатываться через тестирование. Поскольку сейчас мы будем программировать на Java, то в качестве каркаса автономного тестирования возьмем JUnit. Не забывайте, что автоматизация тестов – это тоже разработка ПО, поэтому к вспомогательному коду применимы все обычные рекомендации. Если мы четко выразим свои намерения относительно структуры вспомогательного кода, то сопровождение автоматизированных примеров не составит труда

для того, кому придется это делать впоследствии, – а этим «кем-то» вполне можете оказаться вы сами.

В JUnit, чтобы обозначить, что метод класса является автономным тестом, применяются аннотации (см. строку 4 в листинге 5.2). В JUnit также определены утверждения, которые проверяют состояние или поведение системы и завершают автономный тест ошибкой, если требуемые условия не выполнены. Чаще всего употребляется утверждение `assertEquals(expected, actual)`(см. строку 6 в листинге 5.2), которое проверяет, что значения `expected` и `actual` равны, а если это не так, то завершает тест и печатает сообщение о несовпадении.

Листинг 5.2. Пример автономного теста на Java

```
1  ...
2  public class LightStateTest {
3
4    @Test
5    public void testStateChange() {
6      assertEquals(LightState.RED_YELLOW, LightState.RED.next());
7    }
8  }
```

Я буду также использовать более сложное средство JUnit – параметризованные тесты. Они позволяют отделить технологический процесс от тестовых данных. Процесс аннотируется в тестовом методе, как обычно. Данные же для тестов предоставляет открытый статический (`public static`) метод с надлежащей аннотацией. Дойдя до этого места, мы во всём разберемся подробнее.

Сразу отмечу, что более глубокое изучение JUnit выходит за рамки этой книги. Лично мне понравились книги на эту тему «Test-driven Development by Example» [Bec02], «xUnit Test Patterns» [Mes07] и «Growing Object-oriented Software Guided by Tests» [FP09].

Итак, у нас есть всё, что нужно, чтобы приняться за работу. Вперед!

Резюме

Мы рассмотрели действующие в Германии правила работы светофоров для пешеходов и водителей. Для пешеходов предусмотрено только два цвета: зеленый и красный; для водителей – зеленый, желтый, красный, красный и желтый одновременно (предупреждение, что скоро загорится зеленый). В случае ошибки в системе контроллер должен переключать светофор в режим мигающего желтого света.

Я привел очень краткий обзор каркаса автоматизации и нашего подхода к написанию вспомогательного кода. Вы уже устанавливали интегрированную среду разработки Java IDE самостоятельно, изучали ее? Быть может, даже знакомились с FitNesse? Но даже если нет, все равно приступим к реализации. При необходимости я помогу.

ГЛАВА 6.
Состояния светофора

Мы хотим разработать полную систему управления светофорами для водителей транспортных средств и на перекрестках. Нам придется также решить проблему двух пересекающихся направлений движения.

Первая спецификация для перекрестка описывает правильную последовательность смены состояний светофора. Поэтому первым тестом в FitNesse будет таблица решений, описывающая допустимые переходы состояний светофора для водителей. Я помогу освоить основы. Если вас интересует углубленное изложение, обратитесь к приложению В. Помимо общей информации о FitNesse, я включил также ссылки на онлайновые ресурсы.

Спецификация смены состояний

Начнем с короткой рабочей встречи по выработке спецификаций. В Германии переход состояний светофора выглядит так: после красного света загорается красный одновременно с желтым, затем зеленый, затем желтый и в конце опять красный. Вот и оформим это в виде таблицы (табл. 6.1).

Таблица 6.1. Допустимые состояния светофора для транспортных средств

Предыдущее состояние	Следующее состояние
красный	красный, желтый
красный, желтый	зеленый
зеленый	желтый
желтый	красный

Как, я что-то забыл? А, ну конечно, вы правы. Мигающий желтый означает некорректное состояние системы – в такое состояние систе-

ма переходит при любой неисправности. Это предостерегающий сигнал для водителей. Наша система должна поддерживать это состояние, из которого вывести ее может только специалист-ремонтник. Добавим это состояние в таблицу (табл. 6.2).

Таблица 6.2. Допустимые состояния светофора для транспортных средств

Предыдущее состояние	Следующее состояние
Красный	красный, желтый
красный, желтый	зеленый
Зеленый	желтый
Желтый	красный
мигающий желтый	мигающий желтый

Первый тест

Теперь подготовимся к написанию первого теста. Я скачал последнюю версию FitNesse[1] и сохранил ее в текущем каталоге. Чтобы запустить FitNesse в первый раз, я ввел в командной строке команду `java -jar fitnesse.jar -p 8080`. В текущей версии FitNesse это приведет к извлечению из архива страниц начального вики-сайта. Напечатав сколько-то точек, программа закончит распаковку (см. пример в листинге 6.1) и запустит вики-сервер. Теперь можете открыть свой любимый браузер и перейти по адресу http://localhost:8080 – появится начальная страница FitNesse (рис. 6.1).

Листинг 6.1. Вывод на консоль при первом запуске FitNesse

```
1  FitNesse (v20110104) Started...
2          port: 8080
3          root page: fitnesse.wiki.FileSystemPage at
       ./FitNesseRoot
4          logger: none
5          authenticator: fitnesse.authentication.
       PromiscuousAuthenticator
6          html page factory: fitnesse.html.HtmlPageFactory
7          page version expiration set to 14 days.
```

Последуем инструкциям – нажмем кнопку Edit слева и добавим в существующую таблицу строку, описывающую набор тестов для нашей системы управления светофорами (листинг 6.2). Набор тес-

1 http://fitnesse.org

тов – это собрание тестовых сценариев. Мы хотим, чтобы наши тесты хранились отдельно от остальных страниц вики-сайта FitNesse, поэтому создаем новый пустой набор.

Рис. 6.1. Начальная страница FitNesse, появляющаяся при первом запуске

Листинг 6.2. Добавление ссылки на начальную страницу

```
| '''To Learn More...'''|
| [[A One-Minute Description][FitNesse.UserGuide.
  OneMinuteDescription]]|''What is [[FitNesse][FitNesse.
  FitNesse]]? Start here.''|
| [[A Two-Minute Example][FitNesse.UserGuide.TwoMinuteExample
  ]]|''A brief example. Read this one next.''|
| [[User Guide][FitNesse.UserGuide]]|''Answer the rest of your
  questions here.''|
| [[Acceptance Tests][FitNesse.SuiteAcceptanceTests]]|''
  FitNesse's suite of Acceptance Tests''|
| [[Traffic Light System][TrafficLights]]|''Traffic Light
  System Acceptance Tests''|
```

После сохранения наша система Traffic Light System появится в таблице, но с вопросительным знаком. Это означает, что страница еще не существует. Щелкните по вопросительному знаку, чтобы перейти в окно редактирования, где можно будет добавить новую страницу TrafficLights. Ее содержимое приведено в листинге 6.3.

Листинг 6.3. Корень набора тестов системы управления светофорами

```
1 !1 Traffic Lights control system
2
3 Этот набор состоит из тестов системы управления светофорами.
4
5 ----
6 !contents -R2 -g -p -f -h
7
8 !*< SLiM relevant stuff
9
10 !define TEST_SYSTEM {slim}
11 !path classes
12
13 *!
```

В строках 1–3 приведена краткая документация набора тестов. После строки с четырьмя минусами находится оглавление, автоматически созданное FitNesse (строка 6). Оставшиеся строки – это скрытая секция, в которой говорится, что мы собираемся использовать для описания тестовых таблиц SLiM и что дополнительные вспомогательные классы можно загрузить из папки classes[2]. Сохранив содержимое, мы должны перейти на страницу свойств, нажав кнопку Properties, и изменить тип страницы на suite (набор). Снова сохранившись, мы можем запустить набор, нажав кнопку Suite, хотя ничего хорошего нас в данный момент не ждет, так как в наборе еще нет ни одного теста.

Наш первый тест будет касаться состояний светофора, а следующие – управления различными состояниями на перекрестке. Зная это, разобьем все тесты на два набора: один – для управления светофором на перекрестке, другой – для тестирования смены состояний. Создайте первую страницу, щелкнув по ссылке «Add child» сверху, выберите в качестве типа страницы Suite, назовите страницу TrafficLightStates и пока оставьте предложенное по умолчанию содержимое без изменения. Скоро мы его изменим, а заодно добавим документацию. При нажатии кнопки Add в оглавление набора TrafficLights добавляется только что созданная страница TrafficLightStates. Щелкните по ссылке, ведущей на эту страницу.

2 Если строго следовать принципам TDD, то можно было бы порассуждать о том, следует ли что-то добавлять, не создав сначала теста, завершающегося ошибкой. Но поскольку я работаю с FitNesse вот уже около трех лет, то уверен в своих действиях и знаю, что последние два определения – путь к классам и тестовая система – позже мне действительно понадобятся. Аргументы в пользу педантичного применения TDD в некоторых случаях имеют смысл. Но я стараюсь быть прагматичным, если уверен в своих предположениях и знаю, что если предчувствие меня обманет, то всегда можно будет восстановиться.

Нажмите кнопку **Edit**, чтобы изменить содержимое страницы – ввести осмысленное описание (листинг 6.4).

Листинг 6.4. Оглавление набора TrafficLightStates

```
1  !1 States of traffic lights
2
3  В этом наборе содержатся тесты различных состояний светофора.
4
5  ----
6  !contents -R2 -g -p -f -h
```

И наконец мы можем добавить в набор TrafficLightStates страницу теста CarTrafficStates. Делается это так же, как при добавлении страницы TrafficLightStates в набор TrafficLights. Единственная разница заключается в том, что на этот раз мы задаем тип страницы Test. Открыв новую страницу CarTrafficState, мы можем задуматься о первом тесте. Взглянем на спецификацию (табл. 6.2).

Мы хотим выразить переходы состояний так, как они были описаны на рабочей встрече. Наша таблица выглядит похожей на таблицу решений в SLiM. В таблице решений тестовые сценарии записываются в отдельных строках. Столбец, заголовок которого не заканчивается вопросительным знаком, обозначает входное значение, а столбец, заголовок которого заканчивается вопросительным знаком, – выходное значение, подлежащее проверке. Для переходов состояний нам нужна таблица решений с одним входным значением – предыдущим светом, и одним выходным значением – следующим светом. Запишем первую строку таблицы, соответствующую переходу от красного к красному одновременно с желтым (листинг 6.5).

Листинг 6.5. Первый тест, выражающий переход от красного к красному одновременно с желтым

```
1  !|Traffic Lights            |
2  |previous state|next state? |
3  |red           |red, yellow |
```

Этот тест описывает переход от красного света (запрет движения) к красному одновременно с желтым (скоро включится зеленый). Сохраним первый тест и выполним его. FitNesse сообщает об отсутствии необходимых классов (рис. 6.2).

FitNesse говорит, что мы должны запустить редактор и реализовать классы.

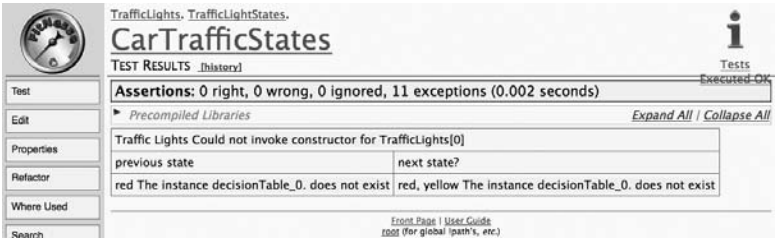


Рис. 6.2. Результат прогона первого теста

Займемся кодированием

Прежде чем я начну объяснять, как пишется вспомогательный код, взглянем на текущую структуру папок, чтобы понять, как настроить IDE. На данный момент проект содержит файл `fitnesse.jar` и папку `FitNesseRoot`, в которой хранятся вики-страницы FitNesse. В корневой набор для нашей системы управления светофорами мы добавили предложение `!path classes`, которое говорит FitNesse, что классы нужно помещать в папку `classes`, включенную в путь поиска классов. Следовательно, откомпилированные версии классов следует помещать в папку `classes`, находящуюся на одном уровне с файлом `fitnesse.jar`. Следуя общепринятому в Java соглашению, исходный код следует хранить в папке `src` на том же уровне, что папка `FitNesseRoot`. Приняв такое соглашение, мы можем настроить IDE для проекта[3].

Первая безуспешная попытка прогнать тест выявила, что мы должны создать класс TrafficLights с конструктором без параметров. Число в квадратных скобках обозначает количество параметров конструктора или функции. Мы создадим этот класс в пакете, подразумеваемом по умолчанию (листинг 6.6).

Листинг 6.6. Первый вспомогательный класс

```
1  public class TrafficLights {
2
3  }
```

Так как в Java конструктор, не имеющий параметров, создается автоматически (если не написан явно), то больше ничего делать не надо.

3 Описание Java IDE выходит за рамки данной книги. Этой теме посвящено немало прекрасных ресурсов. Для проекта, размещенного в github (http://github.com/mgaertne/trafficlights), я пользовался системой Eclipse, но настроить другие IDE тоже не составит труда.

Поскольку мы настроили IDE, так чтобы после любого изменения код перекомпилировался автоматически и откомпилированные версии попадали в папку `classes`, то можно прямо сейчас перезапустить тест в FitNesse и проверить, верны ли наши предположения. Заодно FitNesse скажет, какие еще функции следует реализовать. Результат (рис. 6.3) показывает, что теперь FitNesse нашла класс и ожидает еще два метода: `setPreviousState` с одним параметром и `nextState` без параметров.

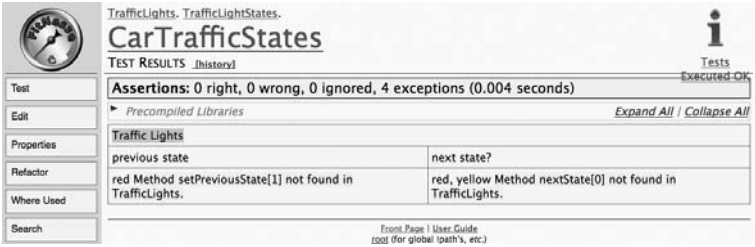


Рис. 6.3. Результат, получившийся после создания класса

В левом столбце, который называется «previous state» (предыдущее состояние) должно находиться значение, которое подается системе для установки начального состояния светофора. Поэтому метод `setPreviousState` принимает один параметр – значение, прочитанное из таблицы. С другой стороны, метод `nextState` должен вернуть значение, которое FitNesse может затем сравнить со значением, хранящимся в таблице. Добавим оба метода (пока пустые) в класс TrafficLights (листинг 6.7).

Листинг 6.7. Вспомогательный класс с пустыми методами

```
1  public class TrafficLights {
2
3    public void setPreviousState(String state) {
4
5    }
6
7    public String nextState() {
8      return null;
9    }
10 }
```

Снова прогоним тест в FitNesse. Наблюдается определенный прогресс, поскольку цвет теста сменился: из желтого – обнаружены исключения или отсутствующий код – он стал красным, то есть имеются ошибки в логике (рис. 6.4).

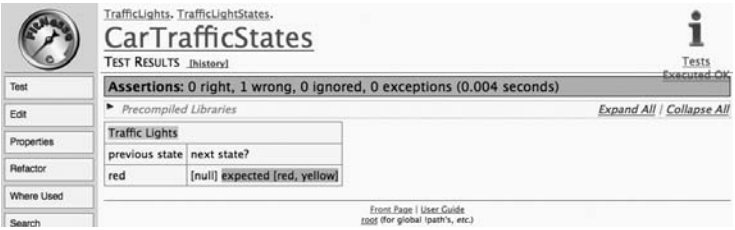


Рис. 6.4. Цвет теста сменился с желтого на красный – прогресс

Вот теперь можно приступить к разработке алгоритма. В первом тесте мы вернем фиксированную строку.

Листинг 6.8. Значение, возвращаемое методом nextState, зашито в код

```
1  public class TrafficLights {
2
3    ...
4    public String nextState() {
5      return "red, yellow";
6    }
7  }
```

Теперь тест проходит (рис. 6.5). Стало быть, пора сохранить результаты в системе управления версиями – всё, включая исходный код и папку `FitNesseRoot`.

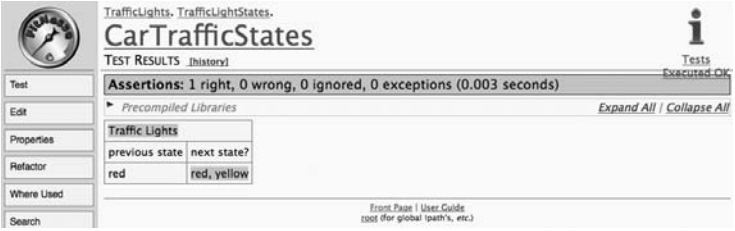


Рис. 6.5. Первый тест прошел

И можно заняться следующим тестом. Добавим вторую строку из таблицы спецификаций – ту, которая описывает переход от красного одновременно с желтым к зеленому (листинг 6.9). После прогона этот тест оказывается красным, так как полученное на выходе значение не совпадает с ожидаемым.

Чтобы этот тест прошел, мы должны запомнить предыдущее состояние и, зная его, решить, каким должно быть следующее. Ну что ж, сохраним состояние в поле типа String и добавим предложение `if` в метод `nextState` (листинг 6.10).

Листинг 6.9. Второй тест, описывающий переход от красного одновременно с желтым к зеленому

```
1 !|Traffic Lights             |
2 |previous state|next state? |
3 |red           |red, yellow |
4 |red, yellow   |green       |
```

Листинг 6.10. Зная предыдущее состояние, решаем, каким должно быть следующее

```
1  public class TrafficLights {
2
3    private String state;
4
5    public void setPreviousState(String state) {
6      this.state = state;
7    }
8
9    public String nextState() {
10     if ("red".equals(state)) return
11         "red, yellow";
12     return "green";
13   }
14 }
```

Снова прогоняем тесты и убеждаемся, что они проходят. И, значит, пора записать их в репозиторий исходного кода.

Раз второй тест прошел, можно точно так же добавить остальные переходы и попутно модифицировать код. Начнем с добавления в таблицу трех оставшихся тестов (листинг 6.11).

Листинг 6.11. Окончательная таблица тестов переходов состояний светофора для водителей

```
1 !|Traffic Lights             |
2 |previous state|next state? |
3 |red           |red, yellow |
4 |red, yellow   |green       |
5 |green         |yellow      |
6 |yellow        |red         |
7 |invalid state |yellow blink|
```

Обратите внимание, что в таблице присутствует и некорректное состояние, в котором светофор мигает желтым.

Чтобы этот тест прошел, мы должны добавить в метод `nextState` еще несколько предложений `if` (листинг 6.12).

Листинг 6.12. Первый тест после реализации всех состояний светофора

```
1  public class TrafficLights {
2  ...
3    public String nextState() {
4      if ("red".equals(state)) return "red, yellow";
5      if ("red, yellow".equals(state)) return "green";
6      if ("green".equals(state)) return "yellow";
7      if ("yellow".equals(state)) return "red";
8      return "yellow blink";
9    }
10 ...
```

Повторно прогоним тесты и убедимся, что они проходят. Всё нормально? Тогда сохраним изменения в репозитории.

Рефакторинг

В этот момент можно было бы перейти к следующей функции и дальнейшим примерам. Однако не стоит торопиться. Есть два момента, которыми я недоволен, и надо бы привести код в порядок. Не исправив дефекты, нет смысла двигаться дальше.

Во-первых, мы реализовали всё в классе, который должен связывать продуктовый код с тестами. Чтобы считать вопрос закрытым, нужно бы вынести этот код в новый класс, или концепцию в коде продукта. В идеале это следует делать, применяя методику разработки через тестирование (TDD) [Bec02]. Сначала мы пишем коротенький тест, прогоняем его и убеждаемся, что он не проходит. Затем реализуем ровно столько кода, чтобы тест прошел, а затем подвергаем код рефакторингу. Не напоминает ли это цикл разработки примеров для парковки? Конечно, только мы будем иметь дело с меньшим фрагментом кода, чем в спецификации.

Новый код мы поместим в параллельный класс. Он называется параллельным компонентом и будет реализовываться параллельно существующему коду. Затем мы заменим вспомогательный код, в котором значения были зашиты, новым кодом, являющимся частью продукта. Если приемочные тесты пройдут, значит, мы все сделали правильно.

Второй момент, который мне не нравится, – то, что весь наш код помещен в пакет по умолчанию. Это плохо – лучше отвести отдельное место для кода, объем которого будет расти. Поскольку этот шаг проще, с него и начнем.

Пакеты

Современные среды разработки располагают инструментами, которые позволяют без труда перенести класс в другой пакет. Это настолько распространенный вид рефакторинга, что он включен в любую IDE, и практически каждый программист либо интуитивно пользуется им с помощью перетаскивания, либо знает, в каком месте IDE находится этот инструмент. Однако нам все-таки нужно сообщить FitNesse, как искать классы в новом пакете. В этом нам поможет таблица импорта. В ней перечислены пакеты, в которых находятся вспомогательные классы. Можно добавить в таблицу строки, показывающие, что в тесте будут использоваться классы из нескольких пакетов. Поместим класс TrafficLights в пакет `org.trafficlights.test.acceptance` и добавим таблицу импорта в начало страницы CarTrafficState, как показано в листинге 6.13.

Листинг 6.13. Окончательный текст страницы примеров после рефакторинга

```
1  !|import                              |
2  |org.trafficlights.test.acceptance|
3
4  !|Traffic Lights             |
5  |previous state|next state? |
6  |red           |red, yellow |
7  |red, yellow   |green       |
8  |green         |yellow      |
9  |yellow        |red         |
10 |invalid state |yellow blink|
```

После рефакторинга и добавления таблицы импорта снова прогоняем тест. Он проходит, поэтому фиксируем изменения в репозитории.

Перечисление LightState

Следующая порция изменений относится только к коду. Мы выделим из вспомогательного кода отдельный класс и соответственно изменим реализацию. Для этого внесем некоторые изменения в дизайн кода. Обработка состояний светофора наводит на мысль о перечислении, поэтому воспользуемся имеющимися в Java классами перечислений. Разумеется, продуктовый код будем разрабатывать в соответствии с принципами TDD.

На первом шаге мы должны написать в JUnit автономный тест, который не проходит. Для этого возьмем данные из первого приемоч-

ного теста – того, где происходит переход состояния из красного в красный одновременно с желтым. Стало быть, первый автономный тест будет выражать именно этот переход. Назовем перечисление LightState, поместим его в пакет `org.trafficlights.domain` и добавим в него функцию `next()`, которая возвращает следующее состояние светофора. В литературе по паттернам проектирования этот паттерн называется «Состоянием» (State) [GHJV94]. Наш первый тест для проверки этой реализации приведен в листинге 6.14.

Листинг 6.14. Первый автономный тест для управления разработкой концепции LightState из предметной области

```
1  package org.trafficlights.domain;
2
3  import static org.junit.Assert.*;
4
5  import org.junit.Test;
6
7  public class LightStateTest {
8
9    @Test
10   public void testStateChange() {
11     assertEquals(LightState.RED_YELLOW, LightState.RED.next());
12   }
13 }
```

Утверждение в строке 11 говорит, что следующим после RED состоянием должно быть RED_YELLOW. Функция `assertEquals` принимает два аргумента: ожидаемое значение (RED_YELLOW) и фактическое значение (RED.next()).

Поскольку класс LightState еще не существует, создадим его в том же пакете и включим в него значения RED и RED_YELLOW, а также абстрактный метод `next()` (листинг 6.15).

Листинг 6.15. Первая версия перечисления, описывающего состояния светофора

```
1  public enum LightState {
2    RED {
3      public LightState next() { return null; }
4    },
5    RED_YELLOW {
6      public LightState next() { return null; }
7    };
8
9    public abstract LightState next();
10
11 }
```

Теперь автономный тест компилируется, и мы можем прогнать его. В результате получим красную полосу, показывающую, что результат неверен. Анализ кода показывает, что функция `next()` для состояния RED возвращает `null` вместо RED_YELLOW. Исправим это, как показано в листинге 6.16.

Листинг 6.16. Реализация перехода из RED в RED_YELLOW в классе LightState

```
1  public enum LightState {
2    RED {
3      public LightState next() { return RED_YELLOW; }
4    },
5    RED_YELLOW {
6      public LightState next() { return null; }
7    };
8
9    public abstract LightState next();
10
11 }
```

Теперь тест становится зеленым. Прогон в FitNesse показывает, что можно без опаски сохранить код в репозитории. Сделаем это.

Реализация остальных переходов состояний, описанных в приемочном тесте, не вызывает сложностей. Переход из состояния RED_YELLOW в GREEN полностью аналогичен переходу из RED в RED_YELLOW. Осознав, что и все прочие тесты будут такими же, я считаю, что будет лучше написать параметризованный тест (листинг 6.17).

Листинг 6.17. Первый и второй автономный тест, записанные в виде управляемого данными теста с помощью имеющегося в JUnit исполнителя Parameterized

```
1  package org.trafficlights.domain;
2
3  import static java.util.Arrays.*;
4  import static org.junit.Assert.*;
5  import static org.trafficlights.domain.LightState.*;
6
7  import java.util.List;
8
9  import org.junit.Test;
10 import org.junit.runner.RunWith;
11 import org.junit.runners.Parameterized;
12 import org.junit.runners.Parameterized.Parameters;
13
14 @RunWith(Parameterized.class)
15 public class LightStateTest {
16
```

```
17   @Parameters
18   public static List<Object[]> data() {
19     return asList(new Object[][] {
20       { RED, RED_YELLOW },
21       { RED_YELLOW, GREEN }
22     });
23   }
24
25   private LightState previousState;
26   private LightState nextState;
27
28   public LightStateTest(LightState previousState,
         LightState nextState) {
29     this.previousState = previousState;
30     this.nextState = nextState;
31   }
32
33  @Test
34   public void testStateChange() {
35     assertEquals(nextState, previousState.next());
36   }
37 }
```

В случае параметризованного теста класс аннотируется исполнителем Parameterized (строка 14). Существует также аннотированный статический метод, который возвращает параметры для класса (строка 18). Эти параметры передаются конструктору (строка 28), который сохраняет их в закрытых полях. Тестовый метод теперь может сравнивать эти поля, а не локальные переменные.

Чтобы этот тест компилировался, мы должны добавить в перечисление значение GREEN, в котором функция `next()` возвращает `null`. Однако в таком виде тест не проходит, а чтобы он проходил, мы должны вернуть GREEN из метода `next()` элемента перечисления RED_YELLOW (строка 8 в листинге 6.18).

Листинг 6.18. Добавлен второй переход

```
1  package org.trafficlights.domain;
2
3  public enum LightState {
4    RED {
5      public LightState next() { return RED_YELLOW; }
6    },
7    RED_YELLOW {
8      public LightState next() { return GREEN; }
9    };
10   GREEN {
11     public LightState next() { return null; }
12   };
```

```
13
14   public abstract LightState next();
15
16 }
```

Зеленая полоса после прогона показывает, что изменения можно сохранить в системе управления версиями.

Точно так же можно реализовать переход из состояния GREEN в YELLOW и из YELLOW в RED. В результате мы получим тесты, показанные в листинге 6.19, и код продукта в листинге 6.20. Не забывайте сохранять код в репозитории после каждого успешного прогона.

Листинг 6.19. Реализованы тесты всех четырех состояний светофора

```
1  package org.trafficlights.domain;
2
3 ...
4
5  @RunWith(Parameterized.class)
6  public class LightStateTest {
7
8    @Parameters
9    public static List<Object[]> data() {
10     return asList(new Object[][] {
11       { RED, RED_YELLOW },
12       { RED_YELLOW, GREEN },
13       { GREEN, YELLOW },
14       { YELLOW, RED }
15     });
16   }
17 ...
18 }
```

Листинг 6.20. Реализован код смены всех четырех состояний светофора

```
1  package org.trafficlights.domain;
2
3  public enum LightState {
4    RED {
5      public LightState next() { return RED_YELLOW; }
6    },
7    RED_YELLOW {
8      public LightState next() { return GREEN; }
9    };
10   GREEN {
11     public LightState next() { return YELLOW; }
12   };
13   YELLOW {
14     public LightState next() { return RED; }
```

```
15   };
16
17   public abstract LightState next();
18
19 }
```

И теперь, чтобы закончить со вспомогательным кодом для FitNesse, мы должны добавить в перечисление еще один элемент, соответствующий состоянию «неизвестно». По умолчанию метод `next()` должен возвращать значение UNKNOWN. Тем самым мы выполним требование немецкого законодательства о том, что в случае ошибки светофор должен мигать желтым светом. Для управления реализацией этого изменения мы снова напишем автономный тест (листинг 6.21).

Листинг 6.21. Реализованы тесты всех четырех состояний светофора и выполнено требование о мигающем желтом

```
1  package org.trafficlights.domain;
2
3 ...
4
5  @RunWith(Parameterized.class)
6  public class LightStateTest {
7
8    @Parameters
9    public static List<Object[]> data() {
10     return asList(new Object[][] {
11       { RED, RED_YELLOW },
12       { RED_YELLOW, GREEN },
13       { GREEN, YELLOW },
14       { YELLOW, RED }
15       { UNKNOWN, UNKNOWN }
16     });
17   }
18
19 ...
```

Чтобы этот тест компилировался, мы добавим значение UNKNOWN в класс LightState и сделаем метод `next()` конкретным, так чтобы он возвращал UNKOWN для любого элемента перечисления, в котором метод `next()` не реализован явно (листинг 6.22).

Листинг 6.22. Окончательная реализация, в которой учтены все состояния светофора

```
1  package org.trafficlights.domain;
2
3  public enum LightState {
4    RED {
```

```
5     public LightState next() { return RED_YELLOW; }
6    },
7    RED_YELLOW {
8     public LightState next() { return GREEN; }
9    };
10   GREEN {
11    public LightState next() { return YELLOW; }
12   };
13   YELLOW {
14    public LightState next() { return RED; }
15   },
16   UNKNOWN;
17
18   public LightState next() {
19    return UNKNOWN;
20   }
21
22 }
```

После прогона этих тестов появляется зеленая полоса, и, значит, мы можем сохранить изменения.

Редактирование состояний светофора

С помощью методики TDD мы разработали компонент, параллельный вспомогательному коду, который необходим для тестов FitNesse. Теперь требуется изменить вспомогательный код, так чтобы можно было воспользоваться новым перечислением. К сожалению, FitNesse понимает только строки, которые нам предстоит преобразовать в элементы перечисления LightState. Можно было бы включить во вспомогательный код длинное предложение if-then-elseif-else, но мне такой дизайн не нравится. Лучше избегать усложнения кода, особенно вспомогательного, который должен быть максимально простым. Вместо этого FitNesse предлагает средство для преобразования строк в объекты предметной области и обратно – редакторы свойств.

Класс редактора свойств должен называться так же, как соответствующий ему класс предметной области, но с добавлением суффикса Editor. Стало быть, для класса LightState нам необходимо реализовать класс LightStateEditor, который должен наследовать классу PropertyEditorSupport и реализовывать методы `getAsText()`и `setAsText(String)`[4]. Разработку класса `LightStateEditor` будем

4 См. руководство пользователя FitNesse по адресу http://fitnesse.org/FitNesse.UserGuide.SliM. CustomTypes.

как обычно вести через тестирование. Первый автономный тест проверяет, что вызов метода `setAsText` с параметром «red» сохраняет в редакторе значение LightState.RED (листинг 6.23).

Листинг 6.23. Автономный тест, проверяющий присваивание значению строки red

```
1  public class LightStateEditorTest {
2
3    private LightStateEditor editor = new LightStateEditor();
4
5    @Test
6    public void setRed() {
7      editor.setAsText("red");
8     assertEquals(LightState.RED, editor.getValue());
9    }
10 }
```

Чтобы этот тест прошел, мы создадим класс LightStateEditor, производный от PropertyEditorSupport, и реализуем в нем метод `setAsText()`, в котором всегда будем вызывать `setValue` с параметром RED (листинг 6.24).

Листинг 6.24. Первоначальная реализация класса LightStateEditor

```
1  package org.trafficlights.domain;
2
3  import java.beans.PropertyEditorSupport;
4
5  public class LightStateEditor extends PropertyEditorSupport {
6
7    public void setAsText(String state) {
8      setValue(LightState.RED);
9    }
10 }
```

Сохранив этот код в репозитории, сделаем автономный тест управляемым данными, снова воспользовавшись исполнителем Parameterized. После добавления преобразования из строки «red, yellow» в соответственный элемент перечисления наш автономный тест принимает вид, показанный в листинге 6.25.

Листинг 6.25. Второй автономный тест, на этот раз управляемый данными

```
1  @RunWith(Parameterized.class)
2  public class LightStateEditorTest {
3
4    @Parameters
5    public static List<Object[]> data() {
```

```
6     return asList(new Object[][] {
7       { "red", RED },
8       { "red, yellow", RED_YELLOW }
9     });
10   }
11
12   private LightStateEditor editor = new LightStateEditor();
13   private String stateName;
14   private LightState state;
15
16   public LightStateEditorTest(String stateName, LightState
state) {
17     this.stateName = stateName;
18     this.state = state;
19   }
20
21   @Test
22   public void setAsText() {
23     editor.setAsText(stateName);
24     assertEquals(state, editor.getValue());
25   }
26 }
```

Чтобы сделать новый тест зеленым, мы добавим условное предложение в код редактора (листинг 6.26).

Листинг 6.26. Еще одно условное предложение для состояния «красный одновременно с желтым»

```
1  public void setAsText(String state) {
2    if ("red".equals(state)) {
3      setValue(LightState.RED);
4      return;
5    }
6    setValue(LightState.RED_YELLOW);
7  }
```

Сохранив текущий код в репозитории, я задумался, в том ли направлении мы движемся. Что будет со следующим тестом? Обработать состояние GREEN так же просто, как в предыдущих примерах.

Но после того как тест для преобразования зеленого света прошел, я понял, что добавил в код еще одно предложение `if`. И этот код нравится мне все меньше и меньше. Поэтому подойдем к реализации редактора по-другому и подвергнем уже написанный код рефакторингу.

Сначала сделаем так, чтобы новый автономный тест проходил, для чего добавим вышеупомянутое предложение `if` (листинг 6.27).

Листинг 6.27. Наличие дублирования – верный признак необходимости рефакторинга

```
1  public void setAsText(String state) {
2    if ("red".equals(state)) {
3      setValue(LightState.RED);
4      return;
5    }
6    if ("red, yellow".equals(state)) {
7      setValue(LightState.RED_YELLOW);
8      return;
9    }
10   setValue(LightState.GREEN);
11 }
```

Увидев зеленую полосу и сохранив код в репозитории, давайте изменим код перечисления LightState, так чтобы конструктор принимал параметр типа String, содержащий текстовое описание состояния (листинг 6.28).

Листинг 6.28. Теперь конструктор LightState принимает описание в параметре типа String

```
1  public enum LightState {
2  ...
3    String description;
4
5    private LightState() {
6      this("");
7    }
8
9    private LightState(String description) {
10     this.description = description;
11   }
12 ...
13 }
```

Чтобы этот код компилировался, мы должны еще добавить явный конструктор без параметров. Необходимо также сделать поле `description` видимым редактору в том же пакете. Теперь можно устранить дублирование, заменив отдельные предложения `if` циклом по элементам перечисления LightState, в котором значение, переданное в параметре, сравнивается с описанием и в случае совпадения вызывается метод `setValue` (листинг 6.29).

Листинг 6.29. Сравнение всех элементов LightState с описанием в цикле

```
1  public class LightStateEditor extends PropertyEditorSupport {
2
```

```
3    public void setAsText(String state) {
4      if ("red".equals(state)) {
5        setValue(LightState.RED);
6        return;
7      }
8      if ("red, yellow".equals(state)) {
9        setValue(LightState.RED_YELLOW);
10       return;
11     }
12     setValue(LightState.GREEN);
13
14     for (LightState lightState: LightState.values()) {
15       if (lightState.description.equals(state)) {
16          setValue(lightState);
17          return;
18       }
19     }
20   }
21 }
```

Теперь можно заменить объявление состояния RED обращением к конструктору, передав ему описание в качестве аргумента (листинг 6.30).

Листинг 6.30. В состоянии RED используется поле описания

```
1  public enum LightState {
2    RED(''red'') {
3    ...
4  }
```

Убедившись, что все тесты проходят[5], мы можем убрать первые две строки из метода `setAsText` в классе редактора. Повторив эту процедуру для состояний RED_YELLOW и GREEN, мы сведем метод `setAsText` к одному циклу (листинг 6.31).

Листинг 6.31. Редактор свойств в минимальном варианте

```
1  public class LightStateEditor extends PropertyEditorSupport {
2
3    public void setAsText(String state) {
4      for (LightState lightState: LightState.values()) {
5        if (lightState.description.equals(state)) {
6          setValue(lightState);
7          return;
8        }
9      }
10   }
11 }
```

5 Надеюсь, вы уже поняли, что мы можем прогонять сразу автономные и приемочные тесты.

Сохраним этот фрагмент в репозитории, после чего добавим тесты для состояний YELLOW и UNKNOWN. И напоследок добавим автономный тест для установки состояния по умолчанию (мигающий желтый), когда в цикле не найдено подходящее значение LightState. И на стадии окончательной подчистки можно убрать из класса LightState конструктор без параметров, потому что он больше не нужен. Получившийся в результате код автономных тестов, редактора и перечисления LightState приведен в листингах 6.32, 6.33 и 6.34 соответственно.

Листинг 6.32. Автономные тесты редактора после завершения реализации метода setAsText

```
1  @RunWith(Parameterized.class)
2  public class LightStateEditorTest {
3
4    @Parameters
5    public static List<Object[]> data() {
6      return asList(new Object[][] {
7        { "red", RED },
8        { "red, yellow", RED_YELLOW },
9        { "green", GREEN },
10       { "yellow", YELLOW },
11       { "yellow blink"", UNKNOWN },
12       { "invalid state", UNKNOWN }
13     });
14   }
15
16   private LightStateEditor editor = new LightStateEditor();
17   private String stateName;
18   private LightState state;
19
20   public LightStateEditorTest(String stateName, LightState state) {
21     this.stateName = stateName;
22     this.state = state;
23   }
24
25   @Test
26   public void setAsText() {
27     editor.setAsText(stateName);
28     assertEquals(state, editor.getValue());
29   }
30 }
```

Листинг 6.33. Код LightStateEditor, разработанный через автономные тесты

```
1  public class LightStateEditor extends PropertyEditorSupport {
2
3    public void setAsText(String state) {
```

```
4     for (LightState lightState: LightState.values()) {
5       if (lightState.description.equals(state)) {
6          setValue(lightState);
7          return;
8       }
9     }
10    setValue(LightState.UNKNOWN);
11  }
12 }
```

Листинг 6.34. Код перечисления LightState после реализации первого метода редактора

```
1  public enum LightState {
2    RED("red") {
3      public LightState next() { return RED_YELLOW; }
4    },
5    RED_YELLOW("red, yellow") {
6      public LightState next() { return GREEN; }
7    },
8    GREEN("green") {
9      public LightState next() { return YELLOW; }
10   },
11   YELLOW("yellow") {
12     public LightState next() { return RED; }
13   },
14   UNKNOWN("yellow blink");
15
16   String description;
17
18   private LightState(String description) {
19     this.description = description;
20   }
21
22   public LightState next() {
23     return UNKNOWN;
24   }
25
26 }
```

Теперь, имея автономный тест, мы можем приступить к разработке второго метода для FitNesse – `getAsText()`. Мы повторно воспользуемся управляемым данными автономным тестом, поскольку будем работать по существу с теми же данными. Добавим тестовый метод `getAsText()` в класс автономного теста (листинг 6.35).

Листинг 6.35. Автономный тест для метода `getAsText` в классе LightStateEditor

```
1  @RunWith(Parameterized.class)
2  public class LightStateEditorTest {
```

```
3
4  ...
5    @Test
6    public void getAsText() {
7      editor.setValue(state);
8      assertEquals(stateName, editor.getAsText());
9    }
10 }
```

При первом запуске тесты не проходят, потому что реализация `getAsText()` по умолчанию не возвращает ожидаемых результатов. Реализуем метод `getAsText()` в классе LightStateEditor, так чтобы он получал сохраненное значение, приводил его к типу LightState и возвращал описание состояния (листинг 6.36).

Листинг 6.36. Код метода `getAsText` в классе LightStateEditor

```
1  public class LightStateEditor extends PropertyEditorSupport {
2  ...
3    public String getAsText() {
4      LightState state = (LightState) getValue();
5      return state.description;
6    }
7  }
```

При прогоне этих тестов произошло нечто для меня удивительное. В последнем автономном тесте проверяется, что неизвестная строка описания приводит к установке значения UNKNOWN. Поскольку отказ теста оказался неожиданным, я задумался, насколько правилен выбранный дизайн. Мы поместили знание о следующем состоянии непосредственно в класс состояния. Разрабатывая редактор для включения нового понятия предметной области в FitNesse, мы решили, что ответственность за выбор состояния UNKNOWN по умолчанию возлагается на редактор. А правильное ли это место?

Заново оценив требования, я предлагаю перенести ответственность за выбор значения UNKNOWN, когда больше ничего не подходит, в класс перечисления, потому что впоследствии это может понадобиться для демонстрации выполнения требований закона в предметных классах. Таким образом, мы должны вынести из метода `setAsText` и оформить в виде отдельной функции код, который получает строковый параметр и возвращает соответствующий ему элемент LightState. Мне кажется уместным назвать эту функцию `valueFor`. Сделаем ее открытым статическим методом класса LightState (листинги 6.37 и 6.38).

Листинг 6.37. Метод `valueFor`, перенесенный из LightStateEditor в LightState

```
1  public enum LightState {
2  ...
3    public static LightState valueFor(String stateName) {
4      for (LightState state: values()) {
5        if (state.description.equals(stateName)) return state;
6      }
7      return UNKNOWN;
8    }
9
10 ...
11 }
```

Листинг 6.38. Измененная реализация класса редактора

```
1  public class LightStateEditor extends PropertyEditorSupport {
2  ...
3    public void setAsText(String state) {
4      setValue(LightState.valueFor(state));
5    }
6  ...
7  }
```

После этих изменений тесты все равно не проходят, но теперь мне ясно, что эти тесты проверяют не поведение редактора, а само перечисление LightState. Вы согласны? Чтобы сообщить о своем намерении человеку, который будет сопровождать код, – это вполне можете быть вы сами – необходимо переименовать тестовые классы. Класс LightStateTransitionTest будет теперь называться LightStateTest, а из LightStateEditorTest удаляется проверка значения по умолчанию (листинг 6.39). Наконец, для нового метода `valueFor()` мы добавляем новый управляемый данными тест LightStateTest, который проверяет правильность поведения для значения по умолчанию (листинг 6.40).

Листинг 6.39. Из автономного теста редактора исключена проверка значения по умолчанию

```
1  @RunWith(Parameterized.class)
2  public class LightStateEditorTest {
3
4    @Parameters
5    public static List<Object[]> data() {
6      return asList(new Object[][] {
7        { "red", RED },
8        { "red, yellow", RED_YELLOW },
9        { "green", GREEN },
10       { "yellow", YELLOW },
```

```
11      { "yellow blink", UNKNOWN }
12    });
13  }
14 ...
15 }
```

Листинг 6.40. Новый класс автономного теста для метода `valueFor`

```
1  package org.trafficlights.domain;
2
3  import static java.util.Arrays.*;
4  import static org.junit.Assert.*;
5  import static org.trafficlights.domain.LightState.*;
6
7  import java.util.List;
8
9  import org.junit.Test;
10 import org.junit.runner.RunWith;
11 import org.junit.runners.Parameterized;
12 import org.junit.runners.Parameterized.Parameters;
13
14 @RunWith(Parameterized.class)
15 public class LightStateTest {
16
17   @Parameters
18   public static List<Object[]> data() {
19     return asList(new Object[][] {
20       { "red", RED },
21       { "red, yellow", RED_YELLOW },
22       { "green", GREEN },
23       { "yellow", YELLOW },
24       { "yellow blink", UNKNOWN },
25       { "invalid value", UNKNOWN }
26     });
27   }
28
29   private String stateName;
30   private LightState state;
31
32   public LightStateTest(String stateName, LightState state) {
33     this.stateName = stateName;
34     this.state = state;
35   }
36
37   @Test
38   public void valueFor() {
39     assertEquals(state, LightState.valueFor(stateName));
40   }
41
42 }
```

Убедившись, что теперь все автономные тесты проходят, мы можем сохранить изменения в системе управления версиями. Мы уже почти закончили. Последний шаг – воспользоваться новым перечислением во вспомогательном классе. Откройте класс TrafficLights и измените сигнатуру метода `setPreviousState()`, так чтобы он принимал параметр типа LightState. Соответственно измените тип поля `state` и тип возвращаемого значения для метода `nextState`, в котором должна остаться только строка `return state.next()` (листинг 6.41).

Листинг 6.41. Окончательная реализация класса TrafficLights, в которой используется понятие предметной области LightState

```
1  package org.trafficlights.test.acceptance;
2
3  import org.trafficlights.domain.LightState;
4
5  public class TrafficLights {
6
7    private LightState state;
8
9    public void setPreviousState(LightState state) {
10     this.state = state;
11   }
12
13   public LightState nextState() {
14     return state.next();
15   }
16 }
```

Вернитесь в браузер и выполните тест FitNesse. Так как он проходит, мы можем записать изменения в репозиторий и поздравить себя с победой. Видите, насколько проще стал вспомогательный код? Нам очень помогла концепция предметной области, которую мы раньше упустили из виду. Самое время выпить чашечку кофе.

Резюме

А пока вы отдыхаете, посмотрим, чего мы достигли. Мы ввели состояние светофора и карту переходов между различными состояниями. Из приемочных тестов мы извлекли концепцию предметной области и добавили для нее ряд автономных тестов. Наконец, мы добавили редактор для новой концепции предметной области и подключили его к имеющемуся приемочному тесту. А за последние несколько минут нас посетило несколько озарений.

Обратив внимание на проблему, на которую так настойчиво указывали тесты, мы осознали пробел в нашем понимании предметной области. После того как ответственность за установку состояния LightState по умолчанию была переложена на класс LightState, вспомогательный код стал гораздо проще. Одно из достоинств разработки приложения через внешние приемочные тесты заключается в обнаружении концепций, отсутствующих в предметном коде. Если бы мы писали тесты для уже существующего кода, то в ходе их автоматизации должны были бы переделать все приложение, чтобы сделать его более тестопригодным. А если структура приложения определяется приемочными тестами, то недостатки дизайна обнаруживаются на более ранней стадии, когда изменения внести гораздо проще. Так как еще нет никаких классов, зависящих от не оптимально спроектированных предметных классов, то у нас остается большая свобода для модификации. Очевидно, что мы только что сэкономили программистам много часов, избавив их от необходимости придумывать собственную логику установки значения по умолчанию. Хорошая работа, говорите? Не могу не согласиться.

ГЛАВА 7.
Первый перекресток

В этой главе мы займемся первой функцией системы управления светофорами. Предположим, что имеются две пересекающиеся дороги; основная задача нашей системы – управлять светофорами в обоих направлениях. Закон требует, чтобы ни при каких обстоятельствах два светофора не горели зеленым светом одновременно. Мы будем иметь это в виду, разрабатывая примеры для контроллера.

Спецификации контроллера

Обратимся к спецификациям контроллера. У нас есть два направления. Контроллер управляет сменой состояний, вероятно, включая следующее состояние по событию таймера. За один раз контроллер изменяет состояние только одного светофора. Например, если в одном направлении горит зеленый, а в другом – красный, то контроллер сменит зеленый на желтый, перед тем как зажечь красный одновременно с желтым вместо красного. В противном случае имело бы место недопустимое состояние.

Таким образом, можно упростить контроллер, ограничившись только случаем, когда одномоментно может быть изменено состояние только одного светофора. Мы должны рассмотреть предыдущие состояния для обоих направлений и вычислить оба следующих состояния, но можем предположить, что контроллер изменяет только первое из двух состояний.

Сначала рассмотрим допустимые переходы – сценарий, которые я называю «в добрый путь». Большинство спецификаций будут отталкиваться именно от этого. Если оба светофора горят красным и мы изменим состояние первого, то получится красный одновременно с желтым на первом светофоре и красный на втором. После этого кон-

троллер переключит красный одновременно с желтым на зеленый. Второй светофор так и останется красным. От зеленого мы перейдем к желтому и затем обратно к красному. Это можно оформить в виде табл. 7.1.

Теперь перейдем к недопустимым состояниям. Перебирая остальные случаи, я прихожу к выводу, что любая другая ситуация ведет к недопустимому состоянию, и, значит, контроллер должен включить мигающий желтый. Поскольку состояние первого светофора может измениться только в момент, когда второй горит красным, любая другая комбинация недопустима и приведет к ДТП, если, конечно, мы не примем контрмер. Например, если во втором направлении горит желтый, то мы не вправе включать никакой свет, кроме красного. Если включить зеленый, то машины, проезжающие на желтый, могут оказаться на перекрестке. Поэтому мы можем добавить к нашей таблице еще одну строку (табл. 7.2).

После этих предварительных размышлений приступим к реализации контроллера.

Таблица 7.1. Сценарий смены состояний «в добрый путь»

Первый светофор	Второй светофор	Следующее состояние первого светофора	Следующее состояние второго светофора
red	red	red, yellow	red
red, yellow	red	green	red
green	red	yellow	red
yellow	red	red	red

Таблица 7.2. Все смены состояний, который нам предстоит рассмотреть

Первый светофор	Второй светофор	Следующее состояние первого светофора	Следующее состояние второго светофора
red	red	red, yellow	red
red, yellow	red	green	red
green	red	yellow	red
yellow	red	red	red
в любом	другом случае	yellow blink	yellow blink

Управление разработкой кода контроллера

К настоящему времени мы описали состояния системы в терминах цветов светофора. Как и раньше, будем тестировать в FitNesse и начнем с первого приемочного критерия.

Создайте в FitNesse новый набор тестов и назовите его CrossingControl. Добавьте в новый набор тестовую страницу TwoCarCrossings. Сначала рассмотрим тесты для сценария «в добрый путь», на основе которых будем управлять поведением контроллера. Итак, если в одном направлении горит зеленый свет, а во втором – красный, то после перехода состояний зеленый должен смениться желтым, а красный остаться красным. Поместим эти условия в таблицу решений, как показано в листинге 7.1.

Листинг 7.1. Первый тест контроллера

```
1 !|FirstLightSwitchingCrossingController                |
2 |first light|second light|first light?|second light?|
3 |green      |red         |yellow      |red           |
```

При попытке выполнить этот тест FitNesse говорит, что отсутствует класс `FirstLightSwitchingCrossingController`. Помните, в какой пакет мы поместили связующий код? Вот туда же поместим и новые классы.

Мы могли бы добавить таблицу импорта в тест перекрестка, как раньше делали для теста состояния. Но это привело бы к дублированию имени пакета и заставило бы тех, кому предстоит сопровождать систему, приспосабливаться к нашему нынешнему решению. В конечном итоге мы могли бы получить набор тестов, в котором имя пакета со связующим кодом размазано по всему проекту. Чтобы избежать этого кошмара, поступим по-другому.

Для начала создайте в наборе TrafficLight новую страницу SetUp, оставив для нее тип по умолчанию. Поместите таблицу импорта в страницу SetUp, после чего вернитесь к странице теста состояний светофора. Теперь в ней появилась свернутая секция, озаглавленная SetUp. Раскрыв ее, мы обнаружим, что каркас автоматически включил только что созданную нами настроечную страницу. Осталось удалить повторную директиву импорта из теста; для этого нажмите кнопку Edit и удалите лишнее.

Теперь можно выполнить тест контроллера светофоров на перекрестке. FitNesse сообщает, что не хватает класса. Мы можем реали-

зовать связующий код, создав класс `FirstLightSwitchingCrossingController` в пакете `org.trafficlights.test.acceptance`. При прогоне теста с пустым классом FitNesse говорит, что пыталась вызвать два метода установки и два метода чтения – по одной паре для каждого светофора. Добавим эти четыре метода в класс и прогоним тест еще раз. Результат показан в листинге 7.2.

Листинг 7.2. Первоначально пустой связующий класс для тестов контроллера

```
1  package org.trafficlights.test.acceptance;
2
3  import org.trafficlights.domain.LightState;
4
5  public class FirstLightSwitchingCrossingController {
6
7    public void setFirstLight(LightState state) {
8    }
9
10   public void setSecondLight(LightState state) {
11   }
12
13   public LightState firstLight() {
14     return LightState.UNKNOWN;
15   }
16
17   public LightState secondLight() {
18     return LightState.UNKNOWN;
19   }
20 }
```

Если теперь вернуть из метода `firstLight` состояние YELLOW, а из метода `secondLight` – состояние RED, то первый тест должен пройти. Прогнав его в FitNesse и убедившись в правильности предположения, мы можем сохранить изменения в репозитории исходного кода.

Далее, как и раньше, можно начать добавлять в таблицу новые примеры. Поскольку эта таблица покрывает только переключение первого светофора, то продолжим рассматривать случаи из сценария «в добрый путь», когда на втором светофоре горит красный свет, а на первом свет меняется – с зеленого на желтый, потом на красный одновременно с желтым и, наконец, на красный. Получающиеся тестовые примеры приведены в листинге 7.3. Он практически повторяет таблицу из спецификации.

Листинг 7.3. Базовые тесты для контроллера, который переключает только первый светофор

```
!|FirstLightSwitchingCrossingController                |
|first light|second light|first light?|second light?|
|green      |red         |yellow      |red           |
|yellow     |red         |red         |red           |
|red, yellow|red         |green       |red           |
|red        |red         |red, yellow |red           |
```

Анализируя примеры, мы замечаем, что состояние второго светофора никогда не изменяется. Чтобы этот приемочный тест прошел, мы можем сохранить состояние первого светофора в переменной экземпляра связующего класса и возвращать следующее состояние из метода `firstLight` (листинг 7.4).

Листинг 7.4. Этот связующий класс контроллера переключает состояние

```
package org.trafficlights.test.acceptance;

import org.trafficlights.domain.LightState;

public class FirstLightSwitchingCrossingController {

  LightState firstState;

  public void setFirstLight(LightState state) {
    firstState = state;
  }

  public void setSecondLight(LightState state) {
  }

  public LightState firstLight() {
    return firstState.next();
  }

  public LightState secondLight() {
    return LightState.RED;
  }
}
```

При повторном прогоне тест проходит, так что пора сохранять изменения в репозитории.

Но меня беспокоит, что мы так и не использовали состояние второго светофора. Пока в этом не было необходимости. Однако на случай недопустимого состояния нам придется запоминать оба состояния и переключать второй светофор в режим мигающего желтого, если получившаяся в итоге комбинация двух цветов запрещена законом.

Прежде чем приступить к разработке этого случая, немного подправим уже написанный код. Учтя возможность недопустимой конфигурации сейчас, мы упростим последующий рефакторинг кода. Будем продвигаться мелкими шажками, реализуя только изменения, необходимые для проверки комбинации состояний обоих светофоров. Прежде всего я хочу сохранить состояние второго светофора в переменной при вызове метода установки (листинг 7.5).

Листинг 7.5. Состояние второго светофора сохраняется в поле

```
1  public class FirstLightSwitchingCrossingController {
2
3    LightState firstState;
4
5    LightState secondState;
6
7  ...
8
9    public void setSecondLight(LightState state) {
10     secondState = state;
11   }
12 ...
13 }
```

После этого изменения все тесты по-прежнему проходят.

Еще одна вещь, которую нужно сделать, прежде чем переходить к недопустимой конфигурации, – вернуть ранее сохраненное состояние второго светофора из метода чтения (листинг 7.6).

Листинг 7.6. Состояние второго светофора читается из поля

```
1  public class FirstLightSwitchingCrossingController {
2
3  ...
4    public LightState secondLight() {
5      return secondState;
6    }
7
8  }
```

Тесты проходят, время сохраняться.

С точки зрения проектирования программы один момент выглядит странно. Есть два метода чтения и два метода установки. Метод чтения для состояния второго светофора просто возвращает ранее запомненное значение. Именно так и должен вести себя любой метод чтения. Но для первого светофора метод чтения возвращает измененное состояние, то есть его вызов определенно имеет побочный эффект – изменяет состояние первого светофора, правда, пока не физически. Это надо бы поправить.

Метод чтения должен только возвращать ранее вычисленное значение. Этот принцип называется разделение команды и запроса (Command/Query-Separation)[1]. Но тогда остается открытым вопрос, где все-таки изменять состояние. В SLiM после выполнения всех методов установки для таблицы решений вызывается метод `execute()`, если он определен в фикстурном классе. Таким образом, мы должны добавить в класс `FirstLightSwitchingController` метод `execute()`, который и изменит состояние первого светофора (листинг 7.7).

Листинг 7.7. Состояние первого светофора изменяется в методе execute

```
1  public class FirstLightSwitchingCrossingController {
2  ...
3    public LightState firstLight() {
4      return firstState;
5    }
6
7  ...
8
9    public void execute() {
10     firstState = firstState.next();
11   }
12 }
```

Убедившись, что тесты проходят, поделимся своими достижениями с коллегами, прибегнув к услугам репозитория.

Теперь можно добавить первый тест, приводящий к недопустимой комбинации. Возьмем первую допустимую конфигурацию и изменим состояние второго светофора во втором столбце. В результате оба светофора должны перейти в режим мигающего желтого (листинг 7.8).

Листинг 7.8. Первая недопустимая конфигурация

```
1 !|FirstLightSwitchingCrossingController                |
2 |first light|second light|first light?|second light?|
3 |green      |red         |yellow      |red           |
4 |yellow     |red         |red         |red           |
5 |red, yellow|red         |green       |red           |
6 |red        |red         |red, yellow |red           |
7 |green      |red, yellow |yellow blink|yellow blink  |
```

Я ожидаю, что контроллер предотвращает недопустимые конфигурации до и после переключения любого светофора, то есть переводит оба светофора в режим, уведомляющий водителей о неполадках.

Поскольку в методе `execute` логика проверки отсутствует, то на вновь добавленной строке тест спотыкается. Чтобы он прошел, мы

1 http://pragprog.com/articles/tell-dont-ask.

должны проверять совместимость состояний до переключения первого светофора. Комбинация зеленого в одном направлении и красного одновременно с желтым в другом недопустима, так как разрешает водителям движение в первом направлении, хотя со второго тоже вот-вот поедут.

В уже написанный метод `execute` связующего класса добавим строку, в которой проверяется, совместимы ли состояния первого и второго светофора (листинг 7.9).

Листинг 7.9. Контрольная проверка, выполняемая контроллером перед изменением состояния светофора

```
1  public class FirstLightSwitchingCrossingController {
2  ...
3    public void execute() {
4      if (!LightState.RED.equals(secondState)) {
5         firstState = LightState.UNKNOWN;
6         secondState = LightState.UNKNOWN;
7         return;
8      }
9      firstState = firstState.next();
10   }
11 }
```

Если второй светофор показывает не красный свет, то конфигурация недопустима, и мы переводим оба светофора в состояние `UNKNOWN`, показывая, что система неисправна. Предложение `return` предотвращает опасность. Прогоняем тесты, видим, что они проходят, но перед тем как сохранять изменения, немного подчистим метод `execute()`.

Прежде всего, создадим отдельный метод для проверки условия в предложении `if`, чтобы впоследствии, если понадобится проверять дополнительные условия – а я уверен, что понадобится, – его можно было расширить. Назовем этот метод, в котором будет проверяться допустимость конфигурации, `isValidLightStateConfiguration()` (листинг 7.10).

Листинг 7.10. Метод проверки, вынесенный из предложения if

```
1  public class FirstLightSwitchingCrossingController {
2  ...
3    public void execute() {
4      if (!isValidLightStateConfiguration()) {
5         firstState = LightState.UNKNOWN;
6         secondState = LightState.UNKNOWN;
7         return;
8      }
9      firstState = firstState.next();
```

```
10   }
11
12   private boolean isValidLightStateConfiguration() {
13     return LightState.RED.equals(secondState);
14   }
15 }
```

Прогнав все тесты, можно еще вынести код, который переводит оба светофора в предупреждающее состояние. Для этого перенесем обе строки, в которых состояниям первого и второго светофора присваивается значение `UNKNOWN`, в метод `warningConfiguration()` (листинг 7.11).

Листинг 7.11. Установка предупреждающего состояния вынесена в отдельный метод

```
1  public class FirstLightSwitchingCrossingController {
2  ...
3    public void execute() {
4      if (!isValidLightStateConfiguration()) {
5         warningConfiguration();
6         return;
7      }
8      firstState = firstState.next();
9    }
10
11   private boolean isValidLightStateConfiguration() {
12     return LightState.RED.equals(secondState);
13   }
14
15   private void warningConfiguration() {
16     firstState = LightState.UNKNOWN;
17     secondState = LightState.UNKNOWN;
18   }
19 }
```

Прогнав тесты, я могу со спокойной душой зафиксировать изменения.

Далее начнем добавлять новые примеры недопустимого поведения, переставляя оставшиеся состояния. Поскольку таких примеров довольно много, заведем для них отдельную таблицу (листинг 7.12). Добавляйте примеры по одному и каждый раз проверяйте, проходят ли тесты.

Листинг 7.12. Все перестановки недопустимых конфигураций помещены в отдельную таблицу

```
1  !2 Допустимые комбинации
2
```

```
!|FirstLightSwitchingCrossingController                |
|first light|second light|first light?|second light?|
|green      |red         |yellow      |red           |
|yellow     |red         |red         |red           |
|red, yellow|red         |green       |red           |
|red        |red         |red, yellow |red           |

!2 Недопустимые комбинации

!|FirstLightSwitchingCrossingController                 |
|first light  |second light|first light?|second light?|
|green        |red, yellow |yellow blink|yellow blink |
|green        |green       |yellow blink|yellow blink |
|green        |yellow      |yellow blink|yellow blink |
|yellow       |red, yellow |yellow blink|yellow blink |
|yellow       |green       |yellow blink|yellow blink |
|yellow       |yellow      |yellow blink|yellow blink |
|red, yellow  |red, yellow |yellow blink|yellow blink |
|red, yellow  |green       |yellow blink|yellow blink |
|red, yellow  |yellow      |yellow blink|yellow blink |
|red          |red, yellow |yellow blink|yellow blink |
|red          |green       |yellow blink|yellow blink |
|red          |yellow      |yellow blink|yellow blink |
|yellow blink |red         |yellow blink|yellow blink |
|yellow blink |red, yellow |yellow blink|yellow blink |
|yellow blink |green       |yellow blink|yellow blink |
|yellow blink |yellow      |yellow blink|yellow blink |
```

Одна комбинация работает неправильно – когда, первый светофор мигает желтым, а на втором горит красный. Если в одном направлении уже индицируется неисправность, то мы вправе ожидать, что и в другом будет то же самое. Чтобы обеспечить такое поведение, добавим в связующий код специальный случай для состояния `UNKNOWN` (листинг 7.13).

Листинг 7.13. После добавления обработки состояния UNKNOWN

```
public class FirstLightSwitchingCrossingController {
...
  private boolean isValidLightStateConfiguration() {
    return !LightState.UNKNOWN.equals(firstState)
       && LightState.RED.equals(secondState);
  }

...
}
```

Теперь все тесты проходят. Запишем результаты в репозиторий.

Рефакторинг

Как и раньше, мы могли бы счесть работу законченной и двигаться дальше. Но меня начинает тревожить дизайн как самого кода, так и примеров. Посмотрим, что с этим можно сделать.

Рефакторинг примеров. Что касается примеров, то разделение между сценарием «добрый путь» и всеми остальными случаями привело к значительной избыточности в результатах, ожидаемых для недопустимых комбинаций. Во второй таблице ожидаемый результат всегда один и тот же. Следовательно, мы можем вообще избавиться от двух правых столбцов и выразить свое намерение, сказав, что все следующие примеры представляют недопустимые комбинации (листинг 7.14).

Листинг 7.14. Таблица, из которой удалена избыточная информация о недопустимых комбинациях

```
1  ...
2  !2 Недопустимые комбинации
3
4  !|scenario       |invalid combination|firstLight||secondLight|
5  |set first light |@firstLight                                 |
6  |set second light|@secondLight                                |
7  |execute                                                      |
8  |check           |first light        |yellow blink            |
9  |check           |second light       |yellow blink            |
10 ...
```

В данном случае желаемого можно достичь, вынеся общую часть в таблицу сценария. Таблица сценария – это аналог функции в языке программирования. В других каркасах она могла бы обозначаться каким-нибудь ключевым словом.

Читая таблицу сверху вниз, мы видим, что наш сценарий называется `invalid combination` и принимает два параметра: `firstLight` и `secondLight` (строка 4). Затем описываются его шаги. Сначала устанавливается состояние первого светофора, потом – второго (строки 5 и 6), затем вызывается метод `execute` (строка 7) и проверяются результаты, получившиеся после переключения первого светофора (строки 8 и 9). Оба светофора должны мигать желтым светом.

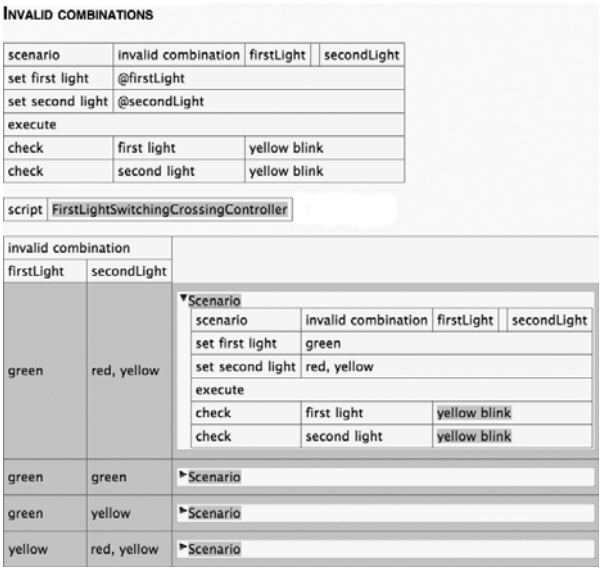
Таблицы сценариев используются в SLiM в сочетании с таблицами скриптов. Так, мы можем объявить, что класс, которые ранее применялся в таблице решений, является актором таблицы скрипта. Актор работает аналогично первой строке таблицы решений. Разница лишь в том, что на странице одного теста можно объявить

несколько акторов таблицы скрипта, тогда как класс для таблиц решений единственный (строка 11 в листинге 7.15). В окончательной таблице (листинг 7.15) перечислены все недопустимые комбинации состояний светофоров. В строке 13 говорится, что определенный выше сценарий `invalid combination` будет вызываться несколько раз. В строке 14 определяется, какой столбец содержит первый, а какой – второй параметр сценария. Начиная со строки 15, идут все возможные комбинации.

Листинг 7.15. Избыточная информация о недопустимых комбинациях удалена

```
1  ...
2  !2 Недопустимые комбинации
3
4  !|scenario       |invalid combination|firstLight||secondLight|
5  |set first light |@firstLight                                 |
6  |set second light|@secondLight                                |
7  |execute                                                      |
8  |check           |first light        |yellow blink           |
9  |check           |second light       |yellow blink           |
10
11 !|script|FirstLightSwitchingCrossingController|
12
13 !|invalid combination    |
14 |firstLight  |secondLight|
15 |green       |red, yellow|
16 |green       |green      |
17 |green       |yellow     |
18 |yellow      |red, yellow|
19 |yellow      |green      |
20 |yellow      |yellow     |
21 |red, yellow |red, yellow|
22 |red, yellow |green      |
23 |red, yellow |yellow     |
24 |red         |red, yellow|
25 |red         |green      |
26 |red         |yellow     |
27 |yellow blink|red        |
28 |yellow blink|red, yellow|
29 |yellow blink|green      |
30 |yellow blink|yellow     |
```

Все тесты проходят. Но результат выглядит несколько иначе, чем раньше. Теперь для каждой строки таблицы недопустимых комбинаций имеется сворачиваемая секция с меткой «scenario». Щелкнув по стрелке, мы можем раскрыть секцию и увидеть результаты отдельных шагов сценария (рис. 7.1).



INVALID COMBINATIONS

scenario	invalid combination	firstLight	secondLight
set first light	@firstLight		
set second light	@secondLight		
execute			
check	first light	yellow blink	
check	second light	yellow blink	

script	FirstLightSwitchingCrossingController

invalid combination	
firstLight	secondLight
green	red, yellow
green	green
green	yellow
yellow	red, yellow

Scenario

scenario	invalid combination	firstLight	secondLight
set first light	green		
set second light	red, yellow		
execute			
check	first light	yellow blink	
check	second light	yellow blink	

Scenario

Scenario

Scenario

Рис. 7.1. Таблица сценария приводит к появлению сворачиваемых секций

Таблицы сценариев автоматически включаются из страницы, для которой, по соглашению, выбрано специальное имя (так же, как для страниц SetUp и TearDown). Чтобы сделать тест более компактным, поместим определение сценария в страницу с именем `ScenarioLibrary`, создав ее в наборе `CrossingControl`. Еще раз прогнав тесты после этой манипуляции и убедившись, что все они проходят, мы можем сохранить изменения в репозитории.

Сейчас в таблице сценария есть одна неудачная конструкция. Мы напрямую ссылаемся на метод `execute`. На нем взгляд спотыкается. Исправим это, оставив в теле метода `execute` только вызов нового открытого метода, на который и будем ссылаться. Имя метода должно раскрывать его назначение. Поскольку источником идеи является таблица сценария, то и изменение внесем сначала в нее.

Вопрос в том, как бы лучше выразить смысл метода `execute` в этом контексте. Поскольку мы занимаемся переключением светофоров, то очень неплохо будет звучать «switch first light» (переключить первый светофор) (листинг 7.16).

Листинг 7.16. Теперь таблица сценария ссылается на операцию с понятной семантикой

```
!|scenario       |invalid combination|firstLight||secondLight|
|set first light |@firstLight                                  |
|set second light|@secondLight                                 |
|switch first light                                            |
|check           |first light        |yellow blink            |
|check           |second light       |yellow blink            |
```

Разумеется, теперь все тесты, в которых используется этот сценарий, не проходят. Поэтому перенесем тело метода `execute()` в новый метод, который сделаем открытым (листинг 7.17).

Листинг 7.17. Тело метода execute перенесено в метод switchFirstLight()

```
public class FirstLightSwitchingCrossingController {
...
 public void execute() {
   switchFirstLight();
 }

 public void switchFirstLight() {
   if (!isValidLightStateConfiguration()) {
      warningConfiguration();
        return;
   }
   firstState = firstState.next();
 }
...
}
```

Прогнав тесты, убеждаемся, что плоды нашего труда готовы к сохранению.

Рефакторинг связующего кода. Пока что мы реализовали решения прямо в связующем классе, а не в классе предметной области. С концепцией контроллера вроде бы всё нормально, но, занимаясь некорректными комбинациями, я заподозрил, что еще одной концепции не хватает. Вспомните о валидаторе – компоненте, который в наших тестах проверял совместимость состояний. Это как раз то, что прекрасно легло бы в модель предметной области. Нам нужно проверять, является ли комбинация состояний светофоров допустимой. Мне кажется, что поручить эту обязанность валидатору было бы естественно и разумно.

Но меня беспокоят еще две вещи. Сейчас наш контроллер управляет состояниями только двух светофоров. В будущем, возможно, количество направлений увеличится – например, мы захотим, чтобы

у машин, поворачивающих налево, был собственный светофор, или чтобы движение на Т-образном перекрестке регулировалось отдельно по каждому из трех направлений. Поскольку можно предположить, что в будущем контроллеру придется иметь дело с дополнительными направлениями, любые решения, касающиеся более чем одного состояния, следует отложить до момента, когда будут определены соответствующие примеры.

И еще мне пришла в голову мысль о том, что именно делает наш контроллер. Метод `execute()`, очевидно, просто обновляет состояние одного светофора. Быть может, в будущем придется переключать одновременно несколько светофоров. Чтобы избежать преждевременной оптимизации, оставим текущий дизайн и функциональность как есть. Впоследствии, когда примеры ясно докажут необходимость такого рода изменений, мы их внесем.

Итак, на данный момент я вижу только одно полезное изменение: валидатор состояний двух светофоров. И ясно понимаю, к чему сведется решение. Мы введем объект-стратегию, который будет проверять предполагаемую комбинацию состояний до вызова метода `next()` одного из светофоров.

Будем подвергать имеющийся код рефакторингу мелкими шажками. В основу нового класса валидатора положим метод `isValidLightStateConfiguration()` (листинг 7.18).

Листинг 7.18. Метод, который станет основой нового класса

```
1  private boolean isValidLightStateConfiguration() {
2    return !LightState.UNKNOWN.equals(firstState)
         && LightState.RED.equals(secondState);
3  }
```

Мы разработаем класс валидатора внутри связующего класса, а затем перенесем его в отдельный пакет для использования в продуктовом коде. Но прежде проанализируем эту функцию. В настоящий момент она зависит от состояний обоих светофоров. Перед тем как делать из нее класс, модифицируем ее так, чтобы состояния передавались в виде параметров (листинг 7.19).

Листинг 7.19. Функция проверки после добавления двух параметров

```
1  public class FirstLightSwitchingCrossingController {
2  ...
3    public void switchFirstLight() {
4      if (!isValidLightStateConfiguration(firstState,secondState)) {
5         warningConfiguration();
6         return;
```

```
7     }
8     firstState = firstState.next();
9    }
10
11   private boolean isValidLightStateConfiguration(
        LightState firstState, LightState secondState) {
12    return !LightState.UNKNOWN.equals(firstState)
           && LightState.RED.equals(secondState);
13   }
14 ...
15 }
```

Прогнав тесты в FitNesse, убеждаемся, что по ходу дела ничего не «сломали».

Теперь можно переработать метод `isValidLightStateConfiguration(LightState, LightState)`, превратив его в метод специального класса. Для этого нужно сначала создать пустой класс валидатора внутри CrossingController (листинг 7.20).

Листинг 7.20. Новый пустой класс валидатора

```
1  public class FirstLightSwitchingCrossingController {
2  ...
3    private static class CrossingValidator {
4
5    }
6  }
```

Добавление пустого класса не должно ничего поломать, но на всякий случай прогоним тесты еще раз. Теперь можно перенести функцию `isValidLightStateConfiguration(LightState, LightState)` в только что созданный пустой класс. Средства рефакторинга, имеющиеся в моей IDE, пока не позволяют перенести метод в класс CrossingValidator. Чтобы получить такую возможность, мы должны ввести новый параметр – объект CrossingValidator (листинг 7.21).

Листинг 7.21. Валидатор временно сделан параметром функции проверки, чтобы можно было перенести в него метод

```
1  public class FirstLightSwitchingCrossingController {
2  ...
3    public void switchFirstLight() {
4      if (!isValidLightStateConfiguration(new
           CrossingValidator(), firstState, secondState)) {
5        warningConfiguration();
6        return;
7      }
8      firstState = firstState.next();
9    }
```

```
10
11   private boolean isValidLightStateConfiguration(
        CrossingValidator validator, LightState firstState,
        LightState secondState) {
12     return !LightState.UNKNOWN.equals(firstState)
           && LightState.RED.equals(secondState);
13   }
14 ...
15 }
```

Поскольку такое изменение могло что-то поломать, на всякий случай прогоним тесты. Теперь IDE предлагает перенести функцию `isValidLightStateConfiguration(LightState, LightState)` в класс CrossingValidator. Результат показан в листинге 7.22.

Листинг 7.22. Класс валидатора после переноса в него функции проверки

```
1  public class FirstLightSwitchingCrossingController {
2  ...
3    public void switchFirstLight() {
4      if (!new CrossingValidator().
           isValidLightStateConfiguration(firstState, secondState)) {
5        warningConfiguration();
6        return;
7      }
8      firstState = firstState.next();
9    }
10
11 ...
12   static class CrossingValidator {
13
14     boolean isValidLightStateConfiguration(
         LightState firstState, LightState secondState) {
15       return !LightState.UNKNOWN.equals(
           firstState) && LightState.RED.equals(secondState);
16     }
17
18   }
19 }
```

Да, вы правы, надо бы еще раз прогнать тесты. Всё зеленое, это хорошо. Пойдем дальше и подчистим код. Конструкция `new CrossingValidator()` внутри условия в предложении `if` мне не нравится. Создадим отдельное поле и инициализируем его экземпляром валидатора (листинг 7.23).

Листинг 7.23. В классе контроллера заведено поля для хранения валидатора

```
1  public class FirstLightSwitchingCrossingController {
2
```

```
3    private CrossingValidator validator = new
CrossingValidator();
4   ...
5    public void switchFirstLight() {
6     if (!validator.isValidLightStateConfiguration(
           firstState, secondState)) {
7       warningConfiguration();
8       return;
9     }
10    firstState = firstState.next();
11   }
12  ...
13  }
```

Прогон тестов показывает, что мы ничего не поломали. Теперь почти всё готово для того, чтобы превратить класс валидатора в отдельный класс верхнего уровня. Но сначала надо бы придумать для метода `isValidLightStateConfiguration()` более короткое имя. Имена методов должны быть тем короче, чем в большей области метод виден [Mar08a]. До сих пор речь шла о защищенном методе вложенного класса, поэтому к длине имени претензий не возникало. Но если мы хотим сделать метод открытым, то следует задуматься о переименовании. На ум приходит имя `isValidConfiguration()`, которое по-прежнему выражает наше намерение. После переименования добавим модификатор `public` как к вложенному классу, так и к самому методу. А заодно уберем модификатор `static` из объявления класса (листинг 7.24).

Листинг 7.24. Переименованный метод валидатора

```
1  public class CrossingValidator {
2
3    public boolean isValidConfiguration(LightState
         firstState, LightState secondState) {
4     return !LightState.UNKNOWN.equals(
         firstState) && LightState.RED.equals(secondState);
5    }
6
7  }
```

Прогон тестов показывает, что всё по-прежнему работает. Теперь можно попросить IDE сделать из вложенного класса класс верхнего уровня и поместить его в пакет предметного кода. Ну и, конечно, прогнать тесты FitNesse. Если проблем не возникнет, значит, мы провели изменения корректно.

Но прежде чем помещать новый код в систему управления версиями, нужно устранить еще одну проблему. До сих пор для всех классов

в предметном пакете были как автономные тесты, так и приемочные. А вот для класса `CrossingValidator` автономных тестов в предметном пакете нет. Это чревато неприятными сюрпризами для разработчика, который внесет какие-то изменения и не захочет или не сможет прогонять приемочные тесты – быть может, потому что набор приемочных тестов чрезмерно разрастется по размеру или по времени выполнения. Поскольку мы отвечаем за работу этого класса, то давайте, пока не забыли, задним числом добавим для него несколько автономных тестов. Как и прежде, наиболее элегантно это можно выразить в JUnit с помощью параметризованных тестов (листинг 7.25).

Листинг 7.25. Добавленные задним числом автономные тесты для класса валидатора

```
1  package org.trafficlights.domain;
2
3  import static java.util.Arrays.*;
4  import static org.junit.Assert.*;
5  import static org.trafficlights.domain.LightState.*;
6
7  import java.util.List;
8
9  import org.junit.Test;
10 import org.junit.runner.RunWith;
11 import org.junit.runners.Parameterized;
12 import org.junit.runners.Parameterized.Parameters;
13
14 @RunWith(Parameterized.class)
15 public class CrossingValidatorTest {
16
17  @Parameters
18   public static List<Object[]> values() {
19     return asList(new Object[][] {
20        {RED, RED, true},
21        {GREEN, RED, true},
22        {YELLOW, RED, true},
23        {RED_YELLOW, RED, true},
24        {UNKNOWN, RED, false},
25        {GREEN, GREEN, false},
26        {YELLOW, GREEN, false},
27        {RED_YELLOW, GREEN, false},
28        {RED, GREEN, true},
29     });
30   }
31
32   private LightState firstState;
33   private LightState secondState;
34   private boolean valid;
```

```
35
36   public CrossingValidatorTest(LightState firstState,
            LightState secondState,
37          boolean valid) {
38     this.firstState = firstState;
39     this.secondState = secondState;
40     this.valid = valid;
41   }
42
43   @Test
44   public void isValidConfiguration() {
45     assertEquals(valid, new CrossingValidator().
         isValidConfiguration(firstState, secondState));
46   }
47 }
```

Добавляя автономные тесты, я обратил внимание на дефект в дизайне. Комбинация состояний RED первого светофора и GREEN второго допустима, но наш валидатор считает, что это не так. Это, безусловно, должно вызвать настороженность, так как в будущем мы можем забыть конкретные детали реализации. Так что очень хорошо, что мы занялись автономным тестированием и увидели расхождение между приемочными тестами (которые демонстрируют, что требования удовлетворены) и автономными (выявившими дефект проектирования).

Проблема связана с тем, что наш контроллер проверяет комбинацию состояний до и после переключения первого светофора в одной функции. Но мы строим валидатор так, что он проверяет условие как до, так и после предполагаемого переключения и возвращает результаты для обоих состояний.

Этот анализ оставляет нам два варианта действий. Либо выделить два шага валидации – до и после переключения – и производить переключение только тогда, когда полученная конфигурация допустима, либо оставить всё как есть, но переименовать валидатор, отразив в названии принцип его работы.

Недостаток второго подхода в том, что мы зашиваем в класс валидатора слишком много знаний о том, как его предполагается использовать. Получится весьма специализированный валидатор, применимый только к одному сценарию – переключение первого светофора. Мне как проектировщику такое решение кажется неудачным.

У первого подхода есть очевидное достоинство – ясность предназначения. И для меня оно перевешивает необходимость двойной проверки. Поэтому я все-таки изменю дизайн контроллера. Для начала вернемся к функции, которая переключает первый светофор, и

добавим в нее еще одну проверку после переключения; это не должно нарушить существующее поведение (листинг 7.26).

Листинг 7.26. Добавлена еще одна проверка после переключения

```
1  public class FirstLightSwitchingCrossingController {
2  ...
3    public void switchFirstLight() {
4      if (!validator.isValidConfiguration(firstState, secondState)) {
5         warningConfiguration();
6         return;
7      }
8      firstState = firstState.next();
9
10     if (!validator.isValidConfiguration(firstState, secondState)) {
11        warningConfiguration();
12     }
13   }
14 ...
15 }
```

Прогон приемочных тестов показывает, что всё нормально. Теперь займемся дефектным валидатором и последним автономным тестом. Тест комбинации красного с зеленым не проходил. Это наводит на мысль о том, что в функции проверки недостает какого-то условия. Добавим его (листинг 7.27)

Листинг 7.27. Исправленная функция проверки

```
1  public class CrossingValidator {
2
3    public boolean isValidConfiguration(LightState
         firstState, LightState secondState) {
4      if (LightState.UNKNOWN.equals(firstState)) return false;
5      if (LightState.RED.equals(secondState)) return true;
6      if (LightState.RED.equals(firstState) &&
           LightState.GREEN.equals(secondState)) return true;
7      return false;
8    }
9
10 }
```

Теперь все тесты проходят, и мы можем добавить новые автономные тесты. Попробуем другие перестановки состояний. На комбинации RED и RED YELLOW мы снова получаем ошибку (листинг 7.28).

Листинг 7.28. На последнем тесте обнаружена очередная ошибка

```
1  @RunWith(Parameterized.class)
2  public class CrossingValidatorTest {
```

```
3
4  @Parameters
5   public static List<Object[]> values() {
6     return asList(new Object[][] {
7        {RED, RED, true},
8        {GREEN, RED, true},
9        {YELLOW, RED, true},
10       {RED_YELLOW, RED, true},
11       {UNKNOWN, RED, false},
12       {GREEN, GREEN, false},
13       {YELLOW, GREEN, false},
14       {RED_YELLOW, GREEN, false},
15       {RED, GREEN, true},
16       {UNKNOWN, GREEN, false},
17       {GREEN, RED_YELLOW, false},
18       {YELLOW, RED_YELLOW, false},
19       {RED, RED_YELLOW, true},
20     });
21   }
22 ...
23 }
```

Теперь нам предстоит изменить добавленное ранее специальное условие. Поскольку мы точно знаем, что в третьей строке метода `isValidConfiguration(LightState, LightState)` состояние второго светофора не может быть RED, то вторую часть условия можно удалить (листинг 7.29).

Листинг 7.29. Модифицированная функция проверки

```
1  public class CrossingValidator {
2
3    public boolean isValidConfiguration(LightState
         firstState, LightState secondState) {
4      if (LightState.UNKNOWN.equals(firstState)) return false;
5      if (LightState.RED.equals(secondState)) return true;
6      if (LightState.RED.equals(firstState)) return true;
7      return false;
8    }
9
10 }
```

Этот код проходит все существующие тесты, так что можно добавлять новые. Следующую ошибку мы получаем на комбинации RED и UNKNOWN. Из анализа функции проверки сразу видно, что для решения проблемы нужно рассматривать еще случай, когда второй светофор мигает желтым (листинг 7.30).

Листинг 7.30. После добавления проверки нахождения второго светофора в режиме мигающего желтого

```
1  public class CrossingValidator {
2
3    public boolean isValidConfiguration(LightState
         firstState, LightState secondState) {
4      if (LightState.UNKNOWN.equals(firstState) ||
           LightState.UNKNOWN.equals(secondState)) return false;
5      if (LightState.RED.equals(secondState)) return true;
6      if (LightState.RED.equals(firstState)) return true;
7      return false;
8    }
9
10 }
```

Убедившись, что теперь тесты проходят, добавим оставшиеся перестановки (листинг 7.31). Больше ошибок не обнаруживается. Разумеется, необходимо прогнать и приемочные тесты. Они тоже проходят, поэтому новую версию можно записывать в репозиторий.

Листинг 7.31. Окончательный вариант автономных тестов валидатора

```
1  @RunWith(Parameterized.class)
2  public class CrossingValidatorTest {
3
4  @Parameters
5  public static List<Object[]> values() {
6    return asList(new Object[][] {
7       {RED, RED, true},
8       {GREEN, RED, true},
9       {YELLOW, RED, true},
10      {RED_YELLOW, RED, true},
11      {UNKNOWN, RED, false},
12      {GREEN, GREEN, false},
13      {YELLOW, GREEN, false},
14      {RED_YELLOW, GREEN, false},
15      {RED, GREEN, true},
16      {UNKNOWN, GREEN, false},
17      {GREEN, RED_YELLOW, false},
18      {YELLOW, RED_YELLOW, false},
19      {RED, RED_YELLOW, true},
20      {RED_YELLOW, RED_YELLOW, false},
21      {UNKNOWN, RED_YELLOW, false},
22      {GREEN, YELLOW, false},
23      {YELLOW, YELLOW, false},
24      {RED, YELLOW, true},
25      {RED_YELLOW, YELLOW, false},
26      {UNKNOWN, YELLOW, false},
27      {GREEN, UNKNOWN, false},
```

```
28      {YELLOW, UNKNOWN, false},
29      {RED, UNKNOWN, false},
30      {RED_YELLOW, UNKNOWN, false},
31      {UNKNOWN, UNKNOWN, false}
32    });
33  }
34 ...
35 }
```

После такого урока мы в следующий раз обязательно будем добавлять автономные тесты на ранних стадиях проектирования.

Резюме

Сделаем перерыв и осмыслим, что мы только что сделали. Мы добавили первый перекресток двух дорог. Мы прошли путь от простой смены состояний к контроллеру, который отвечает за координацию смены состояний двух светофоров, предотвращая загорание зеленого в обоих направлениях.

Прорабатывая пример за примером, мы сначала реализовали всё в связующем коде. Но после того как приемочные тесты прошли, мы преобразились в проектировщика и в дальнейших действиях руководствовались знаниями, полученными во время написания связующего кода. Мы решили преобразовать код в новую концепцию предметной области. Добавляя задним числом автономные тесты, мы обнаружили дефект проектирования и изменили проект, так чтобы он более четко выражал наше намерение. Из этого опыта мы извлекли важный урок – одних лишь приемочных тестов недостаточно для управления проектированием.

Это и есть основная разница между методикой TDD, применяемой для управления проектированием, и методикой ATDD, которая предназначена для управления требованиями. TDD фокусируется на крохотных автономных тестах и за счет этого позволяет двигаться в определенном направлении при проектировании классов. Напротив, ATDD фокусируется на требованиях, то есть специфицировании функциональности. TDD имеет дело с технической реализацией, ATDD – с критериями приемки с точки зрения бизнеса.

Если говорить о технологии, то мы снова видели таблицы решений, но еще узнали о таблицах сценариев. Мы могли бы также преобразовать успешные тесты переключения первого светофора, избавившись от последнего столбца, где во всех случаях утверждается, что второй светофор горит красным. Но, подумав, отказались от этого

шага, так как при этом будущему читателю было бы неясно, каков результат успешной смены состояний. В случае недопустимой смены состояний результат всегда один и тот же. Решение скрыть повторяющееся состояние «мигающий желтый» было оформлено в виде таблицы сценария. Дополнительная причина скрыть «мигающий желтый» заключается в том, что теперь мы можем изменить представление недопустимой конфигурации в одном месте для всех тестов, которые еще могут появиться в будущем, – при условии, что можно воспользоваться написанным сценарием.

ГЛАВА 8.
Раскрывай и исследуй

В этой части мы наблюдаем эволюцию системы управления светофорами, которую разрабатываем снаружи внутрь. Одно из основных отличий этой части от первой заключается в том, что мы используем связующий код как руководство для проектирования классов предметной области. Идти таким путем нам помогало также то, что в роли проектировщика системы выступал тот же человек, точнее пара, который писал связующий код.

На самом деле, по ходу реализации связующего кода для автоматизации тестов мы выявили различные варианты дизайна продуктовых классов. В первом случае мы решили реализовать паттерн «Состояние», уяснив, как связующий код различает состояния светофора. Во втором случае мы смогли выделить контроллер, переключающий светофоры, прямо из кода связующего класса.

К сожалению, мы слишком поздно осознали необходимость автономных тестов для связующего кода. Из-за отсутствия тестов, управляющих написанием кода, мы столкнулись с проблемой – когда мы вынесли код контроллера и начали тестировать его, обнаружилось, что у контроллера слишком много обязанностей. Если бы мы написали тесты заранее, то эта проблема была бы выявлена раньше – на этапе анализа тестов. Таким образом, мы получили урок на будущее – чем раньше добавить автономные тесты для вспомогательного кода, тем лучше. Будь у нас тесты, мы обнаружили бы избыток обязанностей, еще не приступив к выделению кода контроллера.

В основе методики ATDD лежит автоматизация тестирования. Поскольку это не что иное как разработка программ, то код автоматизации по возможности следует разрабатывать с применением TDD. Руководствуясь интересами бизнеса, вы, вероятно, захотите начать функциональное тестирование как можно раньше. Но по мере накопления связующего кода остается все меньше и меньше уверенности в отсутствии побочных эффектов при его изменении. Наличие авто-

номных тестов для связующего кода поможет избежать этой ловушки.

Итак, применение ATDD позволяет как управлять разработкой кода продукта, так и раскрывать и исследовать предметную область. Но не забывайте, что лучше реализовывать автономные тесты для своего кода на ранней стадии, чтобы не оказаться потом в безвыходном положении. Подытоживая тему, рассмотрим эти технические особенности ATDD более пристально.

Раскрытие предметной области

В этой части книги мы использовали спецификации как инструмент раскрытия предметной области разрабатываемой системы. Первоначально у нас было весьма смутное представление о структуре кода продукта. Автоматизируя один пример за другим, мы подвергали свои идее проверке. Разобравшись, как мог бы выглядеть продуктовый код, мы начали писать класс, относящийся к предметной области.

Реализация связующего кода помогла раскрыть особенности предметной области. Поняв, как можно было бы обеспечить правильную работу примеров, мы смогли поразмыслить и выявить паттерны, лежащие в основе кода из предметной области. По мере того как в связующем коде накапливалось поведение, характерное для предметной области, становилось очевидным, как выделить его из связующего кода или поместить в параллельный класс.

Разрабатывая примеры для перехода состояний светофора, мы осознали, что необходим контроллер, который будет управлять несколькими светофорами и координировать их переключение. Выделение контроллера придало разработке новый импульс. Видя, в каком направлении движемся, мы извлекли предметный код из связующего. Выяснилось, что мы забыли про валидатор комбинаций состояний. Это открытие помогло еще улучшить код.

В ходе эволюции концепции состояния, представляющей состояние светофора, мы разрабатывали код параллельно существующему связующему коду. Проектируя различные состояния светофора, мы руководствовались приобретенным ранее опытом. Разработка проекта велась через микротесты различных состояний и переходов. Завершив реализацию паттерна «Состояние», мы смогли заменить часть

связующего кода новым классом перечисления и проверить новый объект предметной области с помощью уже написанных приемочных тестов.

Одна из проблем, характерных для такого подхода к ATDD, заключается в том, что неудачность реализации может быть обнаружена слишком поздно. Если бы, подключив новый предметный объект к связующему коду, мы увидели бы много ошибок в приемочных тестах, то пришлось бы откатиться довольно далеко назад к версии, не содержащей ошибок. Такие крупные шаги таят в себе немалый риск в плане проектирования кода.

Из-за этого я обычно стараюсь откладывать столь серьезный шаг до тех пор, пока не стану яснее представлять себе получающуюся структуру программы. В случае перечисления, описывающего состояния светофора, я бы подождал подольше, чем при разработке кода с помощью TDD. Вообще говоря, я уверен, что могу полагаться на собственный прошлый опыт, но когда дело доходит до приемочных тестов, я стараюсь загнать эту мысль подальше и подойти на шаг ближе к вырисовывающемуся дизайну. Это типичное компромиссное решение. Я понимаю, что если ждать слишком долго, то структура связующего кода, возможно, начнет быстро ухудшаться, но вместе с тем осознаю, что если перейти к рефакторингу чересчур рано, то дело может окончиться еще хуже – только на этот раз для продуктового кода.

Объемный рефакторинг не составляет серьезной проблемы, потому что современные IDE одинаково хорошо поддерживают как мелкий, так и крупный рефакторинг. Поэтому чем дольше я могу позволить себе экспериментировать с различными вариантами предметного кода, тем более взвешенным окажется окончательное решение.

Еще один момент важен при работе в команде, где занято не только два программиста. Ваша цель – предъявить свою работу коллегам как можно раньше. Непрерывная интеграция [DMG07] мелких изменений помогает избежать кошмара объединения. Когда изменений немного, за побочными эффектами можно наблюдать во время объединения. В противном случае можно оказаться в ситуации, когда необходимо объединить плоды месячных трудов, находящиеся в разных ветках. Я слышал, что в одной компании на это потребовалось две недели. Обычно я сохраняю результаты очень часто – несколько раз в день.

Управление разработкой продуктового кода

Начиная с примеров, мы разрабатываем спецификацию программы. Автоматизируя примеры, мы непосредственно проверяем, соответствует ли программа спецификации. Поэтому Гойко Аджич предложил изменить название «Acceptance Test-driven Development» на «Specification by Example» (специфицирование через примеры) [Adz09, Adz11].

В первой части мы видели, что, применяя методику ATDD, программисты и тестировщики могут работать параллельно. В этой части мы разрабатывали тесты в паре. Переход от примера к продуктовому коду направляет проектирование классов предметной области. Вместо того чтобы работать параллельно, мы начали с примеров. Автоматизируя примеры, мы не только раскрыли для себя предметную область, но и поняли, как надо реализовывать код продукта.

Преимущество такого применения ATDD в том, что для каждой строки продуктового кода у нас уже имеются автоматизированные приемочные тесты. Архитектура системы тестопригодна по определению и в значительной мере уже покрыта автоматизированными тестами.

Еще одно достоинство заключается в том, что автоматизированные тесты покрывают фактические требования. Примеры выражают именно то, что важно специалистам в предметной области, а не проверяют малосущественные условия.

Высокая степень покрытия продуктового кода тестами дает ценную обратную связь при переходе к следующей итерации. В 2008 году я имел удовольствие повозиться с кодом FitNesse. Поскольку это открытая система, я мог скачать исходный код и посмотреть, как он устроен. Поначалу я просто экспериментировал, добавив одну-две функции. Позже я исправлял ошибки в системе. В состав дистрибутива входило около 2000 автономных тестов и 200 приемочных. На их полный прогон уходило меньше пяти минут. Таким образом, через пять минут после внесения изменений я уже знал, поломалось что-то или нет. Это помогало мне не только изучать исходный код, но и обнаруживать побочные эффекты внесенных мной изменений.

Хороший набор функциональных тестов помогает вновь пришедшим сотрудникам освоить систему, как мне помогли тесты FitNesse. Благодаря небольшому времени выполнения и быстрой обратной связи коллеги могут сразу приступить к работе с исходным кодом.

Такой подход позволяет без труда добиться коллективного владения кодом и тестами. В подобной системе каждый может без опаски вносить любые изменения, не оправдываясь невозможностью изменять что-либо в коде, потому что он, мол, написан кем-то другим.

Если программа нуждается в развитии или сопровождении, то мы возвращаемся к исходным примерам. Так как примеры, реализованные в автоматизированных тестах, содержат всю историю проекта до настоящего момента, то мы уже имеем готовую документацию. Поскольку тесты можно выполнить и одним нажатием кнопки узнать, отвечает ли продукт спецификации, то тем самым мы имеем исполняемую спецификацию продукта. Это кстати означает, что спецификацию следует поддерживать в актуальном состоянии. Все эти преимущества мы получаем автоматически, если разрабатываем код через примеры.

Тестируйте связующий код

Один из уроков, которые я извлек после катастрофически неудачной попытки выделить валидатор, заключается в том, что связующий код необходимо тестировать. Как только мы выделили поведение и задним числом написали автономные тесты, обнаружилось, что первоначальный дизайн содержал дефект. Мы могли бы предотвратить такое развитие событий, если бы с самого начала разрабатывали связующий код через тесты.

По зрелом размышлении, этот совет можно назвать оторванным от реальности. Я часто ловлю себя на исследовании предметной области разрабатываемого продукта. На этапе исследования новые вопросы о предметной области возникают с такой скоростью, что я откладываю вопрос об автономных тестах. В режиме исследования я пытаюсь как можно больше узнать о самой предметной области. Почувствовав, что понимаю ее достаточно хорошо, я возвращаюсь к тому коду, который был написан до перехода в режим исследования, и начинаю всё сначала, но уже с применением TDD. Я поступаю так сознательно, чтобы раскрепостить сознание. В режиме осмысления мой мозг работает, как во время мозгового штурма. Правое полушарие задействовано в полной мере, в голову приходят замечательные мысли. Прервав этот поток мыслей и идей, я рискую утратить сосредоточенность.

К сожалению, это иногда выходит мне боком. Когда запас новых идей истощается, я остаюсь с кодом, который трудно тестировать.

Оказавшись в такой ситуации, я выбрасываю всё, что натворил, и начинаю с начала. На первый взгляд, мучительно выкидывать код, который так хорошо выглядит в редакторе, но конечным-то результатом является не сам код, а мысли и уроки, усвоенные по ходу его написания. Я уверен, что, начав заново, я в итоге получу более гибкий дизайн и даже уже известные шаги смогу с самого начала поверить автономными тестами.

Вообще говоря, тестирование связующего кода может происходить двумя путями. В первом случае связующий код примитивен, не содержит никаких управляющих конструкций – ветвлений и циклов, и все функции не превышают десяти строчек. Такой код прост для понимания. Он настолько короткий, что читатель с первого взгляда улавливает его назначение. Выделяя функции из связующего кода, я обращаю особое внимание на именование методов – чтобы мои намерения были очевидны. Когда код настолько прост, у меня нет сомнений, что и через год я его пойму. Что касается его тестирования, то я вполне удовлетворен, если приемочные тесты проходят. В данном случае для тестирования связующего кода достаточно самих приемочных тестов.

Иное дело, когда связующий код содержит ветвления и циклы. Даже если и в этом случае методы и функции не длиннее десяти строк, я все равно иногда испытываю тревогу по поводу условной логики. Я знаю, что если я сейчас ничего не предприму, то через год буду ломать себе голову. В таком случае я пишу автономные тесты. Это помогает читателю понять ход моих мыслей и при необходимости внести в код изменения. Одних лишь приемочных тестов теперь недостаточно, потому что для покрытия всех путей исполнения их придется выполнить слишком много.

В общем случае, встречая в связующем коде условное предложение, я задумываюсь над тем, как написать для него автономный тест. Работая в одной компании, я столкнулся необходимостью интерфейса к вызовам служб в системе на основе EJB. Я без труда мог имитировать этот интерфейс для автономных тестов моего связующего кода. Руководствуясь рекомендациями из книги «Growing Object-oriented Software, Guided by Tests» [FP09], я создал текучий API для тестирования и имитации удаленных вызовов тестируемой системы, чтобы можно было быстро прогонять автономные тесты. Поскольку этот API был мне хорошо известен, я сумел написать тесты связующего кода с высоким уровнем покрытия (свыше 90 %). Позже, прогоняя тесты на реальной системе, я смог заменить исполнитель служб таким,

в котором для получения настоящего локального и удаленного адреса EJB-службы использовалось отражение. Класс ServiceExecutor стал шлюзом для обращений к другой системе, который можно было легко подменить на этапе автономного тестирования.

Поскольку все обращения к удаленной службе проходят через единственную точку, я назвал этот паттерн Needle Eye (Игольное ушко). Вызов внешней системы стал таким угольным ушком. Но для этого нужно, чтобы либо на эту подсистему налагались какие-то ограничения, либо было принято какое-то соглашение. В нашем случае имена пакетов EJB2-служб следовали строгому соглашению. Это позволило нам воспользоваться отражением для поиска локального и удаленного адреса оконечной точки службы. Мы инкапсулировали вызовы удаленной системы в простом интерфейсе, который предоставлял метод `execute`, принимающий объект общего класса входа в службу и возвращающий полученный от службы результат (листинг 8.1).

Листинг 8.1. Сигнатура исполнителя EJB2-службы для паттерна «Игольное ушко»

```
1  public Output execute(Input serviceInput);
```

В классах фикстур мы прибегли к внедрению зависимости в конструктор, чтобы заменить конкретный исполнитель службы, в котором применялось отражение для поиска адреса реальной службы, имитацией, удовлетворяющей требованиям автономного теста. Теперь мы могли заодно моделировать исключения, возбуждаемые тестируемой системой, добиваясь тем самым высокого уровня покрытия связующего кода автономными тестами. Это вселяло в нас уверенность в правильности кода тестов, эволюционирующих вместе с проектом.

Цените свой связующий код

Вообще говоря, я отношусь к тестам по меньшей мере с тем же уважением, что к коду продукта. А обычно ставлю связующий код даже выше продуктового, потому что он находится между двумя критически важными для разработки точками.

С одной стороны, продуктовый код может изменяться на каждой итерации. Существующая функциональность может модифицироваться чуть ли не ежедневно. Поэтому код, который напрямую взаимодействует с тестируемым приложением, крайне нестабилен.

В книге «Agile Software Development – Patterns, Principles, and Practices» [Mar02], Роберт Мартин упоминает «Принцип устойчивых зависимостей». Пакеты должны зависеть только от более устойчивых пакетов. Поскольку код, находящийся рядом с неустойчивым кодом приложения, сам по себе неустойчив, то имеет смысл инкапсулировать изменения в этой части кода и сделать их зависимыми от более устойчивых частей. Обычно я достигаю этой цели либо за счет обертки, либо путем реализации паттерна делегирования (см. [GHJV94]).

Второе соображение, касающееся связующего кода, связано с нестабильностью языка тестов. Тестовые данные образуют язык, представляющий предметную область приложения. Они эволюционируют вместе с разработкой приложения. Это означает, что связующий код, который связывает тестовые данные с тестируемым приложением, также должен эволюционировать. Связующий код должен обладать достаточной гибкостью, чтобы изменяться по мере уточнения нашего понимания базовых концепций.

Обычно я помещаю поведение, которое изменяется вместе с развитием языка тестов, в отдельные классы. Это относится, например, к таким концепциям, как денежные суммы (число с плавающей точкой вместе со знаком валюты), интервалы или моменты времени. Иногда задачу можно решить и по-другому: завести в продуктовом коде перечисление и реализовать необходимое преобразование с помощью редактора свойств, как мы видели в примере перечисления LightState. Основное достоинство такого подхода в том, что метод преобразования пишется легко. Иногда я обнаруживаю, что некоторая концепция так или иначе необходима в предметной области. Тогда остается только радоваться, что у меня в связующем коде уже есть механизм преобразования.

Между этими двумя неустойчивыми интерфейсами находится тонкий слой кода, который связывает два представления предметной области. Чем тоньше этот слой, тем обычно лучше спроектированы предметные классы. Обнаружив, что для связывания концепций предметной области и приемочных тестов приходится писать большой объем кода, я задумываюсь о правильности дизайна. И как правило выясняется, что моему предметному объекту недостает какой-то важной концепции. Перенеся эту концепцию из связующего кода в предметный, я поддерживаю связующий код настолько простым, насколько возможно, – но не проще.

Подводя итог, можно сказать, что мы имеем крайне неустойчивые зависимости от разрабатываемой системы и от тестовых данных. Эти два мира должна соединять тонкая ниточка. Если это не так, то попытайтесь охватить взглядом весь объем продуктового, вспомогательного и связующего кода и понять, что можно улучшить. Если такой возможности не видно, продолжайте писать новые тесты, но внимательно приглядывайтесь к коду. Со временем понимание предметной области может измениться, и вам откроются новые способы усовершенствовать какую-то часть кода.

Резюме

Слово «driven» в аббревиатуре ATDD подчеркивает, что весь процесс разработки приложения должен быть направлен снаружи внутрь. В этой части книги мы видели, как приемочные тесты позволяют раскрыть особенности предметной области приложения.

Но одного лишь раскрытия предметной области мало. Мы должны управлять разработкой продуктового кода через приемочные тесты. Отталкиваясь от приемочных тестов, мы получаем возможность понять, как следует проектировать продуктовые классы. В долгосрочном плане это очень помогает.

Мы также узнали, как высоко следует ценить код, написанный для тестирования системы. Одна из главных причин, по которым автоматизация тестов не работает у моих клиентов, – недостаточно почтительное отношение к коду тестов. Автоматизация тестирования – частный случай разработки ПО. И значит, при создании тестов нужно применять рефакторинг и вообще все те же принципы, что применяются к продуктовому коду. Лично я, когда проектирую связующий код, нахожу особенно полезными принципы единственной обязанности, открытости-закрытости, разделения интерфейсов и инверсии зависимости.

Побочным эффектом применения этих принципов является необходимость тестирования связующего кода. Я хочу сказать, что одного лишь прогона приемочных тестов недостаточно, сам связующий код должен разрабатываться через тестирование. По возможности разбивайте связующий код на небольшие функции, которые можно тестировать по отдельности. Но даже если это невозможно, старайтесь найти способы разорвать связи между тестируемой системой и связующим кодом, например такие, как паттерн «Игольное ушко».

В противном случае небольшое изменение системы может повлечь за собой масштабные изменения ее страховочной сетки – приемочных тестов. Не идите по этому пути.

ЧАСТЬ III.

Принципы разработки через приемочные тесты

В этой части мы ближе познакомимся с принципами, на которых основано всё, что было сделано в предыдущих главах. Почему вообще работает методика специфицирования через примеры? На какие проблемы следует обращать внимание? Можно ли выявить какие-нибудь паттерны в этом подходе?

ГЛАВА 9. Использование примеров

В двух первых частях книги мы начинали с примеров, которые диктовались задачами бизнеса. Чтобы составить хорошие примеры, необходимо знать предметную область или, по крайней мере, иметь доступ к человеку, который такими знаниями обладает. В первом примере команда недостаточно хорошо понимала предметную область, что и выразилось в отзывах на первую неудачную попытку. Поэтому компания Major International Airport Corp. организовала совместную рабочую встречу. Чтобы как-то компенсировать свое незнание предметной области, команда пригласила на нее специалиста со стороны бизнеса – Билла. С его помощью Филлис и Тони сумели сформулировать примеры, демонстрирующие бизнес-правила расчета стоимости парковки в аэропорту.

Во втором примере мы прорабатывали задачу из предметной области, в которой я считал себя достаточно компетентным. Если бы мы зашли в тупик, то осознали бы это, прежде чем стало бы слишком поздно. Для той степени детальности, которая нам требовалась в этом примере, наших знаний предметной области было достаточно, поэтому, систематизировав некоторые априорные соображения, можно было приступать к работе. Примеры – это тот фундамент, на котором покоится методика разработки через приемочные тесты, специфицирования через примеры, разработки на основе поведения – назовите, как хотите. С примерами, каким бы они ни были, можно работать всегда, вне зависимости от того, будете вы их потом автоматизировать или нет.

Как-то раз мне довелось применять эту методику в компании, которая разрабатывала продукт с применением разновидности методики водопада. Я работал в отделе системной интеграции. Мы занимались интеграцией системы тарификации и биллинга для мобильных теле-

фонных вызовов. Мы конфигурировали систему для конечного пользователя, а отдел разработки новой продукции разрабатывал продукт для более широкого контингента клиентов. Система допускала очень гибкую настройку тарифов на услуги мобильной связи. В какой-то момент отдел разработки решил полностью перепроектировать пользовательский интерфейс для конфигурирования продукта. К сожалению, обнаружилось препятствие – включенная в предыдущую версию функциональность технологического процесса прохождения и утверждения новых тарифов. Поскольку мы считались заказчиками и специалистами в предметной области настройки продукта, коллеги из отдела разработки обратились к нам.

Мы провели три встречи, на которых обсудили проблему и пришли к общему мнению относительно нового дизайна, который устроил бы оба отдела. Раньше в системе был определен технологический процесс ввода новых тарифов в систему. Этот процесс у реальных заказчиков по существу никогда и не использовался. В продукте, ориентированном на модульную структуру единого конфигурационного файла, наличие этого процесса только создавало бы лишние сложности. Компонент, отвечающий за конфигурирование, должен был бы поддерживать проверку корректности нескольких наборов параметров и зависимостей между ними, в том числе при любых изменениях. В конечном итоге это привело бы к полной неразберихе еще в и отделе тестирования.

На совещании, длившемся три часа, мы обсуждали технологический процесс в целом. Держа в уме новое решение, мы сформулировали примеры, описывающие будущий процесс с модульной конфигурацией. Когда спустя шесть месяцев появилась новая версия, все были довольны.

На той встрече мы выявили примеры самого верхнего уровня. У отдела тестирования не было возможностей для автоматизации тестов на таком уровне. Обсуждение одних лишь требований позволило правильнее подойти к реализации продукта и удовлетворить пожелания заказчика, то есть нашего отдела.

Вне зависимости от того, собираетесь ли вы автоматизировать примеры, необходимо обращать внимание на ряд моментов. В этой главе мы как раз и поговорим о том, на что обращать внимание при использовании примеров. Основные факторы, от которых зависит успешность применения методики, – это формат записи примеров, степень их детализации и внимание к различиям между реальным и тестируемым приложением.

Используйте подходящий формат

Выражая тесты в виде примеров, следует решить, как записывать сами примеры. На первый взгляд, кажется, что здесь нет ничего сложного, однако за прошедшее десятилетие в области ATDD были сформулированы некоторые паттерны.

Подходящий в конкретном случае формат зависит от команды, проекта и организации. Прежде всего, необходимо понять, кто будет читать созданные вами тесты. Вероятно, это вы сами, программист, которому поручено реализовать описываемую примерами функциональность, и владелец продукт или заказчик. Стоит также принять во внимание людей, который будут в дальнейшем сопровождать тесты, если вы собираетесь передать их другой команде, отделу или даже компании. Как бы то ни было, будущий читатель тестов, вероятно, захочет составить общее представление о той функциональности, которую вы сейчас описываете. Поэтому так важен единообразный способ выражения примеров.

Следует стремиться к тому, чтобы примеры были понятны каждой выявленной группе потребителей. Поэтому не нужно описывать технологические процессы в технических терминах, если они непонятны хотя бы части потенциальной аудитории. Примеры для контроллера светофоров можно было бы сформулировать следующим образом:

> *Если зеленый сигнал включен одновременно в направлении A и B, перевести контроллер светофора в безопасное состояние. В безопасном состоянии светофор постоянно мигает желтым светом. Вывести светофор из безопасного состояния может только специалист-ремонтник, предварительно проведя технический осмотр. Для анализа подобных ситуаций контроллер должен протоколировать все изменения состояния.*

Такой повествовательный подход характерен для традиционных спецификаций. А теперь еще раз посмотрим, в каком виде мы оформляли примеры. Сразу бросается в глаза, что мы не упоминали требование о протоколировании – быть может, оно перенесено на более поздний этап разработки. Однако мы отметили все случаи, когда необходим переход в безопасный режим. Такой декларативный стиль записи примеров даст гораздо больше информации в будущем, когда придется читать старые примеры и составлять новые. И, кроме того, эта информация послужит средством для общения с реальными заказчиками и даже с пользователями системы – например, с вами или со мной.

Каркасы автоматизации тестирования, ориентированные на ATDD, разделяют тестовые данные и код автоматизации, необходимый для исполнения тестов. Поскольку код автоматизации связывает разработку тестов с разработкой приложения, я намеренно называю его связующим кодом. Он может быть написан на разных языках с применением различных инструментов. В большинстве инструментов принято некое оглашение об именовании функций в связующем коде. Благодаря отделению тестовых данных от связующего кода можно определять тестовые примеры независимо от конкретной реализации приложения.

Существует много способов оформить требования к приложению в виде примеров. Самый популярный подход, появившийся в последние годы, заимствован из методики разработки на основе поведения (behavior-driven development – BDD). Этот формат основан на структуре Given-When-Then (дано–если–то), которая помогает описывать, что ожидается от конкретной функциональности.

Популярен также способ выражения примеров в виде таблицы. Вариантов несколько, но во всех встречаются три формы: форма отображения входных данных на выходные, форма запросов и форма действий (как в технологических процессах).

Третий способ записи примеров основан на использовании ключевых слов или тестов, управляемых данными. Ключевые слова позволяют комбинировать различные уровни абстракции в языке тестов. Если целью является тестирование низкого уровня приложения, то можно использовать низкоуровневые функции, но их можно также комбинировать для достижения более высокого уровня абстракции. Благодаря ключевым словам различные уровни абстракции представляются с помощью текучего интерфейса, образующего язык описания предметной области в тестах.

Разработка на основе поведения

Методика BDD впервые была описана в статье Дэна Норта (Dan North) [Nor06], где упомянута парадигма Given-When-Then. Хотя BDD не сводится к применению конструкции Given-When-Then как метода выражения функциональности, за прошедшие годы этот синтаксис практически стал синонимом BDD. Такой формат мы применяли к описанию примеров для стоянки в аэропорту. В качестве примера рассмотрим сценарий поиска в Google термина ATDD с ожиданием большого количества результатов (листинг 9.1).

Листинг 9.1. Простой поиск в Интернете

```
1  Функциональность: поиск в Google
2  Я являюсь пользователем Интернета и хочу использовать Google для поиска
3
4  Сценарий: Поиск термина ATDD
5  Given начальная страница Google
6  When я ищу 'ATDD'
7  Then я найду много результатов
```

В части Given записываются все предусловия одного конкретного поведения. Они составляют контекст. Здесь можно оговорить любые параметры. В строке 5 выше говорится, что я открыл в браузере начальную страницу Google. В примере для аэропорта части Given не было, потому что все тесты основывались на предположении, что в окне браузера открыт калькулятор стоимости парковки, и отходить от этого предположения не было никаких причин. В общем случае в части Given выражаются условия, отличающиеся от подразумеваемых по умолчанию. Например, если в системе имеются учетные записи разных типов, то можно было бы указать, к какому типу учетной записи относится данный пример. Для системы с различными веб-страницами можно сказать, какой страницей оперирует данный пример. Если речь идет о технологическом процессе в приложении, то можно описать все относящиеся к делу данные, которые должны быть введены на предыдущих шагах процесса – непосредственно до начала рассматриваемой в примере операции. Частей Given может быть несколько, и обычно они объединены связкой And (И).

В части When описывается операция и ее параметры. В строке 6 выше мы говорим, что ищем конкретный термин. В данном случае операция сводится к вводу текста в поле на веб-странице и нажатию кнопки по умолчанию. В других примерах речь могла бы идти о зачислении денег на счет в некоторой системе, о вводе различных значений в поля веб-формы или о переходе к процедуре оформления заказа в Интернет-магазине после заполнения корзины. В части When должно быть описано некое действие [dFAdF10]. Это может быть конкретное действие пользователя, событие, произошедшее вне текущей подсистемы (например, асинхронное сообщение от сторонней Интернет-службы или таймаут). Кроме того, в описании каждого сценария должно упоминаться только одно действие, чтобы сценарий оставался четко сфокусированным.

В части Then описываются постусловия после выполнения действия из части When. В большинстве случаев это утверждения о по-

ведении системы. В строке 7 примера выше мы проверяем, что получено много результатов поиска. В случае зачисления денег можно было бы проверить, что система добавила на счет бонусную сумму. В случае веб-формы, например парковочного калькулятора, можно проверить, что рассчитанная стоимость совпадает с ожидаемой. Как и в части Given, несколько предложений Then можно объединять связкой And.

Одна из заповедей разработки на основе поведения – разработка «снаружи внутрь». В BDD поощряется разработка кода на основе требований заказчика. Целью рабочей встречи по парковкам в аэропорту была постановка задачи о калькуляторе на основе бизнес-правил. Команда достигла этой цели, выразив требования и не испортив реализацию кода каким-то предвзятым решением. Задача формулирования требований заключается в том, чтобы исследовать все пространство возможных решений [GW89], которые удовлетворяют бизнес-правилам. По ходу проектирования ищется наилучший компромисс между всеми параметрами решения. Команда, собранная компанией Major International Airport Corp., решила эту задачу, абстрагировав выявленные примеры без привязки к какой-то конкретной реализации пользовательского интерфейса. На самом деле, эти примеры можно реализовать в контексте любого интерфейса, подставив новые определения шагов. Поскольку бизнес-правила вряд ли изменятся при изменении интерфейса, все основанные на них примеры по-прежнему останутся в силе.

Табличные форматы

Из подходов, в которых используются табличные форматы, пожалуй, самым популярным является каркас Framework for Integrated Tests [MC05]. Кроме того, в 2008 Роберт К. Мартин включил в систему FitNesse средство Simple List Invocation Method (SLiM) [Mar08b], позволяющее использовать похожую табличную структуру для выражения требований. В обоих вариантах есть три наиболее употребительных стиля: таблицы, в которых описываются входные данные и ожидаемые выходные данные; таблицы для опроса системы и сравнения наборов значений с теми, что получены от тестируемой системы; таблицы, поддерживающие технологические процессы.

Таблицы решений. Первый набор таблиц принимает входные данные, исполняет какие-то действия в системе и сравнивает получившиеся выходные данные с ожидаемыми. Такие таблицы встречались в примере применения SLiM к системе управления свето-

форами, где мы назвали их таблицами решений. В каркасе FIT они называются столбцовыми фикстурами (ColumnFixture), а в библиотеке FitLibrary – расширении FIT – вычислительными фикстурами (CalculateFixture).

С помощью таблиц решений можно выразить большинство требований. В частности, таблицы в примерах для парковки в аэропорту были таблицами решений. В них входом являлось время стоянки, а получаемым от системы выходом – стоимость парковки; ее и проверял исполнитель тестов. В листинге 9.2 приведены примеры для парковки с доставкой в указанное клиентом место, представленные в виде таблицы решений. Знакомо?

Листинг 9.2. Тесты для парковки с доставкой в указанное место, представленные в виде таблицы решений в SLiM

```
!|Parking costs for|Valet Parking |
|parking duration  |parking costs?|
|30 minutes        |$ 12.00       |
|3 hours           |$ 12.00       |
|5 hours           |$ 12.00       |
|5 hours 1 minute  |$ 18.00       |
|12 hours          |$ 18.00       |
|24 hours          |$ 18.00       |
|1 day 1 minute    |$ 36.00       |
|3 days            |$ 54.00       |
|1 week            |$ 126.00      |
```

Не требуется, чтобы было только одно входное или выходное значение. На самом деле, если вообще опустить выходные значения, то получается частный случай таблицы, служащей для задания параметров системы. Это удобно, например, если требуется описать учетные записи, с которыми система будет работать в дальнейшем. В листинге 9.3 приведен соответствующий пример. Такая специальная таблица решений без выходных данных часто называется настроечной таблицей, или SetUpFixture.

Листинг 9.3. Настроечная таблица для подготовки трех учетных записей

```
!|Account Creator                          |
|account name  |account state|role         |
|Susi Service  |active       |service user|
|Tim Technician|active       |technician   |
|Uwe Unemployed|unemployed   |service user|
```

Хотя на количество столбцов в таблице не налагается никаких ограничений, на практике стараются избегать таблиц, содержащих больше 10 столбцов, поскольку их очень трудно читать. Мне встреча-

лись тесты, выраженные с помощью таблиц, содержащих порядка 30 столбцов и нескольких сотен строк[1]. Беда в том, что такие тесты практически невозможно понять. Человек, которому предстоит сопровождать тесты, – не забывайте, что это можете быть и вы сами, – вряд ли за несколько секунд разберется, что означает конкретная строка. Держитесь от таких «ароматных» тестов подальше[2].

Таблицы запросов. Таблицы запросов применяются для сверки получаемых от системы коллекций. В SLiM они называются Query Tables, в FIT – RowFixture, а в FitLibrary – ArrayFixture, SetFixture и SubsetFixture. Часто возникает необходимость проверить порядок элементов в коллекции или убедиться, что она содержит некоторое подмножество. В SLiM частные случаи могут снабжаться префиксом Subset (подмножество) или Ordered (упорядоченное). В FitLibrary имеются классы, которым требуется унаследовать. В листинге 9.4 приведен пример проверки всех зарегистрированных в системе пользователей.

Листинг 9.4. Таблица запроса, в которой проверяется существование введенных ранее данных

```
!|Query:Users in the System                    |
|user name          |role           |user state|
|Tim Tester         |Tester         |active    |
|Paul Programmer    |Programmer     |active    |
|Petra Projectmanager|Project Manager|active    |
```

Таблицы запросов используются после сбора каких-то данных или получения хранящихся в системе сущностей. Например, они могут быть полезны для проверки учетных записей, полученных в результате операции поиска. В случае Интернет-магазина можно опросить корзину заказчика и сравнить ее содержимое с ожидаемым.

Можно проверять значение какого-то одного атрибута элемента коллекции, например его название, или сочетание названия, цены и количества товара в корзине. Обычно в описании таблицы указывается, какие атрибуты проверять. Детальность таблиц определяет, насколько тщательно проверяются полученные от системы данные.

Чем больше данных описано в таблицах, тем больше изменений придется вносить в случае изменения имени атрибута. Проектируя

1 Чтобы сберечь деревья на планете, я не включаю этот пример. Да и вообще не хочу заводить вас не в ту сторону, демонстрируя столь ужасающую конструкцию.

2 В книге «Refactoring: Improving the Design of Existing Code» [FBB+99] Фаулер и Бек называют «ароматами дизайна» нечто не совсем правильное в разработанном проекте. Термин «аромат теста» встречается также в книге «xUnit Test Patterns: Refactoring Test Code» [Mes07].

тесты, учитывайте компромисс между тщательностью тестирования приложения и трудоемкостью сопровождения набора тестов в будущем. С одной стороны, желательно включать в таблицу как можно меньше деталей, но, с другой стороны, при этом возрастает вероятность, что какие-то ошибки останутся незамеченными. Снизить этот риск можно за счет добавления в набор тестов примеров, проверяющих опущенные детали или резервирования времени для исследовательского тестирования. Разумеется, у обоих решений есть свои минусы. Можно пойти и другим путем – включить все мыслимые детали. Но при этом тесты становятся уязвимы, например, относительно переименования атрибутов. Между этими крайностями есть и другие походы к решению задачи. Но в любом случае в каждой конкретной ситуации предстоит определиться с количеством деталей, включаемых в таблицы.

Вероятно, проще всего довериться интуиции и время от времени пересматривать решение. Если выясняется, что приходится сопровождать слишком много тестов, то стоит вернуться назад и подвергнуть тесты рефакторингу.

Таблицы скриптов. Таблицы скриптов применяются для описания технологических процессов в системе. В одном тесте могут сочетаться таблицы скриптов, таблицы решений и таблицы запросов. Альтернативой стилю Given-When-Then, принятому в BDD, является табличный формат, известный под названиями Setup-Execute-Verify (Подготовить–Выполнить–Проверить) или Arrange-Act-Assert (Подготовка–Действие–Утверждение). На стадии подготовки, чтобы задать предусловия, я часто использую таблицы решений без выходных данных. Затем я вызываю одну операцию в таблице скриптов, например пополнение баланса только что подготовленной учетной записи. И на завершающем шаге проверки я с помощью таблицы запросов получаю все балансы, хранящиеся в моей учетной записи, и сравниваю значения, получившиеся после пополнения. Полный пример приведен в листинге 9.5.

Листинг 9.5. Таблица скриптов и полное описание одной операции в системе

```
1  !|script|Account Reload|
2
3  !|AccountCreator          |
4  |account name   |tariff |
5  |prepaid account|prepaid|
6
7  |reload|50.00 EUR|on|prepaid account|
```

```
8
9 !|Query:BalanceChecker|prepaid account|
10 |balance name        |balance value  |
11 |Main balance        |50.00 EUR      |
12 |Bonus balance       |5.00 EUR       |
```

То, что в SLiM называется таблицей скриптов, в FIT известно под названием ActionFixture, а в FitLibrary – под названием DoFixture. Вообще говоря, с их помощью можно выразить в таблице тестов всё что угодно. В каждом инструментальном средстве действуют свои соглашения об именовании функций в связующем коде. Обычно имеется какой-то способ отделить параметры функции от текста. Для образования имени вызываемой функции текст конкатенируется – с применением «верблюжьей нотации»[3].

Таблицы скриптов полезны, когда требуется описать более обширный набор операций. Например, с их помощью можно выразить такой сценарий: запустить систему, ввести какие-то значения, нажать кнопку, сравнить значения и остановить систему. Нередко в тестах встречаются комбинации различных таблиц.

Автоматизация, управляемая ключевыми словами

Чтобы выразить комбинирование нескольких операций, можно использовать ключевые слова. Операция, включающая несколько ключевых слов, становится операцией более высокого уровня. Для моделирования в тестах нужного уровня абстракции можно комбинировать ключевые слова, определенные на нескольких разных уровнях.

В примере системы управления светофорами нам встречался сценарий `invalid combination`, который позволял выразить концепцию верхнего уровня с помощью отдельного ключевого слова (листинг 9.6). В отчете о выполнении теста можно было бы проследить вхождения этого ключевого слова. Из популярных каркасов, основанных на ключевых словах, упомянем еще Robot Framework[4]. В его состав входит целый ряд библиотек для тестирования и дополнительных модулей для тестирования через веб, поддержки SSH, баз данных и Swing.

---

3 Под «верблюжьей нотацией» понимается определенный выбор регистра букв в начале слов для образования имени функции, не содержащего пробелов. Например, из текста This is fun получается имя функции thisIsFun.

4 http://robotframework.org

Листинг 9.6. Использование таблицы сценария как ключевого слова в примере системы управления светофорами (воспроизведение листинга 7.15)

```
...
!2 Недопустимые комбинации

!|scenario       |invalid combination|firstLight||secondLight|
|set first light |@firstLight                                  |
|set second light|@secondLight                                 |
|execute                                                       |
|check           |first light        |yellow blink            |
|check           |second light       |yellow blink            |

!|script|FirstLightSwitchingCrossingController|

!|invalid combination    |
|firstLight  |secondLight|
|green       |red, yellow|
|green       |green      |
|green       |yellow     |
|yellow      |red, yellow|
|yellow      |green      |
|yellow      |yellow     |
|red, yellow |red, yellow|
|red, yellow |green      |
|red, yellow |yellow     |
|red         |red, yellow|
|red         |green      |
|red         |yellow     |
|yellow blink|red        |
|yellow blink|red, yellow|
|yellow blink|green      |
|yellow blink|yellow     |
```

При использовании ключевых слов граница между представлением тестовых данных и кодом автоматизации тестов размывается. Можно использовать ключевые слова, реализованные на языке программирования, применяемом для тестирования приложения, а также ключевые слова для взаимодействия с низкоуровневой функциональностью операционной системы. Можно также конструировать новые ключевые слова из ключевых слов более низкого уровня. А эти ключевые слова в свою очередь комбинировать различными способами и таким образом получать несколько уровней ключевых слов.

Такой подход дает возможность организовывать тестовые данные в виде набора низкоуровневых драйверов автоматизации тестирования. За счет определения новых комбинаций ключевых слов язык автоматизации развивается и становится все более мощным. Но есть

и недостатки – появление глубоко вложенных и трудных для понимания уровней ключевых слов. На момент написания этой книги почти не существовало инструментов рефакторинга, ориентированных на ключевые слова. Поэтому поддержка нескольких уровней ключевых слов может привести к неразберихе. Конечно, найти и заменить текст в нескольких файлах несложно, но текучий стиль представления тестовых данных затрудняет нахождение отдельных вхождений. Например, переименование ключевого слова или расширение списка его параметров может оказаться проблемой. Но теперь большинство современных IDE поддерживают такие способы рефакторинга безопасно. В перспективе есть надежда на появление новых инструментов рефакторинга, ориентированных на автоматизацию, управляемую ключевыми словами, – и даже поддерживающих одновременно низко- и высокоуровневые ключевые слова.

Связующий и вспомогательный код

Конкретный способ автоматизации примеров сильно зависит от используемого каркаса. Например, каркасы, написанные на Java, опираются на механизм аннотаций для поиска кода, исполняющего данную фразу примера. В других каркасах применяются имена функций в верблюжьей нотации. Некоторые каркасы поддерживают методы установки и чтения свойств класса (как в Java Beans), например имени и фамилии владельца учетной записи.

Но вне зависимости от каркаса или инструмента, применяемого для автоматизации примеров, всегда следует помнить, что вы разрабатываете связующий код. Некоторые команды забывают, что создание связующего и вспомогательного кода – частный случай разработки ПО. Когда приложение впервые подвергается тестированию, написанный код может казаться простым, а его сопровождение тривиальным. Но со временем, если не применять к связующему и вспомогательному коду те же принципы, что к разработке продуктового кода, его сопровождение может обернуться кошмаром.

Лично для меня это обычно означает, что начинать надо с простого. Я хочу как можно скорее приступить к прогону автоматизированных приемочных тестов. Как-то мы вдвоем смогли подготовить кое-какие автоматизированные тесты для заказчика, потратив всего-то пару часов. Еще через два часа мы имели общее представление о коде драйвера для прогона тестов и базовый слой связующего кода. Мы понимали, что, когда мы перейдем к более сложным тестам, этот код придется развить.

Со временем, по мере появления всё новой и новой функциональности я приступаю к активному выделению новых компонентов из уже написанного кода. В части II на примере `LightStateEditor` мы видели, что такие компоненты добавляются с помощью методики разработки через тестирование. Фактически мы тестировали тестовый код. В одной компании, где мне довелось работать, мы даже применяли к коду автоматизации различные метрики, в частности, статический анализ и кодовое покрытие. В системе непрерывной интеграции мы завели для этой цели отдельный проект с собственной системой сборки. Сравнив метрики тестового и продуктового кода, мы обнаружили, что бьем последний по всем параметрам. Этот код автоматизации тестов до сих пор – по прошествии двух лет – используется и продолжает совершенствоваться.

Если вы полагаете, что это перебор, позвольте рассказать историю, которую я услышал в 2009 году. В одну команду пригласили внешнего подрядчика для автоматизации тестирования программы. Подрядчик испытывал некоторую тревогу в связи с необходимостью добавлять вспомогательный код со сложной логикой. Поэтому при его написании он применил методику разработки через тестирование. По завершении работы подрядчика покрытие кода приложения тестами возросло на 10 % и составило 15 %.

Разработка кода автоматизации – это разработка ПО. Поэтому к связующему и вспомогательному коду следует применять те же принципы, что к разработке продуктового кода. Мне доводилось видеть код автоматизации тестирования, который никак не проектировался, был плохо документирован и даже изобиловал ошибками. Работая над новым подходом к автоматизации тестирования в одной компании, мы преодолевали эти недостатки, применяя инкрементное проектирование в сочетании с разработкой через тестирование. Последнее помогло нам лучше понять код и одновременно предоставить необходимую документацию. Использование TDD также гарантировало, что код автоматизации не содержит ошибок и, значит, не дает ни ложноположительных утверждений (когда тест завершается успешно несмотря на то, что программа некорректна), ни ложноотрицательных (когда тест не проходит, хотя программа работает правильно).

Необходимость добавления автономных тестов для связующего и вспомогательного кода становится очевидной, если использовать ATDD так, как описано в части II. Работая над приемочными тестами, мы раскрыли для себя предметную область приложения. Обладая

знаниями и опытом в предметной области, мы смогли извлечь предметный код из связующего. Не будь у нас в тот момент автономных тестов для связующего кода, нам пришлось бы либо добавить их перед выделением, либо написать новый предметный код, применяя разработку через тестирование.

Подходящий формат

Хотелось бы сказать несколько слов о подходящем формате. Все вышеупомянутые форматы – взятый из методики BDD, табличный и на основе ключевых слов – применялись на практике, когда я писал эту книгу. За прошедшее десятилетие большинство имеющихся сегодня инструментов подверглось переработке для поддержки тестов, представленных в том или ином из этих форматов. Например, каркас SLiM, являющийся частью FitNesse, предполагает использование таблиц решений и запросов. Но с помощью таблиц скриптов SLiM поддерживает также формат Given-When-Then. В каркасе Robot Framework имеются аналогичные способы использования каждого из трех рассмотренных форматов. Какое бы сочетание формата представления тестовых данных – BDD, таблицы или ключевые слова – и какой бы язык автоматизации тестов вы ни выбрали, скорее всего, найдется хотя бы один каркас, отвечающий вашим предпочтениям.

И это замечательно, потому что позволяет сосредоточиться на использовании подходящего формата для выражения примеров наиболее естественным способом и отложить решение о выборе инструментария на более позднее время. Большинство команд, начинавших с внедрения конкретного инструмента, этим и закончили: внедрили инструмент, а не успешную методику [Adz11]. Очевидно, выбор инструмента автоматизированного тестирования – не самая важная часть стратегии [KBP01].

Примеры, а равно и тесты, получившиеся после автоматизации примеров, представляют интерес для нескольких сторон. Во-первых, это программист, разрабатывающий функциональность, который должен понять пример по его описанию. Во-вторых, это программист, которому поручена автоматизация конкретного примера. В-третьих, это заказчик или владелец продукта, который также будет читать уже автоматизированный тест на этапе анализа кода. В-четвертых, это команда в целом, которая будет обращаться к тестам при планировании следующих итераций, чтобы оценить побочные эффекты. И наконец, это человек, которому предстоит сопровождать тесты и при необходимости вносить в них изменения.

Все заинтересованные стороны не должны тратить много времени, чтобы понять назначение примера и тестов. При первом же взгляде на тест должен быть ясен его контекст и конкретная цель. Время, потраченное на чтение и понимание тестов, потеряно для продуктивной работы. А чем меньше продуктивной работы, тем ощутимее давление извне. А чем сильнее давление, тем вероятнее, что в тестах вы будете срезать углы и искать легких путей. А чем больше срезано углов, тем менее понятны тесты (рис. 9.1). И тут-то вы попадаете в порочный круг. Можно ставить на себе крест.

Чтобы разорвать этот порочный круг, необходимо осознать свою роль в системе. Существует одна очевидная точка принятия решения, в которой определяется, станет ли система порочным кругом или контуром с положительной обратной связью: решение потратить время на чтение приемочных тестов [Wei91].

Рис. 9.1. Влияние понятности назначения тестов на систему в целом

Приняв противоположное решение, вы потратите меньше времени на уяснение смысла приемочных тестов. Тем меньше времени потрачено на это занятие, тем больше его останется для продуктивной работы. Это устраняет давление извне и позволяет не срезать углы в тестах.

Команда использует приемочные тесты как средство достижения взаимопонимания. Разнообразие заинтересованных сторон делает этот факт очевидным. В процессе разработки приложение постоянно переходит из рук в руки. Успешные команды уделяют большое внимание понятности тестов, чтобы уменьшить возникающее при таком

переходе трение. В 2009 года я услышал от Энрике Комба-Рипенхаузена (Enrique Comba-Riepenhausen) историю о том, что заказчик смог прочитать и понять не только его приемочные тесты, но также автономные и даже предметный код. Его команда успешно внедрила концепцию, которую Эрик Эванс назвал общеупотребительным языком [Eva03].

В каждом проекте должен быть некий единый, общеупотребительный язык. Чем чаще требуется перевод с одного языка на другой, тем больше возможностей для недопонимания. Если предметная область описывается на том же языке, что приемочные тесты, то есть уверенность, что заказчик сможет понять тесты. Если на том же языке моделируются еще и понятия, присутствующие в предметном коде, то можно быть уверенным, что все понимают задачу одинаково, и, стало быть, избежать недоразумений впоследствии, когда уже слишком поздно что-то исправлять.

Уточнение примеров

Чтобы получить адекватное представление о функциональности, нужно вместе с заказчиком или владельцем продукта сформулировать существенные примеры. К сожалению, как мы видели в части I, на первой встрече могут быть выявлены не все примеры, необходимые для разработки программы. Сформировав первый комплект примеров, нужно затем уточнить их [Adz11].

Уточнять примеры можно по-разному. Обычно тестировщики знают, как выявить скрытые в первых примерах допущения и граничные условия. В зависимости от предметной области могут существовать ограничения на максимальную длину строки, различные правила проверки или взаимоисключающие бизнес-правила. Если подобные условия были упущены из виду на первой рабочей встрече по выработке спецификаций, имеет смысл попросить тестировщика уточнить примеры.

Другой пример уточнения примеров – наличие бизнес-правила, в котором владелец продукта не уверен. Получив разъяснения, владелец продукта, тестировщик и программист совместно уточняют примеры, сформулированные в первом раунде. Располагая дополнительной информацией, они могут составить более полное представление о функциональности, чем дают первые огрубленные примеры.

Будучи тестировщиком, вы, естественно, проверяете граничные случаи и применяете для уточнения примеров технику представи-

тельного тестирования или классы эквивалентности. На самом деле, для поиска упущений и уточнения примеров подойдет любая техника тестирования. В следующем разделе мы вкратце обсудим различные приемы тестирования, используемые для этой цели. Полное рассмотрение этой темы выходит за рамки данной книги. Если вас интересуют дополнительные сведения о технике тестирования, обратитесь к книгам Ли Коупленда (Lee Copeland) «A Practitioner's Guide to Software Test Design» [Cop04] или Кейнера (Kaner) «Testing Computer Software» [KFN99].

Представительное тестирование

Техника представительного тестирования (domain testing) подразумевает разбиение большой области определения на меньшие подобласти.

Все множество значений, подаваемых на вход системы, разбивается на несколько классов эквивалентности, в которых система ведет себя одинаково. Пример мы видели в задаче о стоимости парковки. Предварительное, грубое разбиение на классы эквивалентности отражало классификацию парковок по типам. Последующие классы определялись бизнес-правилами. Один класс соответствовал нескольким первым часам, другой – нескольким первым дням и еще один – неделям. Внутри каждого класса выбирались представительные данные: находящиеся точно на границе классов, точно в середине и непосредственно перед границей со следующим классом. Из этих данных формировались примеры.

Рассмотрим случай длительной парковки под открытым небом. Согласно бизнес-правилам, плата за стоянку составляет 2 доллара в час, но не более 10 долларов в сутки и не более 60 долларов в неделю. Первый класс эквивалентности для парковки этого типа определяется поведением в первые сутки до пятого часа включительно. Мы включили в набор примеров длительность стоянки 1 час, 3 часа и пять часов. Второй класс эквивалентности начинается с шестого часа и продолжается до конца суток. В качестве представителей этого класса выберем значения 5 часов и 1 минута, 10 часов и 24 часа. Можно выбрать аналогичные значения и для вторых суток, чтобы убедиться, что для них поведение аналогично. Таким образом, мы получаем комбинацию первых двух классов эквивалентности. Но для третьих и четвертых суток внутри этого класса поведение должно быть таким же. Наконец, по достижении недельного максимума в конце шестых суток начинается новый класс эквивалентности, который также можно комбинировать с несколькими неделями.

Утвержденные на рабочей встрече примеры отражают именно такое разбиение (табл. 9.1). Первые пять значений взяты из первого класса эквивалентности, определяемого почасовым тарифом. Следующие четыре значения соответствуют примерам из второго класса эквивалентности и в некоторых случаях комбинируются с первым классом. И последние шесть примеров представляют третий класс эквивалентности в комбинации с первыми двумя.

Таблица 9.1. Примеры для длительной парковки под открытым небом, сформулированные в конце рабочей встречи

Время парковки	Стоимость
30 минут	$2,00
1 час	$2,00
5 часов	$10,00
6 часов	$10,00
24 часа	$10,00
1 сутки 1 час	$12,00
1 сутки 3 часа	$16,00
1 сутки 6 часов	$20,00
6 суток	$60,00
6 суток 1 час	$60,00
7 суток	$60,00
1 неделя и 2 дня	$80,00
3 недели	$180,00

Граничные значения

Тестирование граничных значений упрощается, если заранее провести анализ предметной области. Идея в том, что ошибки особенно вероятны на границах классов эквивалентности. Поэтому традиционная теория рекомендует проверять одно значение до границы, другое на границе и третье после границы. В общем-то, простая математика, и, похоже, в примере выше мы в целом придерживались этой рекомендации.

Обратимся снова к задаче о парковке. Для парковки с доставкой в указанное место было выявлено два класса эквивалентности: в одном случае начислялась плата 18 долларов, в другом – 12 долларов. Граница между ними проходит по значению времени стоянки 5 часов.

Зная это, мы должны включить следующие тесты: время в точности равно 5 часам (граничное значение), четыре часа и 59 минут (до границы) и пять часов и 1 минута (после границы) (см. табл. 9.2).

Таблица 9.2. Граничные значения для парковки с доставкой

Время парковки	Стоимость
4 часа 59 минут	$12,00
5 часов	$12,00
5 часов 1 минута	$18,00

Разумеется, есть и другие (скрытые) граничные значения, например, время стоянки 0 минут. Смысл присутствия тестировщика на рабочей встрече и заключается в том, чтобы распознать такие скрытые граничные условия и классы эквивалентности, которые иначе остались бы подразумеваемыми.

Попарное тестирование

Идея попарного тестирования основана на наблюдении, что ошибки часто проявляются для комбинации двух разных входных значений. Если на вход приложения подается величина A, принимающая значения a1, a2, a3 и величина B, принимающая значения b1, b2, b3, то уверенность в результате можно повысить, рассмотрев случаи, показанные в табл. 9.3.

Таблица 9.3. Примеры попарного тестирования

A	B
a1	b1
a1	b2
a1	b3
a2	b1
a2	b2
a2	b3
a3	b1
a3	b2
a3	b3

Если добавить еще третью величину C, принимающую значения c1 и c2, то получатся случаи, показанные в табл. 9.4.

Таблица 9.4. Примеры попарного тестирования с тремя переменными

A	B	C
a1	b1	c1
a1	b2	c2
a1	b3	c2
a2	b1	c1
a2	b2	c1
a2	b3	c2
a3	b1	c2
a3	b2	c1
a3	b3	c1

Алгоритм гарантирует выбор тестовых примеров, содержащих все комбинации значений двух переменных. Для комбинаторных задач такой подход помогает сократить количество прогоняемых тестов, сохранив базовое покрытие.

В случае задачи о парковке этот подход можно применить к классам эквивалентности. Например, можно включить примеры для 0, 1 и 3 недель, для 0, 1, 3, 6 и 7 дней и для 0, 1, 3, 5 и 6 часов. Алгоритм попарного тестирования даст тестовые примеры, перечисленные в табл. 9.5.

Таблица 9.5. Примеры попарного тестирования для задачи о парковке

Недели	Дни	Часы
0	0	0
1	0	1
3	0	3
1	1	0
0	1	1
0	1	3
3	1	5
3	3	0
0	3	1
1	3	3
3	6	1
0	6	5
1	6	6
1	7	5

Недели	Дни	Часы
0	7	6
3	7	0
3	0	6
1	0	5
3	1	6
0	3	5
1	3	6
3	6	0
0	6	3
3	7	1
1	7	3

Конечно, для всех комбинаций необходимо вычислить еще и стоимость парковки, но это я оставляю в качестве упражнения для читателя.

Сокращение набора тестов

Со временем набор тестов разрастается. Существуют два эвристических граничных значения, определяемых временем выполнения тестового набора.

Первая граница проявляется, когда общее время выполнения превышает несколько минут. Пока оно не столь велико, вы не испытываете дискомфорта, регулярно прогоняя приемочные тесты – почти с той же частотой, что автономные. Но если для их выполнения требуется порядка 10 минут и больше, то это начинает раздражать. Вы начинаете прогонять приемочные тесты реже – как правило, только перед очередным сохранением в системе управления версиями. Это уменьшает ценность обратной связи, обеспечиваемой тестами, и может привести к тому, что в коде останутся незамеченные в течение какого-то времени ошибки. А при работе в команде такой некорректный код может быть даже интегрирован коллегой вместе с последними изменениями – дополнительный источник неразберихи.

Вторая психологическая граница проявляется, когда время выполнения регрессионных тестов достигает двух-трех часов. Лайза Криспин (Lisa Crispin) говорит, что ее команда стремится к тому, чтобы время выполнения неавтономных тестов не превышало 45 минут, иначе программисты перестают обращать внимание на результа-

ты. Пока эта граница не достигнута, можно считать, что с прогоном тестов всё обстоит нормально. Но за ней качество тестов начинает падать. Все больше и больше тестов не проходят при непрерывной интеграции [DMG07]. Начиная с этого момента, приходится тратить все больше рабочего времени на повторный прогон тестов, не прошедших при ночной сборке. Это оставляет все меньше времени на создание новых тестов и исправление ошибок в уже имеющихся. Все силы уходят на «движение назад с завязанными глазами» и, как у лягушки, плавающей в медленно нагреваемой воде, остается все меньше шансов выпрыгнуть до наступления смерти [Wei01, стр. 125].

Последней существование воды признает рыба.

Чтобы предотвратить такое развитие событий, мы должны искать возможность ускорить выполнение тестов. Иногда это означает реорганизацию приемочных тестов с применением интерфейса более низкого уровня – например, использование в архитектуре MVC уровня модели, а не представления (пользовательского интерфейса). Какие-то части медленных подсистем, например базу данных или сторонние компоненты, возможно, придется имитировать. Какое-то время такое решение будет работать, но у него есть по меньшей мере два серьезных недостатка. Во-первых, вы увеличиваете разрыв между вариантом приложения, который проверяют тесты, и вариантом, с которым будут работать пользователи. В следующем разделе «Учет упущений» мы рассмотрим этот вопрос подробнее. Во-вторых, в следующий раз, когда будет достигнуто предельное время выполнения, диапазон ваших возможностей сузится. А если все возможности ускорить выполнение примеров исчерпаны, то остается поступить, как Королева в «Приключениях Алисы в Стране чудес» Льюиса Кэролла, которая, как известно, приказала: «Отрубить им головы!» [Car65].

Такое решение может показаться чересчур радикальным, но в действительности регрессионные тесты обнаруживают лишь 23 % ошибок [Jon07]. Автоматизированные регрессионные тесты находят лишь малую долю ошибок [KBP01]. Поскольку ваши примеры могут рассматриваться как регрессионные тесты, то, пожалуй, стоит считать, что они выполняют свою задачу, если обнаруживают 23 % ошибок при регрессии. Разумеется, проблема в том, что заранее редко известно, какие именно из имеющихся тестов находят эти пресловутые 23 %, – чтобы это узнать, тесты необходимо сначала реализовать. Именно поэтому команда начинает с создания большого набора тестов. И хотя на первых порах наличие большого набора вселяет теплое

чувство уверенности, при достижении второй границы времени выполнения регрессионного набора он, очевидно, становится помехой.

Мне вспоминается проект, в котором набор регрессионных тестов исполнялся около 36 часов. Мы не знали, сколько точно, потому что ни разу не смогли прогнать его от начала до конца. Количество тестов ошеломляло, но свою цель они давно перестали выполнять. Через два месяца мы оставили от этого набора только 10 %, что позволило быстро узнавать о вновь внесенных ошибках. Нечего и говорить, что, не пойдя на такой шаг, мы подписали бы себе смертный приговор.

Для сокращения количества примеров могут пригодиться стандартные приемы тестирования. Оглядываясь назад, я понимаю, что мы могли бы применить к этому 36-часовому набору технику попарного тестирования. Напомню, что смысл ее в том, чтобы комбинировать любые два фактора только один раз. Так можно резко сократить количество тестов, сохранив приемлемое покрытие бизнес-правил.

Если вы уже применили какой-то метод сокращения количества тестов, например уменьшение комбинаторного числа вариантов, а время выполнения по-прежнему превышает пять часов, то можно пойти на разбиение набора. Некоторые команды организуют несколько сравнительно небольших наборов тестов, помечая их признаками, описывающими определенные части приложения. Затем можно применить к наборам тестов технику поэтапной сборки [DMG07]. В этом случае вы сначала исполняете небольшой набор тестов для частей приложения, в которых риск наличия ошибок максимален. Если все они прошли, то можно переходить к большему набору для тестирования прочих частей.

Наконец, есть и такие команды, которые автоматизируют тесты параллельно с разработкой продуктового кода в ходе одной итерации. После того как все тесты пройдут, большая их часть удаляется, а остается только минимальное число регрессионных тестов, включаемых в автоматизированный набор. Гойко Аджич в своей книге «Specification by Example» [Adz11] описывает несколько таких случаев, о которых узнал в ходе опроса 50 команд в 2010 году. По существу, команда всеми силами стремилась к тому, чтобы время выполнения тестов не вышло за пределы первой границы. Они принесли покрытие в жертву быстрому получению результата – или даже изначально не претендовали на сколько-нибудь полное покрытие. Такой подход приемлем, если не забывать об упущениях.

Учет упущений

Автоматизируя тесты, следует учитывать не обнаруживаемые ошибки [KBP01]. Этот урок касается, прежде всего, «цены выбора» при автоматизации тестирования. На каждый тест, который мы автоматизируем, приходится множество тестов, которые никогда не выполняются – ни в автоматическом, ни в ручном режиме. Смысл учета упущений в методике ATDD заключается в том, чтобы найти правильный баланс между автоматизацией тестов и ручным тестированием тех частей приложения, которые не покрываются автоматизированными тестами. Одного лишь автоматизированного тестирования недостаточно. Рассмотрим, к примеру, приложение, которое позволяет вводить персональные данные клиентов. Автоматизированные тесты покрывают создание одного клиента, проверку бизнес-правил и зависимостей между полями, например, ввод неправильных почтовых индексов или несоответствие между почтовым индексом и штатом.

В этом гипотетическом сценарии никакие ручные тесты не были предусмотрены. В итоге после выпуска приложения посыпались жалобы от пользователей. Почти все говорят, что приложение трудно использовать, потому что порядок навигации по полям формы неестественный. Когда вводится только одно значение, проблема не проявляется, но при вводе тысяч записей о клиентах она встает во весь рост.

Как-то раз мне пришлось работать с таким приложением. Проблема заключалась в порядке перехода между полями по нажатию клавиши табуляции на странице ввода данных. После ввода фамилии и нажатия табуляции курсор оказывался не в поле имени, а в поле даты рождения. Раньше, когда использовалось приложение для DOS, проблема с порядком табуляции вообще не возникала. Но после переноса на платформу Windows самого приложения и 300 учетных записей я заметил эту неприятность, которая довольно быстро стала мне поперек горла.

Что бы вы должны были сделать на месте поставщика такого приложения? Конечно, можно было бы описать функциональность порядка табуляции на примерах и потом эти примеры автоматизировать. При каждом внесении в код изменений ошибки такого рода обнаруживались бы. Проблема в том, что подобный тест по необходимости тесно связан с конкретной реализацией пользовательского интерфейса. При добавлении на экран ввода нового поля пришлось бы изменять и тест.

Альтернатива – заранее отвести какое-то время на исследовательское тестирование [KFN99]. В течение этого времени можно исследовать дизайн пользовательского интерфейса с точки зрения удобства работы. После каждого внесения изменений в пользовательский интерфейс это мероприятие следует повторять.

Когда я представлял слушателям проводимых мной курсов и семинаров эти альтернативы, почти все выбирали исследовательское тестирование, обосновывая это высокими затратами на создание и сопровождение автоматизированных тестов и их низкой окупаемостью.

Хотя следующий пример может показаться нетипичным, Юрген Аппело (Jurgen Appelo) все же приводит его в курсе «Как управлять и быть лидером», упоминая «неизвестные неизвестные». Многие руководители совершают одну и ту же ошибку, полагая что учли все риски [App11]. Веками ученые считали, что все лебеди белые. Открытие черного лебедя доказало их неправоту. Труднопредсказуемые и редкие события могут иметь важные последствия [Tal10]. В действительности уверенность в том, что катастрофа невозможна, довольно часто приводит к непредвиденной катастрофе. Джерри Вейнберг придумал для этого феномена термин «эффект Титаника» [Wei91].

Если вы полагаете, что учли все упущения, то, скорее всего, забыли о черном лебеде – неизвестном неизвестном. Не полагайтесь на единственный подход к тестированию ПО, учитывайте возможность, что одни лишь автоматизированные тесты не решают все проблемы. В вашем тестовом ансамбле обязательно будут какие-то прорехи, но вы, как дирижер, должны представить публике полный оркестр.

Сбор оркестра для тестирования

Матрица квадрантов тестирования, подробно обсуждаемая в книге Лайзы Криспин (Lisa Crispin) и Джанет Грегори (Janet Gregory) «Agile Testing» [CG09] и впервые описанная Брайаном Мариком (Brian Marick) [Mar03], может навести на мысли о том, какие еще способы тестирования приложения позволят добиться сбалансированности. В двух словах, все виды тестирования можно распределить по двум направлениям. По первому откладываются тесты, направленные на поддержку команды, разрабатывающей проект, или на проверку соответствия продукта ожиданиям заказчика, а по второму, ортогональному, – технологически-ориентированные и бизнес-ориентированные тесты (рис. 9.2).

Виды тестирования располагаются в четырех квадрантах. В первом квадранте находятся технологические тесты, направленные на

поддержку команды. В эту категорию попадают низкоуровневые автономные тесты и тесты межмодульных сопряжений. Во втором квадранте находятся бизнес-ориентированные тесты, направленные на поддержку команды. Это, в частности, автоматизированные тесты, рассматриваемые в этой книге. В третьем квадранте находятся бизнес-ориентированные тесты, направленные на проверку соответствия продукта ожиданиям и требованиям. Это альфа и бета-тесты, но также приемочные тесты, проводимые пользователем, тесты удобства работы и исследовательские тесты. И наконец, в четвертый квадрант попадают тесты технического характера, направленные на проверку соответствия продукта ожиданиям и требованиям. Из наиболее значимых тестов в этом квадранте отметим тесты производительности и нагрузочные тесты.

Рис. 9.2. Квадранты тестирования, помогающие выявить упущения в методике тестирования

Как видите, приемочные тесты, создаваемые как побочный продукт применения ATDD, относятся ко второму квадранту. Если сосредоточиться только на бизнес-ориентированных тестах для поддержки команды, то, вероятно, будут упущены из виду проблемы производительности, удобства работы и даже просто качества кода, которые в перспективе могут привести к неприятностям[5]. Забыть о

5 В литературе это часто называют «техническим долгом» . Термин был предложен Уордом Каннингемом (Ward Cunningham) в 1992 году: http://c2.com/cgi/wiki?TechnicalDebt.

трех оставшихся квадрантах – значит, оставить проект уязвимым для самых разнообразных рисков.

И напоследок – если вы не согласны со мной по этому вопросу, то можете обратиться к странице калькулятора стоимости парковки, которая создана на основе реальных требований. Все эти требования выполнены в версии, на которую я ссылаюсь в этой книге. Но хотя все приемочные тесты проходят, в калькуляторе имеются тонкие, а иногда даже явные ошибки. Посмотрите, сколько ошибок вы сможете найти за полчаса, а потом перечитайте этот раздел – быть может, полученный опыт изменит ваше мнение.

Резюме

Раздумывая о том, как подступиться к ATDD, попробуйте представить свои данные несколькими способами. Сначала запишите их в виде таблицы, а затем выразите то же самое в стиле BDD. Только после того как будет найдено представление, удовлетворяющее вас, вашу команду и представителя бизнеса, начинайте поиск каркаса, отвечающего вашим целям. Кроме того, решение о выборе каркаса должна принимать вся команда. Следует также иметь в виду, что различные исполнители тестов – например, Selenium или Watir для веб-страниц или SWTBot для SWT-приложений – имеют разные побочные эффекты. Начните с самого многообещающего варианта, реализуйте один тест от начала до конца, а затем подумайте, не стоит ли пересмотреть решение. Приобретя даже минимальный опыт, вы будете яснее видеть достоинства и недостатки и сможете принимать решение осознанно.

Выписать первый набор примеров обычно недостаточно. Используйте свои знания о проектировании тестов, чтобы уточнить этот набор. В числе полезных приемов можно упомянуть тестирование граничных значений, попарное и представительное тестирование. Ближе познакомиться с различными техниками тестирования позволит книга «A Practitioner's Guide to Software Test Design» [Cop04].

Со временем набор тестов разрастается. Наступает момент, когда на прогон всех тестов уходит слишком много времени. На какой-то период проблему можно решить, разбив всё множество тестов на несколько поднаборов или прогоняя их ночью. Но рано или поздно от некоторых тестов приходится избавляться. Принято считать, что психологической границей является время исполнения 90 минут. Со-

берите всю команду и подумайте, как получить от приемочных тестов полезную обратную связь за приемлемое время.

В вашей стратегии тестирования возможны упущения. Рассмотрите четыре квадранта тестирования и задайтесь вопросом, все ли они учтены. Регулярно ли вы проводите исследовательское тестирование? Когда в последний раз вы приглашали пользователя на предмет тестирования удобства работы? Как насчет тестирования производительности и нагрузочного тестирования? Методика ATDD относится только к бизнес-ориентированному квадранту, где собраны виды тестирования, помогающие команде продвигаться вперед. Приемочные тесты – важнейшая часть, которой большинство команд уделяют пристальное внимание, но в угоду им не следует жертвовать содержимым остальных квадрантов.

ГЛАВА 10. Разрабатывайте спецификацию совместно

Для традиционной разработки спецификаций программного обеспечения характерна одна проблема: чтобы результат получился качественным, нужно потратить время и приложить специальные усилия. Однако после того как спецификация зафиксирована, она быстро устаревает, в том числе по причинам, находящимся вне вашего контроля. Например, если конкурент выпустил новую версию программы с какой-то дополнительной функцией, то для сохранения своей доли рынка вы должны немедленно пересмотреть требования. Поскольку это решение заведомо не техническое, в его принятии должен хотя бы участвовать человек, ориентирующийся в бизнес-требованиях.

Чем больше разных мнений учитывается в процессе выработки требований к приложению, тем яснее получается общая картина. Вообще говоря, существует по меньшей мере три разных взгляда. С одной стороны, это взгляд на проблему с точки зрения бизнеса. В гибких командах его носителем обычно является представитель заказчика, или – в случае применения методики Scrum – владелец продукта. С другой стороны, есть техническая сторона. В более традиционно организованных командах эту точку зрения может выражать технический руководитель или ведущий программист. В гибких командах имеет смысл включить хотя бы одного сотрудника, который ориентируется в исходном коде.

Наконец, необходим посредник между этими двумя точками зрения. В традиционных проектах эту роль играет бизнес-аналитик. Но с тем же успехом ее можно поручить и опытному тестировщику.

Сила трех[1]

Но почему наличие трех разных точек зрения помогает разрабатывать спецификации программы? При проектировании программной системы приходится принимать много решений, имеющих две стороны: бизнес-функции и технические ограничения. В коде любой программы в изобилии встречаются компромиссы между тем и другим. С другой стороны, в базах данных об ошибках можно найти множество сообщений о последствиях откровенно неверных решений.

Какие-то решения изменить в процессе разработки трудно, а какие-то – легко. Столкнув разные точки зрения – со стороны бизнес-функций и технических ограничений – на ранней стадии, мы сумеем вовремя отыскать правильные компромиссы и, значит, уменьшить количество как трудных, так и простых для исправления ошибок. И, быть может, вовсе избежать ошибок, находящихся между этими двумя крайностями.

Функции программной системы располагаются в пространстве решений [GW89], где существует много вариантов проектирования, способных удовлетворить требования. Каждый вариант характеризуется своим способом исследования пространства решений, отдающим предпочтение различным атрибутам функций, например производительности или поведению под нагрузкой. Атрибуты ограничивают множество возможных решений теми, которые приемлемы для заказчика.

С другой стороны, технологии, используемой для реализации программы, также присущи ограничения. Они сужают множество вариантов проектирования, удовлетворяющих требованиям к функциональности и дополнительным атрибутам.

Приняв в рассмотрение как бизнес-функции с атрибутами, так и технические ограничения на ранней стадии, участники смогут более осмысленно исследовать пространство возможных проектов. Это одна из главных составных частей разработки по приемочным тестам, благодаря которой методика и работает. И тестировщики могут поспособствовать процессу исследования, поскольку обладают навыками критического анализа функций, атрибутов и ограничений ПО.

Но какую независимую точку зрения может выложить на стол отдельная команда тестирования? За десятилетия преподавания тести-

1 The Power of Three (Сила трех) – название четвертого эпизода седьмого сезона сериала «Доктор Кто». – *Прим. перев.*

рования мы поняли сами и втолковывали другим, что программисты и тестировщики должны работать абсолютно независимо, отталкиваясь от единого документа с изложением требований. Это позволяет избежать феномена, который в современной психологии называется *предвзятостью подтверждения*. Тестировщик не должен быть связан точкой зрения программиста и должен критиковать продукт, когда это оправдано. В некоторых командах это правило доводится до абсурда – программистам и тестировщикам вообще запрещается разговаривать друг с другом.

Но совместное с программистами специфицирование программы еще не делает тестировщика необъективным. Просто программисты и тестировщики вместе с представителями бизнеса занимаются тралением требований к программе. Вы считаете, что в итоге тестировщик оказывается предвзятым? Так, может, и простое чтение одного и того же документа с требованиями ведет к тому же результату? Участие в спецификации примеров вместе с программистами и специалистами в предметной области – то же самое, что присутствие на рабочей встрече по выработке требований – мечта любого тестировщика на протяжении всей его профессиональной карьеры.

На самом деле, команды, у которых Гойко Аджич брал интервью для своей книги «Specification by Example» [Adz11], отмечали, что учет мнения хотя бы одного программиста и хотя бы одного тестировщика с самого начала дает лучшее понимание требований. В примере задачи о парковках мы видели, что в ходе обсуждения возникали ситуации, когда технические вопросы, заданные программистами, проясняли бизнес-процесс, а участие тестировщика помогло лучше понять требования и наглядно представить их в виде таблицы.

В ходе итераций программисты и тестировщики отталкиваются от одного и того же базового понимания функциональности. Примеры, положенные в основу приемочных тестов, позволяют команде выверять направление движения.

Для достижения оптимальных результатов следует включить три роли: специалист в предметной области, например владелец продукта или представитель заказчика, программист и тестировщик. Поэтому Джанет Грегори (Janet Gregory) и Лайза Криспин назвали этот подход «силой трех».

В статье «The Rule of Three Interpretations»[2] [Wei93, стр. 90] Вайнберг напоминает о необходимости рассматривать по меньшей мере три варианта перед принятием окончательного решения.

2 Правило трех интерпретаций. – *Прим. перев.*

Если я не могу предложить по крайней мере три разных интерпретации полученных данных, значит, я недостаточно хорошо обдумал, что они могут означать.

«Сила трех» позволяет компенсировать недостаток воображения. Наличие трех разных точек зрения многократно увеличивает потенциал обдумывания требований к функциональности или пользовательской истории. Именно этот факт лежит в основе эффективности принципа «сила трех».

Применение этого принципа не ограничивается функциональными требованиями. Программист понимает проблемы, связанные с производительностью кода, и может попросить уточнить этот аспект. Тестировщик обычно знает о проблемах, касающихся удобства работы. Программист, тестировщик и специалист в предметной области могут внести ясность в качественные атрибуты предполагаемой функциональности – обычно это называется нефункциональными требованиями. Не забывайте рассмотреть упущения, не покрываемые автоматизированными тестами, и собрать полный оркестр (см. раздел «Сбор оркестра для тестирования» в главе 9), ориентируясь на квадранты тестирования (рис. 9.2). Большую часть требований к производительности, поведению под нагрузкой и устойчивости также можно проверить с помощью автоматизированных тестов.

Организуйте рабочие встречи

В части I мы видели, что на рабочей встрече команда может выработать общее понимание функциональности, которую предстоит реализовать. Но чтобы такая встреча принесла пользу любому сотруднику независимо от того, участвовал он в ней или нет, необходимо иметь в виду некоторые моменты.

Состав участников

Прежде всего, необходимо правильно определить состав участников. Некоторые команды считают необходимым присутствие всей команды разработки вместе с представителями заказчиков из разных компаний. В таких случаях совещание становится более многолюдным, но проводится на каждой второй итерации.

С другой стороны, есть команды, которые делегируют на встречу только по одному представителю от каждой из трех ролей. В любом случае должны быть представлены различные точки зрения. Обычно это означает, что должен присутствовать кто-то, знакомый с бизнес-

стороной продукта. Кроме того, хорошо бы привлечь еще и опытного пользователя [Coc06].

В разделе «Сила трех» я отметил, что следует включать по меньшей мере одного программиста и одного тестировщика. Программист знаком с кодом. В ходе обсуждения спецификации программист сможет задавать вопросы о технической стороне реализации требуемой функциональности. В силу самой природы своей работы программистам свойственно задумываться о том, как можно реализовать некоторую функцию, и держать в уме модель предметной области. Рабочая встреча дает им возможность достичь общего с другими участниками понимания предметной области, которой принадлежит новая функциональность.

Тестировщик, со своим критическим складом ума, может быстро найти пробелы в бизнес-правилах. Кроме того, он обладает уникальной способностью наглядно представлять бизнес-правила в виде таблиц. Вне зависимости от того, применял ли он ранее какой-нибудь из каркасов автоматизации, «дружелюбных к гибкой разработке», опытный тестировщик знает о таблицах решений и о том, как выразить сложные бизнес-правила в табличной форме. Если на совместной рабочей встрече будет выработано такое представление, то программисты, тестировщики и специалисты в предметной области смогут достичь согласия по поводу обсуждаемой функциональности еще до написания какого-либо кода.

Цель рабочей встречи

Чтобы рабочая встреча по выработке спецификаций принесла успех, важно понимать, кто составляет целевую аудиторию. Целевой аудиторией в данном случае является команда разработки – программисты и тестировщики, тогда как специалист в предметной области несет с собой необходимые знания о предполагаемой функциональности. Не забывайте – это очень существенно, – что у заказчиков и внешних специалистов время ограничено [Adz09]. Если вы пригласите их на встречу, где программисты ожесточенно спорят, нужно ли использовать последнюю веб-серверную технологию, и какую именно, то, эта встреча, скорее всего, станет последней.

Чтобы не увлечься обсуждением технических деталей реализации, необходим опытный председатель, который не даст уклониться от рассмотрения функциональности. Цель рабочей встречи – выработать единое понимание бизнес-правил. Обсудить технические аспекты можно будет позже, когда представители бизнеса уйдут. Всё

отведенное на рабочую встречу время должно быть потрачено на составление общей картины.

Частота и продолжительность

В связи с этим встает вопрос о том, как часто проводить рабочие встречи. Конечно, это существенно зависит от доступности специалистов в предметной области. Представители бизнеса обычно довольно сильно заняты, но, с другой стороны, иногда в офисе разработчика постоянно присутствует представитель заказчика, которому можно задавать дополнительные вопросы хоть каждый день.

В зависимости от обстоятельств можно планировать рабочие встречи раз в месяц или незадолго до начала очередной итерации. От частоты встреч со специалистом в предметной области зависит и продолжительность совещания. Если встречи происходят редко, то на каждой приходится обсуждать больше функций и, значит, отводить на нее больше времени. Если же встречи случаются нечасто и тем не менее должны быть короткими, то нужно либо обсуждать меньше функций, либо проводить предварительную подготовку.

Если вы собираетесь обсудить меньше функций, то сверьтесь с очередью запланированных работ, учитывая последние приоритеты. Сначала следует рассматривать те функции, которые с наибольшей вероятностью будут включены в следующую итерацию. Возможно, вы захотите опустить функции, по поводу которых у команды нет вопросов, например, вход в систему или поле комментария на вебстранице. Тогда останется больше времени для обсуждения нетривиальных бизнес-правил в более сложных функциях.

Если вы захотите проделать какую-то подготовительную работу, то опять-таки загляните в очередь работ и заранее подготовьте вопросы специалисту. Список открытых вопросов можно составить на планерке по чистке очереди работ (backlog grooming meeting), которую большинство Scrum-команд все равно проводят. Можно также запланировать отдельное совещание по краткому обсуждению очередных функций за несколько дней до рабочей встречи. Еще один способ составить список идей и вопросов по бизнес-правилам – провести предварительное совещание, на котором наглядно изобразить текущее понимание вещей и представить открытые вопросы в виде диаграммы связей. Но имейте в виду, что если вы включите на рабочей встрече проектор для демонстрации диаграмм связей и заставите всех молча смотреть на свои слайды со всеми опечатками, то успех встречи окажется под угрозой. Лучше использовать что-нибудь попроще,

например каталожные карточки, меловую или магнитно-маркерную доску, и назначить ведущего, который будет записывать полученные ответы.

Траление требований

Требования обычно не собираются [Coh04]. Заказчики и специалисты в предметной области полагают, что точно знают, чего хотят. Но по предъявлении конечного результаты оказывается, что они имели в виду нечто совершенно иное.

В этом случае метафора сбора требований неприменима. Требования можно было бы собрать, если бы они лежали у всех на виду, дожидаясь, когда кто-нибудь подберет их и создаст желаемую программу. Увы, требования не валяются на каждом углу. И команда, как правило, усваивает этот урок на собственном горьком опыте.

Майк Кон (Mike Cohn) предлагает другую метафору. Поскольку требования не лежат повсюду, дожидаясь сборщика, мы должны предпринять осознанные усилия для их идентификации. Существуют способы сделать это на разных уровнях, и тут можно провести аналогию с рыболовными сетями с разным размером ячейки, в которых одна рыба запутывается, а другая свободно проплывает. Эта аналогия отлично подходит для описания различных методов специфицирования через примеры.

В рассмотренных выше примерах мы видели два широко распространенных способа такого специфицирования. В первом случае организуется отдельная группа тестирования, которая параллельно с программистами работает над определенной функциональностью программы. При этом наша сеть предназначена для достижения общего понимания функций до того, как они начнут включаться в существующий продукт.

Команда собралась для траления требований к парковке. Поскольку все более-менее понимали, как устроены парковки для автомобилей, для выработки общего понимания принципов расчета стоимости члены команды просто задавали вопросы. Цель этих вопросов – выловить примеры. Тестировщик Тони быстро нашел способ наглядно проиллюстрировать понимание бизнес-правил с помощью записанных примеров. Глядя на них, Филлис и Билл могли понять, чего не хватает в рыболовной сети.

Впоследствии команда свела количество вопросов к минимуму. В ходе обсуждения примеров все поняли, что на самом деле необхо-

димо. Уменьшение числа вопросов помогло увеличить размер ячейки сетки и выпустить ненужную рыбу.

В результате получилась сеть, которая пропускает неинтересные требования, вроде макрели, оставляя только барракуд. С другой стороны, сеть вылавливает также карпов и щук, если они вас интересуют.

Во втором примере мы видели, что установление параметров в ходе диалога – не единственный способ траления требования. В примере со светофорами мы искали дополнительную информацию о предметной области еще до начала какой-либо реализации. Поскольку, кроме нас, над кодом никто не работал, мы смогли применить другой подход к тралению требований. Мы воспользовались методом, который я сознательно назвал «своевременным тралением». По мере роста объема кода и исчерпания примеров мы могли возвращаться к осмыслению требований.

Такой подход работает не только при разработке кода командой из двух человек, но и когда команда более многочисленна. Даже при наличии первоначального представления о предметной области, я считаю, что, выражая свое понимание в виде конкретных примеров, я углубляю его. Более того, в процессе обдумывания я начинаю видеть различные варианты проектирования модели предметной области и ее реализации в коде.

С ростом объема кода изменяется и мое понимание предметной области. Я больше узнаю о приложении и его предполагаемом использовании. На самом деле, мы отложили проектирование модели предметной области в коде, чтобы принять более обоснованное решение об этой модели.

Можно было бы возразить, что дизайн просматривался в самом начале обсуждения. Опытный проектировщик мог бы предвидеть, в каком направлении будет развиваться код. В какой-то степени это правда. В тот момент, когда у меня сложилось впечатление, что мы забрели куда-то не туда, мы остановились и критически проанализировали написанный код. Этот анализ позволил выбрать правильный размер ячейки для траления требований. Как впоследствии показал дизайн контроллера на перекрестке, может обнаружиться, что вместо карпов мы выловили медузу. Но единственный способ чему-то научиться на опыте – отступить на шаг и извлечь урок.

Осмысление сделанного также помогает выбрать правильный размер ячейки сети для траления требований. Большинство успешных команд начинали с простого [Adz11]. По прошествии нескольких

месяцев с начала реализации первого варианта, команда отмечала какой-то изъян в процессе. Она осмысливала этот изъян и вносила коррективы в соответствии с конкретной ситуацией.

Чтобы найти правильный размер ячейки, нужно начать с чего-то, накопить опыт, а затем осмыслить его, извлечь уроки и внести коррективы. Если ваша команда проводит ретроспективный анализ регулярно, то с процессом извлечения уроков и адаптации вы, наверное, знакомы.

Резюме

Наличие трех разных взглядов на требования поможет отобрать правильные. Действуя совместно, заказчики, программисты и тестировщики могут выработать удачные решения. Организуя на рабочих встречах траление требований, вы сможете учесть точки зрения представителей всех ролей. Кроме того, постарайтесь испытать текущую сеть и извлечь уроки из полученного опыта.

ГЛАВА 11. Автоматизируйте буквально

Упорно поработав над тралением требований и составлением примеров для приложения, было бы глупо выбросить их, не воспользовавшись в своих интересах, – или еще того хуже – просто включить в документ, который будет без дела пылиться на полке. Гораздо разумнее использовать примеры как руководство к автоматизации тестов. Если вы сможете задействовать примеры в ходе демонстрации одной итерации или продукта в целом, то представитель со стороны бизнеса увидит, что ранее согласованные примеры были учтены в реализации.

Достичь этой цели можно с помощью дружелюбного подхода к автоматизации, позволяющего автоматизировать примеры настолько буквально, насколько это возможно. Это означает, что примеры можно представить в некотором табличном формате, но текстовую их часть сохранить практически в том виде, в котором она была сформулирована.

В задаче о парковках мы видели такую буквальную автоматизацию. Тони переформулировал примеры, которые были согласованы с Биллом и Филлис в ходе рабочей встречи. И с помощью дружелюбной автоматизации ему удалось автоматизировать их чуть ли не с точностью до буквы.

Можно было бы возразить, что Тони с самого начала записал примеры в формате, пригодном для автоматизации. Действительно, он, возможно, априори имел в виду имеющиеся средства автоматизации тестирования, ориентированные на гибкую разработку. С другой стороны, выбор табличного представления мог быть продиктован традиционными таблицами решений – в предположении, что тестировщик с ними знаком.

Еще один достойный внимания аспект – сотрудничество между программистами и тестировщиками в вопросе автоматизации ут-

вержденных примеров. В задаче о парковках Тони работал над автоматизацией в паре с Алексом. Во второй задаче мы для достижения того же результата попеременно занимались то созданием тестов, то размышлениями о дизайне.

Наконец, буквальная автоматизация преследует цель отразить в тестах предметную область. Фактически во второй задаче мы видели, что система управления светофорами была сначала представлена в виде тестов, а затем эти тесты легли в основу кода модели предметной области.

Где-то в 2010 году я впервые услышал термин *язык предметно-ориентированных тестов*. Первой моей реакцией на него была мысль о том, что между предметно-ориентированными языками, используемыми в коде и тестах, не должно быть вообще никаких расхождений. Иногда – например, в случае применения ATDD к унаследованному коду – мы вынужденно разделяем их. Но со временем два языка должны объединиться в единый общеупотребительный язык проекта. Если они и дальше будут разделены, то возникнет и будет углубляться недопонимание между заинтересованными сторонами. Один из способов добиться объединения – управлять написанием предметного кода непосредственно по примерам.

Рассмотрим более пристально эти три аспекта: техника автоматизации тестов, дружелюбная к гибкой разработке и ATDD, сотрудничество между программистами и тестировщиками и, наконец, раскрытие предметной области и выведение необходимой и достаточной ее модели из примеров.

Используйте дружелюбную автоматизацию

Впервые я услышал выражение *автоматизация тестирования, дружелюбная к ATDD*, на семинаре по ATDD, который проводила Элизабет Хендриксон в 2009 году. Он подразумевает существование инструментов, поддерживающих ATDD и буквальную автоматизацию.

Сразу оговорюсь: хотя здесь мы обсуждаем инструментальные средства, сама методика ATDD прямого отношения к инструментам не имеет и может применяться с разными инструментами. Если вы внедряете какой-то инструмент, то итогом будет внедрение самого инструмента, но не работоспособной методики. Инструмент – это еще не стратегия тестирования [KBP01].

И все же дайте команде возможность принять решение о выборе инструмента. Тут нужно учитывать различные факторы. Может играть роль как выбор языка программирования, так и забота о том, чтобы тесты были понятны владельцу продукта или представителю заказчика. В зависимости от того, в какой мере к работе с инструментом будут привлечены различные члены команды – программисты, тестировщики и специалисты в предметной области, – предпочтение может быть отдано тому или иному аспекту.

На момент написания этой книги существовали разнообразные инструменты автоматизации тестирования, по-разному поддерживающие ATDD. И ключевые слова, и тесты, управляемые данными, и табличные структуры позволяют начать работу с ATDD. Одни ценят возможность записывать тесты на естественном языке, чтобы примеры можно было читать как документацию. Другие предпочитают табличные структуры или выражение тестов в виде определенным образом структурированных текстовых файлов. Но со временем особенности одного инструмента проникают в другие, поэтому можно взять любой каркас и получить при этом многие возможности, имеющиеся в альтернативных каркасах.

Отчасти это заслуга сообщества разработчиков ПО с открытым исходным кодом. В условиях, когда исходный код основных каркасов приемочного тестирования, доступен, новые функции и исправления ошибок появляются быстро. Обнаружив в каком-то инструменте ошибку, вы можете отправить запрос в список рассылки, организованный поставщиком, или обратиться в службу поддержки. Как правило, полезный ответ поступает в течение одного дня.

В примерах выше мы видели два таких инструмента. Cucumber – это написанный на Ruby каркас, где предпочтение отдается написанию тестов в стиле BDD. Он перенесен на Java, Groovy, Scala, Clojure и .NET, так что с его помощью можно тестировать код на любом из этих языков. Дополнительные сведения о Cucumber см. в приложении A и на сайте The Secret Ninja Cucumber Scrolls [dFAdF10].

Во втором примере мы работали с каркасом приемочного тестирования FitNesse, написанным на Java. Он поддерживает две разные системы тестирования: FIT вместе с FitLibrary и SLiM. Каркас перенесен также на.NET, Ruby, Python и PHP. Поскольку в основе FitNesse лежит вики, то он удобен и для распределенных команд. Дополнительные сведения о FitNesse см. в приложении B.

Из других каркасов упомяну Robot Framework, Concordion, JBehave и такие коммерческие продукты, как Twist и Green Pepper. В бли-

жайшие несколько лет возможно появление новых каркасов, которые исправят некоторые недостатки существующих инструментов и преобразят современный ландшафт в этой области.

У всех этих инструментов есть одна общая черта: простота написания тестов, выражающих требования. Cucumber предпочитает стиль Given-When-Then, FitNesse – табличную структуру, а Robot Framework – ключевые слова. Примеры мы видели выше.

А дружелюбность этих инструментов к ATDD предстает в виде возможности автоматизировать примеры, представленные в определенном формате, путем написания некоторого кода. В шагах Cucumber для решения о том, какой код исполнять, применяется сопоставление с регулярными выражениями. В FitNesse в сочетании со SLiM вызываются функции с именами, определяемыми соглашением. Тем самым обеспечивается разделение обязанностей между тестируемым приложением и текстовым представлением тестов.

При использовании ATDD можно вывести подлежащие тестированию примеры, не написав еще ни единой строки кода. Поскольку инструменты обеспечивают полное отделение тестовых данных от приложения, функциональность приложения можно наращивать параллельно составлению примеров или даже управлять реализацией на основе примеров, как было показано в задаче о светофорах.

Такое разделение обязанностей между тестовыми данными и тестируемым приложением и является основной характеристикой автоматизации, дружелюбной к ATDD и гибким методикам разработки. Инструменты, не обладающим этим свойством, будут препятствовать успешному внедрению ATDD. Отделение тестовых данных – необходимое, хотя и не достаточное условие успешного внедрения ATDD.

Сотрудничайте в осуществлении автоматизации

Автоматизация тестов – это разработка ПО. Эксперты твердят об этом вот уже несколько десятилетий. Но самое интересное, что команды по-прежнему забывают об этом и, когда доходит дело до автоматизации тестов, замирают в полной растерянности, что в конечном счете сводит на нет все усилия. У меня есть что сказать на эту тему, почему я и посвятил автоматизации тестирования отдельную главу 12.

В этом разделе мы обратим внимание на вопрос сотрудничества при автоматизации тестов. В задаче о парковках мы видели, что Тони

работал в паре с Алексом, экспертом по автоматизации тестирования. Сначала Тони продвигался самостоятельно, пока не зашел в тупик, после чего обратился за помощью к Алексу.

Практически на любом читаемом мной курсе по ATDD поднимается вопрос о том, должен ли тестировщик уметь программировать. Многие тестировщики вышли из программистов, потому что для чистого программирования им не хватило увлеченности. Иногда отсюда делают вывод, что они программируют плохо[1]. Если поставить их перед лицом ATDD, то они могут так напугаться, что будут всеми силами сопротивляться внедрению этой методики.

Но тестировщикам не надо бояться ATDD. Упор на автоматизацию тестов еще не означает, что тестировщик не может сотрудничать с программистом. На самом деле, большинство программистов, с которыми мне доводилось работать в паре над автоматизацией тестов, очень и очень интересовались моими навыками в области тестирования. С другой стороны, я и сам узнал много нового о реализации и проектировании программ. Работа в паре – ситуация, в которой и программист, и тестировщик выигрывают.

Если внимательнее присмотреться к каркасам автоматизации тестирования, дружелюбным к ATDD, то выяснится, что они достигают отделения тестов от тестируемого приложения за счет включения промежуточного слоя кода. В большинстве команд, приступающих к внедрению ATDD, тестировшик сопровождает тестовые данные, а программист отвечает за тестируемую систему. Но кто должен писать промежуточный код, связывающий оба аспекта?

Если весь код, транслирующий тестовые примеры в вызовы тестируемого приложения, пишет и сопровождает тестировщик, то он может пропустить какие-то внесенные в систему изменения, о которых программист забыл сообщить. Когда-то я работал в компании, где был выделенный отдел тестирования. Работавшие в нем тестировщики несли всю ответственность за автоматизацию тестов. Из-за отсутствия обмена информацией значительную часть проделанной работы по автоматизации приходилось переделывать.

С другой стороны, если программист, который сопровождает часть приложения, также пишет связующий код для подключения тестов, то при изменении приложения и связующего кода, тесты могут «сломаться», если в них не внесены соответствующие модификации.

В обоих сценариях программист и тестировщик должны взаимодействовать. Это самый низкий уровень сотрудничества в автома-

1 Некоторые тестировщики сами так отзываются о себе. Однако в большинстве случаев я с этим не согласен.

тизации тестирования. Я бы сказал, что команда вовсе не является таковой, если ее члены постоянно не взаимодействуют друг с другом. Помочь здесь может информативное рабочее пространство с «информационными радиаторами» (Information Radiators) [Coc06]. Например, если ощущается недостаток сотрудничества между программистами и тестировщиками, можно вывесить на доске задач матрицу образования пар.

Если вы хотите увеличить отдачу от усилий, затрачиваемых на автоматизацию тестов, то программисты и тестировщики должны сотрудничать теснее. Лучше всего, если они работают над связующим кодом бок о бок, программируя в паре. При такой организации труда количество переходящей информации настолько велико, что опытные пары говорят о более чем двукратном повышении продуктивности. Я испытал это на собственном опыте, хотя точных цифр привести не могу.

Когда носители обеих точек зрения на процесс разработки собираются за одним столом, достигается и еще одна цель – обмен опытом между специалистами. Программист узнает о тестировании и о том, как тестировщик выводит новые тестовые сценарии из имеющейся информации. А тестировщик понемногу изучает программирование, разработку через тестирование и может помочь программисту в написании пригодного для сопровождения кода, который и сам понимает.

От сотрудничества выигрывают обе стороны. Первоначальные опасения по поводу нехватки специалистов в команде за счет сотрудничества могут постепенно рассеяться. Сотрудничество в автоматизации тестирования позволяет команде реализовать то, что Элизабет Хендриксон называет коллективным владением тестами. В этом случае вся команда обладает квалификацией, необходимой для написания новых тестов и кода автоматизации. Это помогает подменять друг друга, если разработка отстает от графика. В качестве побочного продукта вы заодно получаете автоматизированные тесты и общее понимание тестового кода и самих тестов.

Сотрудничество может проявляться в разных формах. Как минимум, тестировщики и программисты должны держать друг друга в курсе по поводу связующего кода. Но в идеале они совместно разрабатывают критически важные части кода автоматизации. Если сотрудничество не налажено вовсе, то в лучшем случае вы получаете непреодолимый барьер между программистами и тестировщиками, а в худшем – непонимание приложения в целом. Это может свести

на нет все усилия по формированию команды. Лично я старался бы держаться подальше от такого разлаженного коллектива.

Изучайте предметную область

В задаче об управлении светофорами мы видели, как с самого начала управлять разработкой предметного кода, отправляясь от тестов. При этом мы прояснили некоторые аспекты самой предметной области. Например, контроллер и валидатор допустимых комбинаций состояний светофоров изначально не были частью проекта. Эти концепции появились в результате написания кода тестирования светофоров.

Оставив в стороне достижения индустрии в создании предметно-ориентированных языков и методик разработки на основе поведения, надо признать существование разрыва между предметной областью, для которой пишется приложение, и моделью предметной области, реализованной в коде. Модель состоит из классов и конструкций, которых в реальной предметной области нет.

В шведской армии бытует такой афоризм [GW89]:

Если карта не соответствует местности, доверяй местности.

Если считать, что модель предметной области – это карта самой предметной области, то в случае, когда модель не соответствует предметной области, мы должны придерживаться концепций, существующих в предметной области, а не в модели.

С другой стороны, как выразился Майкл Болтон:

Если вы заблудились, то сгодится даже старая карта.

Это означает, что если вы запутались в модели предметной области, то ничего страшного в этом нет при условии, что есть возможность сориентироваться на местности. Следует работать как над гибкостью тестов, так и над гибкостью кода. Если вы способы адаптировать код и тесты к новому пониманию предметной области, то все будет нормально.

Тесты и примеры могут направить в нужную сторону попытки вывести модель предметной области из самой предметной области. Они помогают перевести концепции реального мира на язык модели. Кроме того, благодаря тестам можно выявить элементы, отсутствующие в модели и в коде. Примерами могут служить валидатор и контроллер – в реальном мире они не существуют, но помогают моделировать предметную область в программном коде.

В задаче о светофорах мы видели, что эта техника дает наибольший эффект, если программисты и тестировщики вместе работают над предметным кодом и кодом автоматизации. Не во всех командах дело поставлено именно так – и тогда управление проектированием и кодированием модели снаружи внутрь не срабатывает. Большинство команд, приступающих к внедрению ATDD, поначалу пренебрегают этим преимуществом. Поэтому я считаю такую организацию работы продвинутой техникой.

Необходимым условием для этого является тесное сотрудничество между программистами и тестировщиками. В идеале программисты должны знать и поддерживать все тестовые примеры наряду с тестировщиками. В таком случае, оказавшись в тупике, они смогут разрабатывать предметный код снаружи внутрь. Но вообще-то я считаю, что оптимально было бы разрабатывать код именно так на протяжении всего проекта.

Другое необходимое условие для разработки снаружи внутрь – наличие некоторого опыта. Опыт нарабатывается со временем путем применения методики к разным ситуациям и осмысления полученных уроков. Поначалу вы, возможно, не будете понимать, как управлять кодированием снаружи внутрь, но надо заставлять себя. Как только увидите возможность применить этот подход, не упускайте ее.

Если вы готовы пересмотреть свою позицию по поводу разработки снаружи внутрь, спросите себя, есть ли еще какие-нибудь возможности, кроме очевидных. Вспомните сформулированное Джерри Вайнбергом правило трех интерпретаций [Wei93]. Если вы видите менее трех разных подходов к управлению разработкой кода, значит, вы недостаточно размышляли над проблемой – или не задали себе фундаментальных вопросов: «В чем заключается проблема? У кого она возникла? Кто должен ее исправить?»

Не считайте, что нет никакой возможности управлять кодом и моделью предметной области снаружи внутрь, пока не найдете по крайней мере три разных подхода к управлению. «По крайней мере» означает, что можно без особого труда найти и пять таких подходов. Затем выберите самый многообещающий, не забывая о желании управлять разработкой снаружи внутрь. Если вы рассмотрели три разных возможности написать код, но так и не нашли способа организовать управление снаружи внутрь, остановитесь на самом многообещающем способе. Со временем вы приобретете больше опыта в разработке снаружи внутрь и будете видеть больше возможностей. Помните, что код не высечен в камне, и современные интегрированные среды разработ-

ки (IDE) поддерживают многочисленные способы последующей реструктуризации и переписывания кода.

Резюме

При автоматизации примеров нужно обращать внимание на несколько моментов. Во-первых, выберите подходящий инструмент. Пусть решение о его выборе принимает вся команда, поскольку не исключено, что в какой-то момент каждому придется работать с ним. Во-вторых, устраните все препятствия, мешающие получить доступ к этому инструменту, и убедитесь, что инструмент обслуживает процесс, а не наоборот.

После того как все препятствия сняты, ничто не должно мешать сотрудничеству в автоматизации. Тестировщики могут черпать знания о проектировании у программистов, а программисты – узнавать о приемах тестирования от тестировщиков. У такого сотрудничества есть и побочный эффект – знания оседают в головах членов команды и помогают достичь общего понимания задачи, решаемой по заказу.

И в этом разработка снаружи внутрь тоже может помочь. Если предметный код приложения отражает общее понимание, то, возможно, его сможет прочесть и понять даже заказчик. Если с помощью приемочных тестов вы раскрываете для себя предметную область, то можете управлять развитием приложения, проекта и архитектуры в самых разных направлениях. Это дает возможность создавать приложения, пригодные для долгосрочного сопровождения.

ГЛАВА 12. Тестируйте рационально

Работающие тесты – самый ценный актив в любом проекте разработки ПО. Они не только доказывают, что система по-прежнему полезна, но и документируют ее функции – и убедиться в этом можно одним нажатием кнопки, если тесты автоматизированы.

Поскольку работающие тесты представляют такую ценность, вы должны стремиться к тому, чтобы они любой ценой сохраняли работоспособность. Или обдуманно и сознательно принимать решение об их удалении. При наличии современной системы управления версиями избавляться от ставших ненужными файлов не страшно. На самом деле, это даже приносит какое-то облегчение. Многие команды признают, что уменьшение числа тестов не только повысило гибкость работы (см. [Adz11]), но и разгрузило мозги ([Hun08]).

Можно возразить, что той же цели можно добиться за счет хорошей организации тестовых примеров. Но сколько времени вы собираетесь хранить все свои тесты? Несколько лет назад я участвовал в проекте для заказчика из Бразилии, в котором тестировщики автоматизировали вообще все тесты, которые приходили им в голову. Налоговая система в Бразилии разрешает каждому из 27 штатов применять собственные налоги. В рассматриваемой системе программисты должны были сконфигурировать данные для каждого штата.

Первый набор автоматизированных тестов включал около 300 примеров для каждого из 27 вариантов. Их полный прогон занимал более двух суток. Несмотря на все усилия, вложенные в сами тесты и код автоматизации, к моменту, когда я присоединился к проекту, всё оказалось бессмысленным, так как при внесении изменений в программу тесты не давали полезной обратной связи в разумное время. Поэтому бизнес-решения принимались без оглядки на результаты тестов, которые появлялись только спустя два дня. Все затраты на автоматизацию тестирования пошли насмарку.

И как же мы поступили, чтобы восстановить контроль над автоматизацией? Мы были вынуждены выбросить некоторые тесты. Мы проанализировали подход к генерации каждого продукта и выявили сопутствующие ему риски. Далее мы рассматривали только их, а не все риски, характерные для любого подхода. В результате удалось избавиться от большинства тестовых примеров. Но вообще-то для уменьшения числа тестов я бы применил метод попарного тестирования (см. раздел «Попарное тестирование» в главе 9 и [HK11]).

Глубинная проблема заключалась в том, что тесты никто не сопровождал. Они не были рациональными, а содержали кучу мусора, который тяжким грузом ложился на плечи группы тестировщиков. Заинтересованные стороны принимали решения, не принимая во внимание результаты тестов, а это отзывалось ошибками, которые обременяли уже программистов, так как им приходилось переписывать значительные части кода.

Всего этого можно было бы избежать, если бы команда не позволяла тестам становиться «разбитыми окнами». Правило разбитого окна [HT99] основано на наблюдении за нежилыми домами. Какое-то время после выселения последнего обитателя с ними ничего не происходит. Но стоит появиться одному разбитому и не замененному окну, как их становится всё больше и больше. То же самое происходит с продуктовым и с тестовым кодом.

Что касается продуктового кода, то этой теме посвящено несколько отличных книг, например, «Clean Code» [Mar08a] Роберта Мартина[1] или «Refactoring to Patterns» [Ker04] Джошуа Кериевски (Joshua Kerievsky). А если вас интересует тестовый код, то можете начать с книги «xUnit Test Patterns» [Mes07] Джерарда Межароса (Gerard Meszaros). Некоторые упомянутые в этих книгах положения вполне применимы и к приемочным тестам.

С годами я вынес три основных урока из прочитанной литературы на эту тему. Во-первых, автоматизация тестов – разновидность разработки ПО. Поэтому к автоматизации тестов следует применять все надлежащие приемы разработки ПО и, в частности, тестировать сам код автоматизации. Во-вторых, в какой-то момент тесты начинают испускать дурной аромат. Его следует вовремя учуять и рассмотреть возможности усовершенствовать не только саму программу, но и процесс разработки, который привел к появлению аромата. И наконец, во время написания этой книги не было инструментов рефакторинга,

1 Р. Мартин «Чистый код. Создание, анализ и рефакторинг», Питер, 2011. – *Прим. перев.*

которые изменяли бы не только код автоматизации тестов, но и сами тесты – и этот факт достоин всяческого сожаления. Ниже я расскажу о некоторых паттернах, которые встречались мне в различных каркасах и, наверное, могли бы быть автоматизированы.

Разрабатывайте код автоматизации тестов постепенно

Одно из основных преимуществ инкрементного и итеративного подхода к разработке ПО состоит в том, что в условиях всё увеличивающегося масштаба систем становится очень трудно заранее предвидеть все требования и функции. Раньше программы разрабатывались, исходя из предположения, что все требования к очередной версии можно собрать еще до начала кодирования. Только вот почему-то оказывалось, что пока писался код, требования бизнеса поменялись, и систему приходится на ходу латать.

Итеративная и инкрементная разработка подразумевает постепенное наращивание функциональности. Идея в том, чтобы начать с самой простой реализации, которую только можно представить. На следующем шаге она уточняется. Например, можно начать с «ходячего скелета» (см. [Coc06]), а потом нарастить на кости мясо. Поначалу можно опустить код проверки, отвергающий недопустимые входные значения. Успех приходит, когда сначала создается наиболее ценный срез программной системы, а позже рассматриваются особые случаи.

Инкрементный и итеративный подход к автоматизации тестирования применим также при разработке кода, связывающего текстовое представление примеров с тестируемой системой. В задаче о парковках мы видели, как Тони и Алекс сначала добились успешного прогона первого теста, а потом автоматизировали один пример за другим. Впоследствии Тони инкрементно автоматизировал тесты, отталкиваясь от этой основы и добавляя остальные парковки.

Ключ к такому подходу заключается в том, чтобы создать дизайн, который можно было бы гибко расширять в желаемом направлении. После того как была заложена возможность задавать различное время начала и окончания стоянки, выбор того или иного вида парковки уже не представлял никаких сложностей. Принятые проектные решения влияют на гибкость кода автоматизации тестов. В задаче о парковках решение оказалось простым. Алекс и Тони легко могли предвидеть,

что понадобится в будущем, и потому выработали гибкий подход к проектированию.

В реальных проектах зачастую всё оказывается не так красиво, как в этом простом примере. Даже в задаче о светофорах ситуация оказалось сложнее, чем можно было предвидеть. Поэтому в более сложных предметных областях имеет смысл сначала сосредоточиться на основной ветке системы, а затем продвигаться от этого плацдарма в разных направлениях.

Как-то я работал в одной компании над заменой существующей системы автоматизации тестирования. В новой системе мы должны были автоматизировать целый ряд бизнес-сценариев. Можно было бы для начала потратить много времени на анализ, составить план нового проекта и реализовать всё, что было известно на тот момент. Но мы поступили по-другому. Сначала мы проанализировали все сценарии на предмет вероятности ошибок и влияния на бизнес. Затем каждому сценарию приписали важность и начали с самых важных для бизнеса сценариев, поскольку это быстро давало наибольшую отдачу. Через восемнадцать недель переход был завершен. Разобравшись сначала с самыми крупными проблемами, мы затем легко смогли заменить тесты для менее важных с точки зрения бизнеса сценариев. На самом деле, в тех сценариях, которые были автоматизированы первыми, уже учитывались многие риски, что позволило использовать их в качестве основы для последующих сценариев.

Отсюда я вынес важный урок – не помещать всю логику в один большой класс связующего кода. Например, FIT позволяет поместить код преобразования бизнес-объектов, например денежных сумм (число с двойной точностью вместе с символом валюты) непосредственно в класс Fixture. Можно также поместить прямо в класс крупные семантические операции, скажем, добавление учетной записи вместе с адресом. Но этот подход столь же плох, как помещение бизнес-логики в классы пользовательского интерфейса.

Прежде всего, написанный таким образом код трудно автоматически тестировать с помощью автономных тестов. В объектной модели вашего тестового кода нет таких типичных объектов предметной области, как денежные суммы или AccountCreator. Рано или поздно члены команды перестанут понимать, как работать с созданным вами кодом, и примутся заново изобретать велосипед. В худшем случае дело кончится тем, что значительные куски вспомогательного кода будут копироваться из одного места в другое. В этот момент вы уже поставили все усилия по автоматизации тестирования на грань кра-

ха, потому что автоматизировать новые функции будет всё труднее и труднее.

Строя вспомогательный код из компонентов, вы создаете независимые классы-помощники, которые в будущем можно будет исключить или развить. В задаче об управлении светофорами мы таким образом выделили компонент CrossingController. В конечном итоге контроллер светофоров на перекрестке стал концепцией модели предметной области, но разрабатывался он как независимый компонент.

Ниже мы увидим, что у подхода к построению вспомогательного кода автоматизации из небольших компонентов есть и еще одно достоинство. Независимые компоненты проще тестировать автономно. Поскольку автоматизация тестирования – это разработка ПО, следует подвергать сложную логику в самом вспомогательном коде автономному тестированию. В главе 9 я упоминал об одной команде, которая добилась значительного (на 10–15 %) увеличения покрытия всей системы тестами за счет применения к коду автоматизации методики разработки через тестирование. Если вы ловите себя на добавлении одного предложения `if` или `switch` за другим, подумайте о том, чтобы сделать структуру кода более объектно-ориентированной, например применить паттерн «Стратегия» [GHJV94] или независимый компонент.

Прислушивайтесь к тестам

Автономные тесты могут многому научить в плане проектирования классов [FP09]. Если автономный тест трудно написать – а, значит, и прочитать, – то не исключено, что в проекте отсутствует какая-то важная концепция.

По собственному опыту скажу, что это относится не только к автономным, но и к приемочным тестам. Если требуется большой объем вспомогательного кода, особенно отягощенного кучей вложенных `if`, значит, либо в самом вспомогательном коде, либо в приложении пропущена какая-то концепция. Если критерии приемки были основаны на бизнес-примерах, то, вероятнее всего, концепция предметной области отсутствует именно в приложении.

Такого рода пример мы видели в задаче об управлении светофорами. В ней тесты и вспомогательный код стали основанием для включения в код концепций предметной области. Это один из способов прислушиваться к тестам и использовать их для управления реализацией.

Но в некоторых командах тестировщикам не предоставляется доступ к продуктовому коду и модели предметной области. В таком случае сотрудничество между тестировщиками и программистами по ходу работы становится насущной необходимостью. Если тестировщик видит, что тесты становятся длинными и трудночитаемыми, но ничего не может с этим поделать, то база кода начнет быстро деградировать.

С другой стороны, если тестировщик может поставить программистов в известность о своих сомнениях на основании того, что происходит с тестами, то программисты, возможно, увидят проблему в коде и вместе с тестировщиками сумеют найти нестандартное решение.

В идеале следовало бы строить весь код по принципу снаружи внутрь. Тогда вероятность получить длинные или нечитаемые тесты уменьшится. Я чаще оказывался в трудном положении, когда задним числом добавлял приемочные тесты к уже существующему коду. Одна из основных проблем при этом – неудобство или даже невозможность подключить тесты, потому что не были предусмотрены необходимые для сквозных тестов точки входа. Продвигаясь снаружи внутрь, вы по определению создаете все нужные точки подключения.

Применяя ATDD, можно получить эффект от прислушивания к тестам на трех разных уровнях. Во-первых, можно прислушиваться к примерам. Если они получаются длинными и отражают всю многошаговую логику систему, значит, отсутствует какая-то абстракция между бизнес-сценарием и деталями технической реализации.

Рассмотрим пример для задачи о парковках, представленный в листинге 12.1. Этот тест написан с помощью ключевых слов библиотеки Selenium в каркасе Robot Framework. Тестируется случай, когда машина оставлена на экономичной парковке на один день, 23 часа и 11 минут. В этом примере есть несколько недостатков. Самый главный – трудности сопровождения в перспективе. Тест не выражает намерения автора. Он слишком многословен. Его необходимо подвергнуть рефакторингу.

Листинг 12.1. Многословный пример для калькулятора стоимости парковки

```
1  Basic Test
2  Open Browser http://www.shino.de/parkcalc/ firefox
3  Set Selenium Speed 0
4  Title Should Be Parking Calculator
5  Select From List Lot Economy Parking
```

```
6  Input Text EntryTime 01:23
7  Select Radio Button EntryTimeAMPM AM
8  Input Text EntryDate 02/28/2000
9  Input Text ExitTime 12:34
10 Select Radio Button ExitTimeAMPM AM
11 Input Text ExitDate 03/01/2000
12 Click Button Submit
13 Page Should Contain (1 Days, 23 Hours, 11 Minutes)
14 [Teardown] Close Browser
```

Сравните с примерами для экономичной парковки, которые составили Тони, Филлис и Билл (листинг 12.2).

Листинг 12.2. Автоматизированные примеры для экономичной парковки (воспроизведение листинга 3.4)

```
1  Feature: функциональность экономичной парковки
2    Калькулятор стоимости парковки вычисляет стоимость стоянки на
     экономичной парковке.
3
4  Scenario Outline: Calculate Economy Parking Cost
5  When I park my car in the Economy Parking Lot for <parking
     duration>
6  Then I will have to pay <parking costs>
7
8  Examples:
9  | parking duration | parking costs |
10 | 30 minutes       | $ 2.00        |
11 | 1 hour           | $ 2.00        |
12 | 4 hours          | $ 8.00        |
13 | 5 hours          | $ 9.00        |
14 | 6 hours          | $ 9.00        |
15 | 24 hours         | $ 9.00        |
16 | 1 day, 1 hour    | $ 11.00       |
17 | 1 day, 3 hours   | $ 15.00       |
18 | 1 day, 5 hours   | $ 18.00       |
19 | 6 days           | $ 54.00       |
20 | 6 days, 1 hour   | $ 54.00       |
21 | 7 days           | $ 54.00       |
22 | 1 week, 2 days   | $ 72.00       |
23 | 3 weeks          | $ 162.00      |
```

Один из способов справиться этой проблемой – создать слой абстракции по принципу ключевых слов. Тони сделал это автоматически, когда добавил примеры, основанные на бизнес-требованиях. Но, возможно, необходимость в таком слое выявится позже. Тогда нужно будет добавить дополнительные сценарии, как мы сделали в задаче о светофорах при рассмотрении недопустимых комбинаций состояний в контроллере первого светофора на перекрестке. Там была добавле-

на абстракция недопустимого состояния. Оглядываясь назад, можно сказать, что это решение было продиктовано наличием в таблице слишком большого числа повторяющихся строк для недопустимых состояний, приводивших к переходу в режим мигающего желтого. Это демонстрация прислушивания к тестам в случае, когда примеры становятся всё более длинными и избыточными.

Другой способ решения проблемы длинных тестов – ввести концепции предметной области во вспомогательный или предметный код. В задаче о светофорах мы применили этот подход, когда ввели в предметный код концепцию состояния светофора, выделенную из приемочных тестов. Позже, при написании приемочных тестов для перекрестка, выявилась необходимость в валидаторе состояний.

Когда-то я работал над проектом, где необходимо было обрабатывать разные типы учетных записей. У каждой учетной записи могли быть дочерние учетные записи, сопровождающиеся различными подписками. В первой попытке автоматизации мы попытались описать всю иерархию. Но автоматизировав таким образом большинство бизнес-сценариев, мы поняли, что сопровождать тесты будет сложно.

Кроме того, сами тесты подсказали нам, что мы работаем не на том уровне абстракции. Мы устроили совещание вместе с представителями бизнеса, на котором выработали концепцию различных структур тарифов и выразили ее в виде тестовых примеров. Вместо иерархии с одной записью верхнего уровня и одной подписной учетной записью на продукт XYZ мы назвали всю ветвь иерархии подпиской в рамках тарифа XYZ, скрыв за этим термином подразумеваемые продажи различных тарифов подписчику. Пример такой таблицы см. в листинге 12.3.

Листинг 12.3. Таблица для подготовки трех разных иерархий учетных записей с разными тарифами и скрытыми за ними продуктами

```
1 !|Account Creator                      |
2 |account name    |tariff               |
3 |Peter Prepaid   |Prepaid Basic 50     |
4 |Paul Postpaid   |Postpaid Business 100|
5 |Claus Convergent|Convergent Family 150|
```

В этом проекте тестировщики не имели доступа к исходному коду, поэтому мы поместили ответственность за продажу продуктов во вспомогательный код к своим тестам. Мы создали справочную таблицу принятых в бизнесе названий тарифов и применили несколько операций к стандартной учетной записи для каждого тарифа. Закончив, мы обнаружили, что второстепенная сложность «переехала» из

тестовых примеров во вспомогательный код, так что сопровождать тесты стало гораздо легче.

Второй способ прислушиваться к тестам заключается в критическом анализе связующего или вспомогательного кода. В задаче о светофорах именно так была выявлена необходимость перечисления состояний светофора. Связующий код подсказал, что иначе получается много предложений `if-then-else`, которые лишь привносят ненужную сложность.

Именно связующий код подал идею о том, чтобы завести концепцию более высокого уровня. В тот момент примеры было просто читать, а их объем был невелик. Но о коде нельзя было сказать того же. Так что на мысль о концепции `LightState` нас навело прислушивание к связующему коду, а не к самим примерам.

Иногда выясняется, что вспомогательный код чрезмерно усложнился. Это явный признак того, что не хватает какой-то концепции. Остановитесь, осмыслите код и подумайте, что это за концепция. Если ничего не приходит в голову, дайте проблеме отстояться. Если увидите в коде следы какой-то концепции, попытайтесь выделить ее.

Наконец, есть еще и третий способ слушать тесты. Он применяется, если код автоматизации разрабатывается через тестирование. Быть может, становится трудно писать автономные тесты. В этой ситуации обратитесь к рекомендациям из книги Стива Фримэна (Steve Freeman) и Ната Прайса (Nat Pryce) «Growing Object-oriented Software Guided by Tests» [FP09]. Скажу лишь, что прислушиваться к автономным тестам следует так же, как к приемочным. Если хотите копнуть глубже, настоятельно советую почитать книгу Фримэна и Прайса.

Подвергайте тесты рефакторингу

На момент написания этой книги средства реструктуризации существующих тестов еще только зарождались. Надеюсь, что в ближайшем будущем мы станем свидетелями прогресса в этой области. Уже есть кое-какие инструменты, закрывающие этот пробел, но все, что я видел, – это лишь дополнения к другим программам.

В программировании под рефакторингом понимается изменение внутренней структуры кода без изменения его функциональности [FBB+99]. Поначалу рассматривались лишь такие мелкие шажки, как переименование метода или выделение переменной. Со временем на

основе такого низкоуровневого рефакторинга стали появляться более сложные способы, например выделение суперкласса.

Всего несколько лет назад в интегрированных средах разработки (IDE) не было никаких средств рефакторинга. Тогда считалось, что это отнимающая много времени операция, которая может «поломать» всю базу кода. Для языка Smalltalk автоматизированные инструменты рефакторинга существовали, для Java или C++ – нет. Автоматизированный рефакторинг безопаснее ручного выполнения отдельных шагов, поскольку производится лишь в том случае, когда можно гарантировать сохранение функциональности.

Теперь, напротив, трудно встретить IDE, не поддерживающую автоматический рефакторинг. Такие средства, как переименование класса или выделение метода из фрагмента кода, просты, безопасны и активно применяются программистами.

К сожалению, инструментам автоматизации тестирования, дружелюбным к гибкой разработке, недостает поддержки рефакторинга тестов. Конечно, примитивную контекстную замену, скажем, изменение цвета red на blue, можно реализовать и с помощью скрипта оболочки, в котором используется, например, потоковый редактор `sed` и регулярные выражения. Но чтобы произвести более сложную модификацию, например поменять местами два столбца в таблице или добавить в таблицу новый столбец, приходится действовать вручную, что достаточно утомительно.

Я бы хотел, чтобы в современные каркасы приемочного тестирования был включен инструмент типа ReFit Йоханнеса Линка (Johannes Link)[2]. Он позволяет производить поиск и замену, а также до определенного предела изменять структуру существующих примеров. Интегрированная среда, поддерживающая разработку приемочных тестов, возможно, позволила бы преодолеть этот недостаток, но пока мой опыт использования таких сред в интересах тестировщиков показывает, что заложенная в них функциональность намного превышает необходимую.

Большинство инструментов, подаваемых как суперпрограммы для автоматизации тестирования, обладают следующими характеристиками:

- включают некий набор стандартных функций, которые кому-то когда-то понадобились, например, запросы к базе данных;
- включают средства графического представления примеров;

2 http://johanneslink.net/projects/refit.jsp

- лицензионная политика заставляет компанию покупать лицензии, обеспечивающие только интересы тестировщиков, но не программистов.

Первое свойство предоставляет в распоряжение тестировщиков богатый набор функций. К несчастью, у тестировщика может возникнуть искушение использовать эти низкоуровневые функции прямо в тестах. И тогда тесты оказываются тесно связанными с реализацией приложения, хотя по идее должны абстрагироваться от него и фокусироваться на бизнес-целях.

Второе свойство усложняет рефакторинг и модификацию тестов. В конечном итоге придется перерисовать всю очаровательную графику, которая делает работу в IDE стол удобной. Но объем работы, необходимой для изменения хоть чего-нибудь, может подорвать любой проект.

А последнее свойство ухудшает сотрудничество в команде. Поскольку у программистов нет доступа к IDE, они не прогоняют функциональные тесты перед записью изменений в систему управления версиями. Разумеется, если при этом что-то сломается, время реакции окажется довольно большим. Если вы найдете внесенную ранее ошибку в момент, когда программист уже перешел к следующей задаче, то он с трудом вспомнит, что же тогда делал.

С точки зрения рефакторинга тестов, второй аргумент делает использование графических инструментов нецелесообразным. Они, может, и предоставляют возможность работать, не зная ничего о программировании. Но одновременно это означает, что программисты не будут адаптировать интерфейсы к тестам, применяя средства автоматического рефакторинга в своих IDE. И еще вам придется делать много работы вручную, когда понадобится что-то модифицировать. Хотя в наши дни программисты по собственной инициативе вводят какие-то уровни абстракции, чтобы разорвать связь между изменениями в приложении и в тестовых примерах, гибкость, необходимая для включения в тесты дополнительного набора данных, по-прежнему зависит от использования других инструментов, например электронных таблиц.

Возвращаясь к инструментам автоматизации тестирования, дружелюбным к гибкой разработке, следует отметить, что ни в одном из них сейчас нет простых средств для реструктурирования примеров. Я вижу большой потенциал у такого средства, потому что оно облегчило бы тестировщикам решение весьма трудоемких задач, возника-

ющих, когда появление новой функции заставляет вносить массивные изменения в ранее написанные тесты.

А пока такого средства нет, я хотел бы описать два вида рефакторинга, которые нахожу очень удобными. Своими названиями они обязаны аналогичным видам рефакторинга исходного кода, и я полагаю, что в будущем смогут лечь в основу более сложных преобразований.

Выделение переменной

На первом месте в этом списке стоит выделение переменной. Переменная – это по существу место для представления повторяющихся значений. Например, в задаче о светофорах можно было бы поместить значение «yellow blink» в переменную с именем «invalid configuration». После этого для замены всех вхождений строки «yellow blink» достаточно было бы поменять содержимое переменной.

Такой вид рефакторинга полезен, если вы предвидите, что в каком-то направлении в будущем возможны изменения. На первый взгляд, изменение недопустимой комбинации в задаче о светофорах может показаться маловероятным, но предположим, что эта система поставляется в другую страну, где конфигурация недопустимых состояний иная. Очевидно, что таким способом вы подготавливаете тест к возможности настройки системы. Если предполагается развитие системы в этом направлении, то имеет смысл предусмотреть это заранее.

Когда требуется выделить переменную из уже существующих тестов, я часто действую в такой последовательности.

1. Определить, какое значение было бы желательно хранить в переменной.
2. Определить переменную и поместить в нее значение. Прогнать тесты, чтобы выявить нежелательные изменения, например, существование переменной с таким же именем.
3. Заменить первое вхождение значения только что определенной переменной. Прогнать тесты и убедиться, что ничего не сломалось.
4. Повторить предыдущий шаг для каждого вхождения значения. После каждого изменения прогонять тесты.

Выделение ключевого слова

Видя, что один и тот же шаг встречается в нескольких тестах, я рассматриваю возможность выделения ключевого слова, чтобы тесты

было проще читать. Такой вид рефакторинга аналогичен выделению метода из фрагмента исходного кода.

В задаче о светофорах мы сделали примерно это, когда ввели концепцию недопустимой комбинации состояний светофоров (в листинге 12.4 повторены ранее приведенные примеры). Мы объединили несколько шагов в одной таблице тестов, чтобы создать новое ключевое слово, описывающее эту типичную ситуацию. Другой пример – вход в систему по имени и паролю. Функция входа может пригодиться во многих тестах.

Листинг 12.4. Выделение ключевого слова для недопустимой комбинации состояний светофоров (воспроизведен листинг 7.15)

```
...
!2 Недопустимые комбинации

!|scenario       |invalid combination|firstLight||secondLight|
|set first light |@firstLight                                 |
|set second light|@secondLight                                |
|execute                                                      |
|check           |first light        |yellow blink            |
|check           |second light       |yellow blink            |

!|script|FirstLightSwitchingCrossingController|

!|invalid combination    |
|firstLight  |secondLight|
|green       |red, yellow|
|green       |green      |
|green       |yellow     |
|yellow      |red, yellow|
|yellow      |green      |
|yellow      |yellow     |
|red, yellow |red, yellow|
|red, yellow |green      |
|red, yellow |yellow     |
|red         |red, yellow|
|red         |green      |
|red         |yellow     |
|yellow blink|red        |
|yellow blink|red, yellow|
|yellow blink|green      |
|yellow blink|yellow     |
```

В отличие от переменных, ключевые слова могут принимать параметры. Выделяя ключевое слово, мы по существу создаем функцию, не возвращающую значение.

Мне доводилось работать с системами, где создавалось несколько уровней ключевых слов, причем ключевые слова более высокого

уровня представляли высокоуровневые концепции, и в их реализации могли использоваться ключевые слова нижних уровней. Но вводить слишком много уровней ключевых слов тоже нехорошо, поскольку в настоящее время не существует простых средств, позволяющих увидеть иерархию ключевых слов целиком, а, значит, их поиск и сопровождение становится трудной задачей.

Ниже описаны шаги, которые я обычно применяю для выделения ключевого слова из существующего примера.

1. Определить, какому сценарию я хотел бы присвоить более удобное имя.
2. Создать ключевое слово, принимающее все необходимые параметры. Прогнав тесты, убедиться в отсутствии нежелательных побочных эффектов, связанных с заведением одноименных ключевых слов.
3. Заменить первое вхождение сценария новым ключевым словом, передав ему все необходимые параметры. Прогнать тесты и убедиться, что ничего не сломалось.
4. Повторить предыдущий шаг для каждого вхождения сценария. После каждого изменения прогонять тесты.

Резюме

Для построения рациональных тестов следует рассматривать автоматизацию тестирования как разработку ПО, а, значит, применять все методики, к которым мы привыкли при написании продуктивного кода. Это относится не только к автономному тестированию кода автоматизации, но и к его рефакторингу. Наконец, при создании связующего кода следует отдавать предпочтение слабо связанным проектам и архитектурам.

Иногда выясняется, что какой-то тест трудно написать или автоматизировать. В таком случае следует прислушаться к тестам и подумать о том, правильно ли конструируется приложение. Используйте информацию, сообщаемую тестами, для устранения проблем в коде.

Когда в предметной области что-то изменяется, вы сталкиваетесь проблемой модификации большого числа тестов. В применении к исходному коду под словом *рефакторинг* понимается изменение структуры кода без изменения поведения программы. Для рационального создания и сопровождения тестов нам необходим инструмент, подде-

рживающий простые виды рефакторинга. Мы познакомились с двумя такими видами: выделение переменной и выделение ключевого слова. Я ожидаю, что в скором будущем появятся и другие.

ГЛАВА 13. Успешное внедрение ATDD

На этом заканчивается наше путешествие в мир разработки через приемочные тесты, которая иначе называется специфицированием через примеры или гибким приемочным тестированием. Приступая к составлению примеров бизнес-сценариев в собственном проекте, старайтесь выразить критерии приемки в одном из рассмотренных форматов. Не тратьте много времени на поиск самого правильного формата. Внедрение новой методики означает, что придется изучать новые вещи, экспериментировать с различными идеями и смотреть, что подходит для вас, а что нет. Не впадайте в ступор от разнообразия имеющихся вариантов. Начните с какого-то конкретного бизнес-сценария и постепенно расширяйте набор приемочных тестов.

Сейчас я исхожу из того, что вы раньше не применяли ATDD и работаете над проектом, который в каком-то виде уже существует. Требуется задним числом написать приемочные тесты для уже существующих частей. Вероятно, вам придется оговорить с представителем заказчика выделение на это дополнительного времени. Временные затраты на создание первого прототипа необходимо ограничить. Как-то нам было поручено создать прототип для приложения на базе SWT. Я не был знаком с драйверами для SWT-приложений, и мы выбрали SWTBot. Полдня мы вдвоем с напарником потратили на изучение способа подключения SWTBot к SWT-приложению. Мы выбрали такой сценарий работы системы, который хотя и был достаточно сложным для первого прототипа, но казался нам вполне реальным.

Мы начали с прямолинейного подхода, поместив практически всё в одну гигантскую тестовую функцию, и так прогнали первый тест. Затем мы приступили к рефакторингу тестового кода, чтобы он стал более модульным и пригодным для долгосрочного сопровождения. Еще полдня в следующем забеге мы потратили на переход от JUnit к

FitNesse. Это был первый шаг на пути создания страховочной сетки приемочных тестов.

Что включить в первый набор приемочных тестов, не имеет особого значения. Ищите самый распространенный бизнес-сценарий. В одной компании мы провели анализ рисков для различных бизнес-сценариев и начали с наиболее рискованных и тех, которые выполнялись чаще всего. Там самым мы заложили прочный фундамент и смогли использовать первый набор тестов и код автоматизации как основу для рассмотрения последующих сценариев.

По мере накопления опыта работы с выбранным подходом вы должны обязательно побеседовать с представителем бизнеса о требованиях. Попытайтесь записать примеры в знакомом вам формате. При общении с представителем бизнеса пользуйтесь таблицами, продемонстрируйте ему конечные результаты по завершении автоматизации примеров.

Не забывайте приглашать на рабочие встречи по выработке спецификаций нужных людей в нужное время. Не нужно приглашать много народу, но все, кто полезен для дела, должны присутствовать. Выбор времени проведения также важен. Если за два дня до конца текущего забега владелец продукта в смысле Scrum еще не принял решения о том, принять или отвергнуть его результаты, то обсуждать, какие требования следует реализовать в следующем забеге, преждевременно – с точки зрения как команды, так и владельца продукта.

Автоматизировав первые несколько примеров, постарайтесь включить их в систему непрерывной интеграции. Автоматизированные тесты выполняют свое предназначение, только если выполняются регулярно, и вам не нужно постоянно думать об этом. Для большинства каркасов имеются подключаемые модули, реализующие интерфейс с наиболее распространенными на сегодня системами непрерывной интеграции. Если такого модуля нет, то, вероятно, существует команда, которая уже сталкивалась с подобной ситуацией. Задайте вопрос на официальном сайте или обратитесь в службу поддержки. Быть может, люди, сопровождающие каркас, тоже знают, как поместить результаты вашей работы в конкретную систему непрерывной интеграции.

По мере увеличения количества тестов позаботьтесь о страховочной сетке для них. Быть может, в будущем вам понадобится изменить организацию тестов, чтобы отразить новое понимание предметной области. В одной компании мы первоначально организовали тесты по бизнес-сценариям. Со временем мы поняли, что изменения как

правило связаны с некими моделями тарифов. Поэтому в основу организации были положены именно тарифные модели, а не сценарии. Какая бы организация ни была выбрана сначала, впоследствии ее можно будет без труда изменить.

Обращайте также внимание на базу кода автоматизации. Не забывайте автономно тестировать код, предназначенный для автоматизации тестов. Это может оказаться непростым делом, если используется много ключевых слов или таблиц сценариев, поскольку обычно они автономному тестированию не поддаются.

Не забывайте про рефакторинг кода. Старайтесь инкапсулировать концепции предметной области в отдельных объектах, даже если не имеете возможности выделять предметные классы из кода автоматизации. В случае веб-страниц это может означать создание объектов, представляющих каждую страницу. В других случаях в роли объектов предметной области могут выступать денежные суммы, учетные записи, пользователи или квитанции. Если создать для каждой такой концепции отдельный класс, то в дальнейшем будет проще развивать код и, значит, избежать кошмара сопровождения кода автоматизации, с которым сталкиваются многие и многие команды.

Возможно, со временем, осознав недостатки первых шагов, вы захотите переписать кое-какие ранние примеры. Не пожалейте усилий на то, чтобы изменить их сейчас, пока проект не стал слишком сложным. Велико искушение просто скопировать пример, изменить какие-то данные и сохранить в репозитории результат. Не идите этим путем. Подумайте о том, как включить новый пример таким образом, чтобы последующие добавлять было проще. Возможно, вам удастся выразить тесты в виде таблицы или отыскать в примерах новый уровень абстракции. Копирование и вставка – вещь простая, но ее следует расценивать как симптом надвигающейся проблемы в коде.

Чтобы начать, выберите какой-нибудь подход, поработайте с ним некоторое время, осмыслите полученный опыт и внесите коррективы. Можно долго рассуждать о том, какой подход лучше. Но гораздо эффективнее просто приступить к работе и решать проблемы по мере поступления. Понаблюдайте, как мелкие, казалось бы, изменения со временем приводят к заметным улучшениям. Я полагаю, что именно в этом секрет успеха внедрения методики – не только ATDD, но и любого процесса разработки ПО.

ПРИЛОЖЕНИЕ А. Cucumber

В этом приложении приводится краткое введение в систему Cucumber. Дополнительные сведения можно найти в книге «The RSpec book» [CAH+10], а также на сайте Secret Ninja Cucumber Scrolls [dFAdF10] или на официальном сайте Cucumber по адресу http://cukes.info. Там рассматриваются, в частности, такие более сложные темы, как инициализация и очистка и организация файлов с описанием функциональности.

Cucumber – это инструмент автоматизации тестирования, опирающийся на методику разработки на основе поведения (BDD) . Это означает, что примеры записываются в формате Given-When-Then (Дано-Если-То).

Cucumber основан на Ruby. Если Ruby уже установлен, то выполните команду `gem install cucumber` с правами администратора и дождитесь завершения установки. Если установщик сообщает об отсутствующих зависимостях, запустите команду `gem` с флагом `-y`. Последние версии `gem` автоматически устанавливают зависимости по умолчанию.

Файлы функционала

Прежде всего, вам понадобится «файл функционала» (feature file), в котором описывается предполагаемая функциональность приложения. Каждый такой файл состоит из строки Feature[1], в которой задается название и описание функционала и одного или нескольких разделов Scenario Outline, в которых описываются сценарии. Каждый сценарий может содержать произвольное количество шагов Given, один шаг When и один или несколько шагов Then.

1 Cucumber поддерживает различные языки, и ключевые слова, такие, как Feature, When и т. п., могут быть переведены. Но поскольку перевод не стандартизован, в тексте приводятся англоязычные варианты. – *Прим. перев.*

Шаг Given необязателен. Его часто опускают, если общая инициализация системы производится отдельно или запуск теста не предполагает каких-то особых предусловий. Шаги When и Then обязательны, если вы вообще хотите что-то протестировать.

В листинге A.1 приведен пример файла функционала.

Листинг A.1. Пример файла функционала в Cucumber

```
1  Feature: функциональность парковки с доставкой в указанное место
2    Калькулятор стоимости парковки вычисляет стоимость стоянки на
     парковке с доставкой в указанное место.
3
4  Scenario Outline: Calculate Valet Parking Cost
5  When I park my car in the Valet Parking Lot for <parking
     duration>
6  Then I will have to pay <parking costs>
7
8  Examples:
9  | parking duration | parking costs |
10 | 30 minutes       | $ 12.00       |
11 | 3 hours          | $ 12.00       |
12 | 5 hours          | $ 12.00       |
13 | 5 hours 1 minute | $ 18.00       |
14 | 12 hours         | $ 18.00       |
15 | 24 hours         | $ 18.00       |
16 | 1 day 1 minute   | $ 36.00       |
17 | 3 days           | $ 54.00       |
18 | 1 week           | $ 126.00      |
```

Определения шагов

Процесс разработки снаружи внутрь предполагает, что далее следует связать с шагами какой-то программный код. В Cucumber этот код называется определением шага и зависит от языка программирования. Если используется Ruby (язык Cucumber по умолчанию), то определение шага представляет собой замыкание, которому соответствует образец для сравнения. Каждый шаг Given начинается ключевым словом Given, за которым следует регулярное выражение, сопоставляемое со строкой файла функционала. Шаги When и Then устроены аналогичны. В листинге A.2 приведен пример определения шагов для показанного выше файла функционала.

Листинг A.2. Определения шагов для файла функционала из листинга A.1

```
1  When /^I park my car in the Valet Parking Lot for (.*)$/ do |
       duration|
```

```
2    $parkcalc.select('Valet Parking')
3    $parkcalc.enter_parking_duration(duration)
4  end
5
6  Then /^I will have to pay (.*)$/ do |price|
7    $parkcalc.parking_costs.should == price
8  end
```

Механизмы подключения кода на других языках программирования в принципе аналогичны, но применяются конструкции конкретного языка, например аннотации Java. Детали синтаксиса для всех поддерживаемых языков описаны в документации.

Продуктовый код

Теперь можно приступать к разработке продуктовой системы. В задаче о парковках мы использовали промежуточный слой, который иногда называют страничными объектами. Он обеспечивает подключение тестов к страницам тестируемого веб-сайта. Такая стратегия больше всего подходит для веб-приложений. Если же речь идет о коде, не относящемся к пользовательскому интерфейсу, то его разработку можно вести непосредственно на основе определений шагов.

ПРИЛОЖЕНИЕ B. FitNesse

В этом приложении приводится краткое введение в систему FitNesse – вики для приемочного тестирования. Дополнительные сведения можно найти на сайте http://fitnesse.org. В книге «Fit for Developing Software» [MC05] также затрагивается FitNesse, хотя материал уже немного устарел. В частности, не рассматриваются таблицы для подсистемы Simple List invocation Method (SLiM), которые Роберт К. Мартин добавил в 2008 году.

На рис. B.1 представлена архитектура FitNesse с адаптерами для двух систем тестирования – FIT и SLiM. Тестовые сценарии оформляются в виде иерархически организованных вики-страниц. FitNesse исполняет тесты с помощью FIT- или SLiM-клиента – какого именно, задается путем определения некоторой переменной. Эту переменную можно переопределить внутри иерархии и тем самым смешивать тесты, написанные для FIT и SLiM. Однако при запуске всего набора тесты для FIT должны исполняться отдельно от тестов для SLiM.

FIT-клиент или исполнитель SLiM, показанные справа от линии «граница процесса», вызывают различные методы, следуя определенным соглашениям. Эти методы обращаются к тестируемой системе, заставляя ее выполнить указанные в тестах действия или сохранить какие-то результаты внутри нее самой.

Более старая система FIT поставляется по лицензии GPLv2. Поскольку для организации связи между тестами и приложением в ней применяется наследование, то получается, что весь код ваших фикстур подпадает под действие GPLv2. Сам я не юрист, но для некоторых компаний такая схема лицензирования неприемлема, поэтому Роберт К. Мартин предложил систему, в которой вместо наследования используются соглашения. Далее я не буду обсуждать FIT, а остановлюсь только на тестировании с помощью SLiM.

Структура вики

Каждая вики-страница имеет имя, записываемое в «верблюжьей нотации». Так, страница для обновления баланса учетной записи может называться AccountReloading. Имя, записанное в верблюжьей нотации, должно содержать не менее двух букв в верхнем регистре и не должно содержать пробелов.

Вики-страницы FitNesse можно группировать в наборы с иерархической структурой. Например, можно включить в один набор взаимосвязанные части системы, которые должны тестироваться вместе. FitNesse позволяет прогонять только тесты, принадлежащие какому-то поднабору, изолированно от остальных. С другой стороны, можно прогнать и все тесты, начиная с самого верхнего уровня, или, наоборот, только тесты в одном каком-то листовом узле.

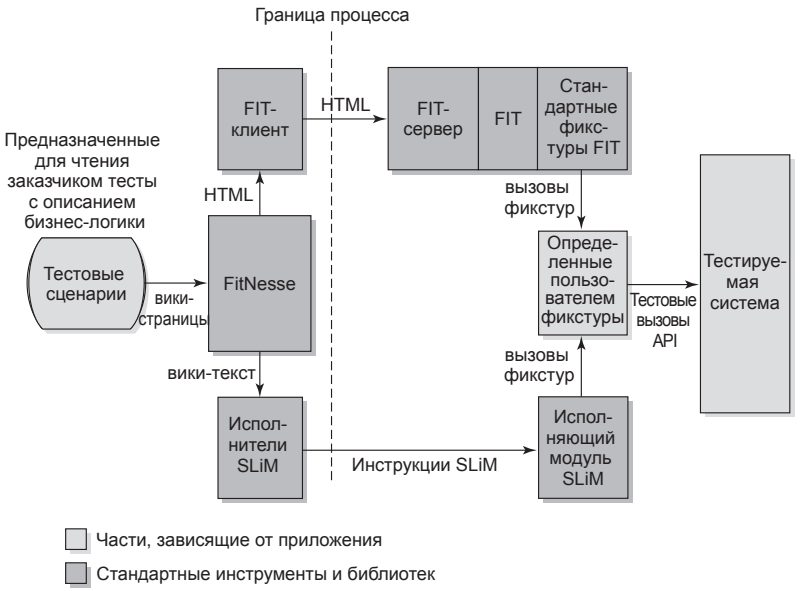


Рис. В.1. Архитектура FitNesse с двумя системами тестирования

Вики-сайт удобен для организации знаний. Поскольку одна вики-страница может ссылаться на другие, FitNesse позволяет включать в тестовые сценарии написанную вами документацию и вообще всю дополнительную информацию, например, типичные последовательности операций в системе.

Таблицы SLiM

Для выражения примеров, описывающих поведение приложения, можно использовать различные таблицы. Самая удобная из них – таблицы решений. В ней записываются входные данные и ожидаемые результаты выполнения некоторой операции. В листинге B.1 приведен пример.

Листинг B.1. Таблица решений SLiM в FitNesse

```
!|Светофоры |
|previous state|next state? |
|red           |red, yellow |
|red, yellow   |green       |
|green         |yellow      |
|yellow        |red         |
|invalid state |yellow blink|
```

Для наборов значений предусмотрены таблицы запросов. Они удобны, когда требуется проверить какие-то наборы данных в системе. Например, можно запросить список всех учетных записей или записей, удовлетворяющих некоторому ограничению. В таблице запросов список объектов записывается по одному в строке. Можно определить несколько подлежащих проверке свойств. Пример приведен в листинге B.2.

Листинг B.2. Таблица запросов SLiM в FitNesse

```
!|Query:активные учетные записи |
|account number|account name    |
|123           |John Doe        |
|42            |Jane Doe        |
|24141         |Luke Skywalker  |
|51214         |Darth Vader     |
```

Таблицы решений и таблицы запросов можно использовать в одном и том же тесте. Обычно это означает, что они включаются в состав какого-то плана выполнения. Именно здесь в игру вступают скрипты. В таблицах решений можно опускать ожидаемые результаты, тогда таблица решений превращается в таблицу настройки параметров для теста. Затем в таблице скрипта исполняется некоторое действие, например операция с ранее созданной учетной записью. По завершении ее результаты можно проверить в самом скрипте или с помощью таблиц запросов.

Такой формат аналогичен формату Given-When-Then, применяемому в реализации методики BDD в Cucumber. С помощью таблицы

настройки описываются все шаги Given, с помощью действий в таблице скрипта – шаг When, а шагу Then соответствуют либо действия в таблице скрипта, либо таблица запросов.

Этот механизм становится еще более эффективным в сочетании с таблицами сценариев, которые позволяют выделять из таблиц скриптов общие шаги, выполняемые несколько раз. С помощью таблиц сценариев организуется слой высокоуровневых абстракций. В каком-то смысле они аналогичны ключевым словам в каркасе Robot Framework (http://robotframework.org) – еще одном инструменте автоматизации тестирования, дружелюбном к гибкой разработке.

Вспомогательный код

Вспомогательный код для таблиц SLiM можно писать на разных языках. «Родным» для FitNesse является язык Java, но SLiM перенесена также на Ruby, Python, .NET, PHP и другие языки. Далее мы будем рассматривать только Java.

Для таблиц решений необходимо реализовать метод установки для каждого столбца с входными данными и метод чтения для каждого столбца с выходными данными. Исполнитель SLiM автоматически найдет подходящие методы чтения и установки по имени столбца.

Для таблиц запросов необходимо вернуть три вложенных списка строк. Самый внутренний список описывает отдельные ячейки, содержащие метку заголовка столбца и данные. Ячейки объединяются в строки в среднем списке. А самый внешний список содержит все списки, соответствующие строкам. Чтобы привыкнуть к этому, мне потребовалось некоторое время. Но после того как один раз разберешься, проблем уже не возникает.

В таблицах скриптов имя вызываемой функции конструируется путем преобразования содержимого строки в верблюжью нотацию. Этой функции передаются параметры, представленные в столбцах с четными номерами. Таблица, показанная в листинге B.3, транслируется в вызов метода `scriptWithSomeParameters(String value1, String value2)` из класса `ScriptTable`.

Листинг B.3. Таблица скрипта SLiM в FitNesse

```
1  !|script|ScriptTable|
2  |script with|param1|some|param2|parameters|
```

Дополнительные сведения можно найти в руководстве пользователя FitNesse и в учебных пособиях Бретта Шухерта (Brett Schuchert), на которые есть ссылки на главной странице сайта FitNesse.

ПРИЛОЖЕНИЕ C. Robot Framework

В этом приложении приводится краткое введение в систему Robot Framework. Прилагаемая к ней документация не может не вызвать восхищения. Если захотите ознакомиться с системой подробнее, обратитесь к руководству пользователя[1]. При освоении Robot лично мне очень пригодилась документация по встроенным библиотекам[2].

Для работы с Robot вам понадобится язык Python. Существует также реализация на Java, которая позволяет писать связующий код на Java вместо Python. Инструкции по установке приведены по адресу http://code.google.com/p/robotframework/wiki/Installation. Сама установка не вызывает никаких сложностей. По завершении установки на вашем диске окажется исполняемый файл `pybot` или `jybot` в зависимости от того, какую реализацию Python вы устанавливали. Для чистого Python в путь `PATH` будет добавлен файл `pybot`, для реализации Python под управлением виртуальной машины Java (Jython) – файл `jybot`. Далее будет рассматриваться только `pybot`. Если вы используется Jython, замените всюду `pybot` на `jybot`.

В каркасе Robot тестовые данные могут записываться в разных форматах: HTML, формат с разделителями-табуляторами и обычный текст. Как правило, я пользуюсь обычным текстом, потому что такая запись удобнее для чтения. Но и для обычного текста существует несколько представлений: с пробелами или вертикальными черточками и пробелами в качестве разделителей, а также в формате reStructuredText. Я буду использовать только формат с пробелами или вертикальными черточками, потому что он больше всего подходит для печатного издания. Сведения о других форматах можно найти в руководстве пользователя.

1 http://code.google.com/p/robotframework/wiki/UserGuide
2 http://code.google.com/p/robotframework/wiki/TestLibraries

Секции

В каркасе Robot тестовые файлы состоят из нескольких секций. Библиотеки ключевых слов (разновидность метаданных), используемые в тестах, описываются в секции Settings. Тестовые сценарии определяются в секции Test Cases, а определенным вами нестандартные ключевые слова – в секции Keywords. Наконец, в секции Variables можно определять переменные.

В разделе Settings можно документировать тестовые сценарии, перечислять ключевые слова для инициализации и завершения как отдельных тестов, так и всего набора, а также загружать ключевые слова из другого текстового файла. В листинге С.1 приведен пример раздела Settings для тестов, которые я написал для калькулятора стоимости парковки. В строке 4 упоминается файл `resource.txt`, играющий ту же роль, что файл `env.rb`, встречавшийся нам в первом примере для Cucumber.

Листинг С.1. Каркас Robot Framework: секция Settings для тестов калькулятора стоимости парковки

```
1  ** Settings ***
2  | Documentation | A test suite with test cases for valet
     parking. |
3
4  | Resource | resource.txt |
5
6  | Suite Setup | Open ParkCalc |
7  | Test Setup | Input Parking Lot | Valet Parking |
8  | Suite Teardown | Close Browser |
9 ...
```

Тестовые сценарии в Robot Framework чаще всего записываются с помощью ключевых слов разного происхождения. Тесты для некоторых примеров, описывающих парковку с доставкой машины в указанное клиентом место, приведены в листинге С.2. Здесь мы видим содержимое секции Test Cases. Имейте в виду, что способ инициализации для каждого теста выбирается из раскрывающегося списка (см. строку 7 в листинге С.1). Поэтому этот шаг явно не указан в тестовых сценариях.

Листинг С.2. Каркас Robot Framework: секция Test Cases для тестов калькулятора стоимости парковки

```
1  ...
2  *** Test Cases ***
3
```

```
4  Less Than Five Hours
5  | | Input Entry Date | 05/04/2010 | 12:00 | AM |
6  | | Input Leaving Date | 05/04/2010 | 01:00 | AM |
7  | | Calculated Cost Should Be | $ 12.00 |
8
9  Exactly Five Hours
10 | | Input Entry Date | 05/04/2010 | 12:34 | AM |
11 | | Input Leaving Date | 05/04/2010 | 05:34 | AM |
12 | | Calculated Cost Should Be | $ 12.00 |
13
14 More Than Five Hours
15 | | Input Entry Date | 05/04/2010 | 12:00 | AM |
16 | | Input Leaving Date | 05/04/2010 | 12:00 | PM |
17 | | Calculated Cost Should Be | $ 18.00 |
18
19 Multiple Days
20 | | Input Entry Date | 05/04/2010 | 12:00 | AM |
21 | | Input Leaving Date | 05/08/2010 | 12:00 | AM |
22 | | Calculated Cost Should Be | $ 72.00 |
```

В секции Keywords файла resource.txt определены три ключевых слова: `Input Entry Date`, `Input Leaving Date` и `Calculated Cost Should Be`. В их определениях (листинг C.3) используются низкоуровневые ключевые слова для работы с драйвером Selenium и названия полей, выбираемых из формы.

Листинг C.3. Каркас Robot Framework: секция Keywords для тестов калькулятора стоимости парковки

```
1  *** Keywords ***
2
3  | Open ParkCalc |
4  | | Open Browser | ${PARKCALC URL} | ${BROWSER} |
5  | | Set Selenium Speed | ${DELAY} |
6  | | Title Should Be | ${PAGE_TITLE} |
7
8  | Input Parking Lot | [Arguments] | ${lot} |
9  | | Select From List | ParkingLot | ${lot} |
10
11 | Input Entry Date | [Arguments] | ${date} | ${time} | ${ampm} |
12 | | Input Text | StartingDate | ${date} |
13 | | Input Text | StartingTime | ${time} |
14 | | Select Radio Button | StartingTimeAMPM | ${ampm} |
15
16 | Input Leaving Date | [Arguments] | ${date} | ${time} | ${ampm} |
17 | | Input Text | LeavingDate | ${date} |
18 | | Input Text | LeavingTime | ${time} |
19 | | Select Radio Button | LeavingTimeAMPM | ${ampm} |
20
21 | Calculated Cost Should Be | [Arguments] | ${cost} |
```

```
22 | | Click Button | Submit |
23 | | ${actual} = | Get Text | ${COST_ITEM} |
24 | | Log | Actual costs: ${actual} |
25 | | Page Should Contain | ${cost} |
```

В определениях ключевых слов встречаются переменные, например `$COST_ITEM` и `$BROWSER`. Они определяются в отдельной секции (листинг С.4). Каркас Robot Framework позволяет переопределять эти значения по умолчанию при каждом выполнении. Для этого служат флаги командной строки, задаваемые при запуске `pybot`. Детали смотрите в руководстве пользователя; способов несколько, и одни могут показаться вам удобнее, чем другие.

Листинг С.4. Каркас Robot Framework: секция Variables для тестов калькулятора стоимости парковки

```
1  *** Variables ***
2  | ${BROWSER} | firefox |
3  | ${DELAY} | 0 |
4  | ${PARKCALC URL} | http://www.shino.de/parkcalc/ |
5  | ${COST_ITEM} | //tr[td/div[@class='SubHead'] = 'estimated
     Parking costs']/td/span/b |
6  | ${PAGE_TITLE} | Parking Cost Calculator |
```

Поскольку во всех четырех показанных выше тестах используются одни и те же ключевые слова, то тесты можно было бы упростить. Для этого в каркасе Robot Framework предусмотрены шаблоны тестов. Именно так запускаются тесты, управляемые данными. Я выделил ключевое слово Valet Parking, включив в него все три шага, выполняемые после выбора парковки с доставкой. В качестве параметров это ключевое слово принимает дату и время прибытия и убытия и ожидаемую стоимость и сравнивает фактически полученный результат с ожидаемым. В листинге С.5 показано, как выглядят примеры, переработанные под шаблоны тестов. Для представления всех шести значений времени стоянки и присваивания им «говорящих» имен я воспользовался переменными. Сохранить в одной переменной несколько значений позволяют имеющиеся в Robot Framework переменные-массивы.

Листинг С.4. Каркас Robot Framework: использование шаблона ключевого слова для определения тестов, управляемых данными

```
1  *** Test Cases ***
2
3  | Valet Parking Test | [Template] | Valet Parking |
4  | | @{FOR_ONE_HOUR} | $ 12.00 |
5  | | @{FOR_FIVE_HOURS} | $ 12.00 |
```

```
6  | | @{FOR_ONE_DAY} | $ 18.00 |
7  | | @{FOR_THREE_DAYS} | $ 54.00 |
```

Библиотечный код

Помимо использования обширных встроенных библиотек, вы можете самостоятельно создавать код для управления приложением. Работает это примерно так же, как в FitNesse. Вам необходимо определить статическую функцию в написанном на Python коде, которая соответствует имени используемого ключевого слова. Но вместо верблюжьей нотации имя функции записывается с подчерками. Подробные примеры имеются в документации для разработчиков библиотек[3].

3 http://robotframework.googlecode.com/hg/doc/python/PythonTutorial.html

СПИСОК ЛИТЕРАТУРЫ

[Adz09] Adzic, Gojko. Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing. Neuri Limited, 2009. ISBN 9780955683619.

[Adz11] Adzic, Gojko. Specification by Example: How Successful Teams Deliver the Right Software. Manning Publications, 2011. ISBN 9781617290084.

[App11] Appelo, Jurgen. Management 3.0: Leading Agile Developers, Developing Agile Leaders (Addison-Wesley Signature Series (Cohn)). Addison-Wesley, 2011. ISBN 9780321712479.

[Bec02] Beck, Kent. Test-Driven Development: By Example. Addison-Wesley Professional, 2002. ISBN 9780321146533.

[CAH+10] Chelimsky, David, Astels, Dave, Helmkamp, Bryan, North, Dan, Dennis, Zach, and Hellesoy, Aslak. The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends (The Facets of Ruby Series). Pragmatic Bookshelf, 2010. ISBN 9781934356371.

[Car65] Carroll, Lewis. Alice's Adventures in Wonderland and Through the Looking-Glass. Tribeca Books, 1865. ISBN 9781936594061.[1]

[CG09] Crispin, Lisa, and Gregory, Janet. Agile Testing: A Practical Guide for Testers and Agile Teams. Addison-Wesley, 2009. ISBN 9780321534460.

[Coc06] Cockburn, Alistair. Agile Software Development: The Cooperative Game (2nd Edition). Addison-Wesley, 2006. ISBN 9780321482754.

[Coh04] Cohn, Mike. User Stories Applied: For Agile Software Development. Addison-Wesley, 2004. ISBN 9780321205681.

[Cop04] Copeland, Lee. A Practitioner's Guide to Software Test Design. Artech House, 2004. ISBN 9781580537919.

[dFAdF10] de Florinier, David, Adzic, Gojko, and de Florinier, Annette. The Secret Ninja Cucumber Scrolls, 2010. http://cuke4ninja.com

1 Л. Кэрролл «Алиса в стране чудес».

[DMG07] Duvall, Paul M., Matyas, Steve, and Glover, Andrew. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley, 2007. ISBN 9780321336385.

[Eva03] Evans, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2003. ISBN 9780321125217.[2]

[FBB+99] Fowler, Martin, Beck, Kent, Brant, John, Opdyke, William, and Roberts, Don. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999. ISBN 9780201485677.[3]

[FP09] Freeman, Steve, and Pryce, Nat. Growing Object-Oriented Software, Guided by Tests. Addison-Wesley, 2009. ISBN 9780321503626.

[GHJV94] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John M. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. ISBN 9780201633610.

[GW89] Gause, Donald C., and Weinberg, Gerald. Exploring Requirements: Quality Before Design. Dorset House Publishing Co Inc., U.S., 1989. ISBN 9780932633736.

[HK11] Heusser, Matthew, and Kulkarni, Govind, editors. How to Reduce the Cost of Software Testing. Auerbach, 2011. ISBN 9781439861554.

[HT99] Hunt, Andrew, and Thomas, David. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, 1999. ISBN 9780201616224.[4]

[Hun08] Hunt, Andy. Pragmatic Thinking and Learning: Refactor Your Wetware (Pragmatic Programmers). Pragmatic Bookshelf, 2008. ISBN 9781934356050.

[Jon07] Jones, Capers. Estimating Software Costs: Bringing Realism to Estimating. McGraw-Hill Osborne Media, 2nd edition, 2007. ISBN 9780071483001.

[KBP01] Kaner, Cem, Bach, James, and Pettichord, Bret. Lessons Learned in Software Testing. Wiley, 2001. ISBN 9780471081128.

[Ker04] Kerievsky, Joshua. Refactoring to Patterns. Addison-Wesley, 2004. ISBN 9780321213358.

2 Э. Эванс «Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем», Вильямс, 2010.

3 Мартин Фаулер «Рефакторинг. Улучшение существующего кода», Символ-Плюс, 2008.

4 Э. Хант, Д. Томас «Программист-прагматик. Путь от подмастерья к мастеру», Лори, 2009.

[KFN99] Kaner, Cem, Falk, Jack, and Nguyen, Hung Q. Testing Computer Software, 2nd Edition. Wiley, 2nd edition, 1999. ISBN 9780471358466.

[Mar02] Martin, Robert C. Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, 2002. ISBN 9780135974445.

[Mar03] Marick, Brian. My Agile Testing Project, 2003. http://www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1

[Mar08a] Martin, Robert C. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008. ISBN 9780132350884.[5]

[Mar08b] Martin, Robert C. Fitnesse---The Fully Integrated Standalone Wiki and Acceptance Testing Framework, 2008. http://www.fitnesse.org

[MC05] Mugridge, Rick, and Cunningham, Ward. Fit for Developing Software: Framework for Integrated Tests. Prentice Hall, 2005. ISBN 9780321269348.

[Mes07] Meszaros, Gerard. xUnit Test Patterns: Refactoring Test Code. Addison-Wesley, 2007. ISBN 9780131495050.

[Nor06] North, Dan. Behavior Modification. Better Software Magazine, 3 2006.

[Tal10] Taleb, Nassim Nicholas. The Black Swan: Second Edition: The Impact of the Highly Improbable: With a new section: "On Robustness and Fragility." Random House Trade Paperbacks, 2010. ISBN 9780812973815.[6]

[Wei86] Weinberg, Gerald M. The Secrets of Consulting: A Guide to Giving and Getting Advice Successfully. Dorset House Publishing, 1986. ISBN 9780932633019.

[Wei91] Weinberg, Gerald M. Quality Software Management: Volume 1, Systems Thinking. Dorset House, 1991. ISBN 9780932633729.

[Wei93] Weinberg,GeraldM. Quality Software Management: Volume 2, First-Order Measurement. Dorset House, 1993. ISBN 9780932633248

[Wei01] Weinberg, Gerald M. More Secrets of Consulting: The Consultant's Tool Kit. Dorset House, 2001. ISBN 9780932633521.

5 Р. Мартин «Чистый код», Питер, 2011.

6 Талеб Нассим Николас «Черный лебедь. Под знаком непредсказуемости», КоЛибри, 2009.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

К

Н

О

П

P

C

T

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499)782-38-89.

Электронный адрес **books@alians-kniga.ru**.

Маркус Гэртнер

ATDD – разработка программного обеспечения через приемочные тесты

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16 .
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 23,52. Тираж 100 экз.

Веб-сайт издательства: www.дмк.рф