Искусство автономного тестирования

с примерами на C#

Рой Ошероув

Второе издание

ДМК ИЗДАТЕЛЬСТВО

MANNING

Рой Ошероув

Искусство автономного тестирования с примерами на C#

Второе издание

The Art of Unit Testing Second Edition

WITH EXAMPLES IN C#

ROY OSHEROVE

MANNING
SHELTER ISLAND

Искусство автономного тестирования с примерами на C#

Второе издание

РОЙ ОШЕРОУВ

ДМК ПРЕСС
издательство
Москва, 2014

УДК 004.438.NET
ББК 32.973.26-018.2
О96

О96 Ошероув Р.
Искусство автономного тестирования с примерами на C#. 2-е издание / пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2014. – 360 с.: ил.

ISBN 978-5-94074-945-5

Во втором издании книги «Искусство автономного тестирования» автор шаг за шагом проведет вас по пути от первого простенького автономного теста до создания полного комплекта тестов – понятных, удобных для сопровождения и заслуживающих доверия. Вы и не заметите, как перейдете к более сложным вопросам – заглушкам и подставкам – и попутно научитесь работать с изолирующими каркасами типа Moq, FakeItEasy или Typemock Isolator. Вы узнаете о паттернах тестирования и организации тестов, о том, как проводить рефакторинг приложений и тестировать «нетестопригодный» код. Не забыл автор и об интеграционном тестировании и тестировании работы с базами данных.

Примеры в книге написаны на C#, но будут понятны всем, кто владеет каким-нибудь статически типизированным языком, например Java или C++.

УДК 004.438.NET
ББК 32.973.26-018.2

ISBN 978-1-61729-089-3 (англ.)
ISBN 978-5-94074-945-5 (рус.)

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ РОБЕРТА С. МАРТИНА КО ВТОРОМУ ИЗДАНИЮ

Было это году в 2009. Я выступал на конференции норвежских разработчиков в Осло (ах, Осло в июне!). Мероприятие проводилось на огромной спортивной арене. Организаторы разделили трибуны на секции, установили перед каждой секцией помосты и развесили между ними черные полотнища, так что получилось восемь отдельных «залов» для заседаний. Помню, я как раз заканчивал доклад, который был посвящен TDD[1], а, может, SOLID[2] или астрономии или еще чему-то, когда внезапно с соседнего помоста раздалось хриплое пение, сопровождаемое громкими гитарными аккордами.

Драпировки не мешали мне взглянуть, что там происходит, и я увидел на соседней сцене парня, который производил весь этот шум. Разумеется, это был Рой Ошероув.

Те из вас, кто меня знает, в курсе, что я тоже в принципе могу под настроение запеть посредине технического доклада о софте. Поэтому, вернувшись к своей аудитории, я подумал, что мы с этим Ошероувом – родственные души, и надо бы познакомиться поближе.

Так я и поступил. И надо сказать, он внес немалый вклад в мою последнюю книгу «The Clean Coder» и три дня вместе со мной вел семинар по TDD. Мой опыт общения с Роем оказался исключительно приятным, надеюсь, что мы еще не раз встретимся.

Уверен, что и ваш опыт общения с Роем – путем прочтения этой книги – будет не менее приятным, поскольку эта книга – нечто особенное.

---

1 Test-driven development – разработка через тестирование. – *Прим. перев.*

2 Single responsibility, Open-closed, Liskov substitution, Interface segregation и Dependency inversion – принципы объектно-ориентированного проектирования: единственной обязанности, подстановки Лисков, разделения интерфейсов и инверсии зависимости. – *Прим. перев.*

Вы когда-нибудь читали романы Миченера[3]? Я – нет, но говорят, что все они начинаются «с атома». Книга, которую вы держите в руках, – не роман Джеймса Миченера, но тоже начинается с атома – атома автономного тестирования.

Не впадайте в заблуждение, пролистывая первые страницы. Это не *просто* введение в автономное тестирование. Это лишь начало и, если вы – опытный разработчик, то первые главы можете проглядеть по диагонали. Но из последующих глав, опирающихся друг на друга, будет возведена конструкция, поражающая своей глубиной. Честно скажу, когда я читал последнюю главу (еще не зная, что она последняя), то думал, что в следующей речь пойдет о мире во всем мире – ну действительно, о чем еще говорить, после того как решена проблема внедрения автономного тестирования в упрямо сопротивляющихся организациях с унаследованными системами?

Эта книга техническая – очень техническая. В ней уйма кода. И это хорошо. Но Рой не ограничивается одними лишь техническими проблемами. Иногда он достает гитару и разражается песней – рассказывает случаи из своей практики или пускается в философские рассуждения о смысле проектирования или о том, что такое интеграция. Похоже, ему доставляет искреннее удовольствие потчевать нас историями о грубейших ошибках, которые он совершал в далеком 2006 году.

Да, кстати, не переживайте по поводу того, что все примеры написаны на C#. Я хочу сказать, что по существу-то C# ничем не отличается от Java. Правда? Да и вообще это не имеет значения. Для формулирования своих мыслей Рой может использовать C#, но уроки, извлекаемые из этой книги, в равной мере применимы к Java, C, Ruby, Python, PHP и любому другому языку программированию (за исключением разве что COBOL).

Неважно, кто вы: новичок, еще незнакомый с автономным тестированием и методикой разработки через тестирование или, ас, поднаторевший в этом деле, – что-то для себя в этой книге вы обязательно найдете. Так что готовьтесь с удовольствием послушать, как Рой будет исполнять песню «Искусство автономного тестирования».

Да, Рой, а ты, пожалуйста, настрой уже эту гитару!

Роберт С. Мартин (дядюшка Боб)
CLEANCODER.COM

---

[3] Джеймс Элберт Миченер – американский писатель, автор более 40 произведений, в основном исторических саг, описывающих жизнь нескольких поколений в какомлибо определённом географическом месте. Отличался тщательной проработкой деталей в своих произведениях. – *Прим. перев.*

ПРЕДИСЛОВИЕ МАЙКЛА ФЭЗЕРСА КО ВТОРОМУ ИЗДАНИЮ

Когда Рой Ошероув сообщил мне, что работает над книгой об автономном тестировании, я испытал огромную радость. Идея тестирования витала в отрасли уже много лет, но в материалах, посвященных автономному тестированию, все же ощущался недостаток. На моей книжной полке стояли книги о тестировании вообще и о разработке через тестирование в частности, но до сих пор не было исчерпывающего справочника по автономному тестированию – книги, в котором тема раскрывалась бы с азов и до общепринятых практических приемов. И это поистине удивительно. Ведь автономное тестирование – не новая концепция. Как же мы дошли до жизни такой?

Высказывание о том, что наша отрасль еще очень молода, стало уже чуть ли не общим местом. Но это правда. Не прошло еще и ста лет, как математики заложили основы нашей отрасли, а оборудование, достаточно быстрое, чтобы воплотить их идеи в жизнь, создано лишь 60 лет назад. В нашей отрасли изначально существовал разрыв между теорией и практикой, и только сейчас мы начинаем осознавать, как это отразилось на ней.

Когда-то давно машинное время стоило дорого. Мы запускали пакетные программы. Программистам выделялось время для прогона их задач, они набивали программы на перфокартах и относили колоды в машинный зал. Если в программе была ошибка, то вы теряли выделенное вам время, поэтому приходилось, сидя за столом, мысленно на бумаге проигрывать все возможные сценарии, все граничные случаи. Сомневаюсь, что тогда сама идея автоматизированного автономного тестирования кому-то могла прийти в голову. Зачем использовать машину для тестирования, когда можно задействовать ее для решения задач, ради чего она и была построена? Из-за скудости ресурсов мы пребывали во мраке.

Позже, когда машины стали быстрее, мы отравились ядом интерактивных вычислений. Мы могли вводить и изменять код по собственной прихоти. Идея проверки кода за столом ушла в прошлое, и мы растеряли дисциплину прежних лет. Мы знали, что программировать трудно, но это означало лишь, что мы должны проводить больше времени за компьютером, меняя линии и символы, пока не найдем нужное заклинание.

Мы перешли от скудости сразу к изобилию, проскочив промежуточные этапы, но теперь исправляем это упущение. Автоматизированное автономное тестирование сочетает дисциплину проверки за столом с новым представлением о компьютере, как о ресурсе для разработки. Мы можем писать автоматизированные тесты на том же языке, на котором написана сама программа, чтобы контролировать свою работу – не однократно, а так часто, как способны эти тесты прогонять. Не думаю, что в разработке программного обеспечения есть еще какая-нибудь столь же действенная практика.

Сейчас, в 2009 году, когда я пишу эти строки, книга Роя уже передана в производство, и я очень рад этому. Она являет собой практическое руководство, которое поможет вам на первых шагах и будет служить отличным справочником по решению связанных с тестированием задач. «Искусство автономного тестирования» – не книга об идеализированных сценариях. Она научит вас, как тестировать реальный код, как пользоваться популярными каркасами и, самое главное, как писать код, удобный для тестирования. Книга с таким названием должна была бы появиться много лет назад, но тогда мы не были к ней готовы. Теперь готовы. Радуйтесь.

Майкл Фезерс

ВСТУПЛЕНИЕ

В одном из самых крупных провалившихся проектов, над которыми я работал, автономные тесты были. Или мне так казалось. Я возглавлял группу программистов, писавших приложение для выставления счетов, и разрабатывали мы, как положено, через тестирование – писали сначала тест, потом код, наблюдали, как тест не проходит, изменяли код, чтобы тест прошел, производили рефакторинг – и все по новой.

Первые несколько месяцев все шло как по маслу. У нас были тесты, доказывавшие, что код работает. Но со временем требования изменялись. И мы были вынуждены изменять код в соответствии с новыми требованиями, а при этом переставали работать тесты, и их тоже приходилось исправлять. Код все еще работал, но написанные нами тесты стали такими хрупкими, что малейшее изменение в коде приводило к их отказу, хотя сам код работал правильно. Задача модификации кода в классе или методе обернулась сущим кошмаром, так как необходимо было править все сопутствующие автономные тесты.

Хуже того, некоторые тесты стали непригодны, потому что писавшие их люди уволились, и никто не знал, как эти тесты сопровождать и что вообще они проверяют. Имена наших методов автономного тестирования оказались недостаточно понятными, а некоторые тесты зависели от других. В конце концов, не прошло и шести месяцев, как мы выбросили большую часть тестов.

Проект с треском провалился, потому что мы довели дело до того, что тесты приносили больше вреда, чем пользы. На их сопровождение и понимание уходило больше времени, чем экономилось, поэтому мы перестали их использовать. Впоследствии при работе над другими проектами мы уже подходили к автономным тестам более обдуманно, и это принесло свои плоды, позволив сэкономить кучу времени на отладке и интеграции. Но со времен того первого неудачного проекта я собираю наиболее удачные приемы автономного тестирования и применяю их в последующих проектах. Каждый новый проект пополняет эту копилку.

Именно рассказу о том, как писать автономные тесты – и при этом делать их удобными для сопровождения и восприятия, а также достойными доверия, – и посвящена эта книга. Ни язык, ни интегрированная среда разработки (IDE) значения не имеют. В начале мы рассмотрим основы создания автономного теста, затем перейдем к тестированию взаимодействий и к изложению рекомендаций по написанию, управлению и сопровождению автономных тестов в реальных условиях.

БЛАГОДАРНОСТИ

Выражаю огромную благодарность Майклу Стивенсу (Michael Stephens) и Нермине Миллер (Nermina Miller) из издательства Manning, которые терпеливо сопровождали меня на всех этапах долгого пути по написанию этой книги. Спасибо также всем сотрудникам издательства, которые работали над выпуском второго издания, иногда оставаясь невидимыми для меня.

Спасибо Джиму Ньюкирку (Jim Newkirk), Майклу Фэзерсу (Michael Feathers), Джерарду Мезарошу (Gerard Meszaros) и многим другим, у кого я черпал вдохновение и идеи, составившие эту книгу. Отдельное спасибо дядюшке Бобу Мартину за то, что он согласился написать предисловие ко второму изданию.

Хочу также поблагодарить за ценные отзывы рецензентов, которые читали рукопись на различных этапах подготовки: Аарон Колкорд (Aaron Colcord), Алессандро Кампеизм (Alessandro Campeism), Алессандро Галло (Alessandro Gallo), Билл Соренсен (Bill Sorensen), Бруно Соннино (Bruno Sonnino), Камаль Чакар (Camal Cakar), Дэвид Мадурос (David Madouros), д-р Франсиш Буонтемпо (Frances Buontempo), Дрор Хэлпер (Dror Helper), Франческо Гогги (Francesco Goggi), Иван Пазминьо (Iván Pazmiño), Джейсон Хэйлз (Jason Hales), Джуао Ангело (João Angelo), Калеб Педерсон (Kaleb Pederson), Карл Метивир (Karl Metivier), Мартин Скурла (Martin Skurla), Мартин Флэтчер (Martyn Fletcher), Пол Стэк (Paul Stack), Филипп Ли (Philip Lee), Прадип Челлаппан (Pradeep Chellappan), Рафаэль Фариа (Raphael Faria), Тим Слоун (Tim Sloan).

Я признателен также Рикарду Нильсону (Rickard Nilsson) за техническую редактуру окончательного варианта рукописи непосредственно перед сдачей в печать.

И напоследок хочу сказать спасибо читателям ранних вариантов книги, публикуемых по программе предварительного ознакомления издательства Manning, за комментарии в сетевом форуме. Вы помогли книге стать такой, какая она есть.

ОБ ЭТОЙ КНИГЕ

Из всего, что я слышал об обучении, пожалуй, самым умным был совет: если хочешь чему-то по-настоящему научиться, начни это преподавать (не помню, кто это сказал). Работа над первым изданием этой книги, которое вышло в 2009, стала для меня именно таким опытом изучения. Я начал писать книгу, потому что устал снова и снова отвечать на одни и те же вопросы. Но были и другие причины. Я хотел попробовать что-то новое; я хотел поставить эксперимент; мне было интересно, чему я смогу научиться, работая над книгой – любой книгой. Автономное тестирование было предметом, в котором я хорошо разбирался. По крайней мере, я так думал. Беда в том, что чем больше опыта, тем глупее себя ощущаешь.

В первом издании есть места, с которыми я сегодня не согласен – например, что понятие «автономная единица» (unit) относится к методу. Это совершенно неверно. Автономная единица – это единица работы, об этом я говорю в первой главе второго издания. Она может быть совсем мелкой – как метод – или достаточно крупной – несколько классов (а то и сборок). Есть и другие изменения, о которых вы в свое время узнаете.

Что нового во втором издании

Во второе издание я добавил материал об ограниченных и неограниченных изолирующих каркасах, а также новую главу 6 о том, что считать хорошим изолирующим каркасом, и о внутреннем устройстве каркасов типа Typemock.

Я больше не использую RhinoMocks. Держитесь от него подальше. Этот продукт мертвый – по крайней мере, в данный момент. Основы изолирующих каркасов я демонстрирую на примере NSubstitute и рекомендую также FakeItEasy. Я по-прежнему не в восторге от MOQ – по причинам, которые объясняются в главе 6.

Я включил несколько новых приемов в главу о внедрении автономного тестирования на уровне организации.

В код примеров внесено множество проектных изменений. Я почти полностью отказался от установки свойств и использую главным

образом внедрение зависимости через конструктор. Добавлено рассмотрение принципов SOLID, но лишь в объеме, достаточном, чтобы разжечь в вас интерес к самостоятельному изучению этой темы.

В разделах главы 7, относящихся к сборке, тоже есть новая информация. За время, прошедшее с выхода первого издания, я много узнал об автоматизации сборки и соответствующих паттернах.

Я не рекомендую использовать методы подготовки и показываю, как можно реализовать ту же функциональность по-другому. Я также использую последние версии NUnit, поэтому применяемый в книге NUnit API частично изменился.

Изменению подверглись средства, относящиеся к унаследованному коду, описанные в главе 10.

Последние три года я работал не только с .NET, но и с Ruby, и это легло в основу новых соображений о проектировании и тестопригодности, которые я излагаю в главе 10. Все материалы в приложении об инструментах и каркасах актуализированы, сведения об устаревших инструментах исключены.

Предполагаемая аудитория

Эта книга для всех, кто пишет код и хочет узнать о передовой практике автономного тестирования. Все примеры написаны на C# с использованием Visual Studio, поэтому работающим на платформе .NET они будут особенно полезны. Но приведенные рекомендации равным образом относятся к большинству, если не ко всем объектно-ориентированным, статически типизированным языкам (в частности, VB.NET, Java, and C++). Если вы архитектор, разработчик, руководитель группы, инженер по контролю качеству (пишущий код) или начинающий программист, то эта книга для вас.

Структура книги

Если вы никогда не писали автономных тестов, то лучше читать книгу от корки до корки, чтобы составить полную картину. Ну а те, кто уже имеет опыт, могут читать главы выборочно в любом удобном порядке. Книга состоит из четырех частей.

В первой части вы научитесь основам написания автономных тестов: узнаете, как работать с каркасом тестирования (NUnit) и что такое атрибуты автоматизированного тестирования, например `[Test]` и `[TestCase]`. Здесь же рассказывается об утверждениях, игнорировании некоторых тестов, тестировании единицы работы, трех типах

значений, возвращаемых автономным тестом, и трех соответствующих им типах тестов: тесты, основанные на значениях, тесты, основанные на состоянии, и тесты взаимодействия.

Во второй части рассматриваются приемы разрыва зависимостей: подставные объекты, заглушки, изолирующие каркасы и соответствующие им способы рефакторинга кода. В главе 3 вводится понятие о заглушках и показывается, как их вручную создавать и использовать. В главе 4 дается представление о тестировании взаимодействия с помощью написанных вручную подставных объектов. В главе 5 обе идеи сводятся вместе и демонстрируется, как изолирующие каркасы позволяют их объединить и автоматизировать. В главе 6 содержится углубленное обсуждение ограниченных и неограниченных изолирующих каркасов и их внутреннего устройства.

Третья часть посвящена различным способам организации тестового кода, приемам его запуска и переработки структуры, а также передовым методам написания тестов. В главе 7 обсуждаются иерархии тестов, а также вопросы использования API инфраструктуры тестирования и включения тестов в автоматизированную процедуру сборки. В главе 8 даются рекомендации по созданию тестов, которые были бы удобны для чтения и сопровождения и заслуживали доверия.

В четвертой части речь идет о внедрении новой методологии в организации и о работе с уже существующим кодом. В главе 9 обсуждаются проблемы, с которыми приходится сталкиваться при попытке внедрить автономное тестирование в организации, и способы их решения. Здесь же перечисляются вопросы, которые вам могут задать, и предлагаются ответы на них. Глава 10 посвящена автономному тестированию существующего унаследованного кода. Описываются два способа решить, с чего начинать тестирование, и рассматриваются некоторые инструменты тестирования нетестопригодного кода. В главе 11 мы поговорим о весьма важной теме проектирования с учетом тестопригодности и о существующих сегодня вариантах.

В приложении описываются инструменты, которые могут оказаться полезны для тестирования.

Графические выделения и загрузка исходного кода

Исходный код к этой книге можно скачать со страницы https://github.com/royosherove/aout2, с сайта книги по адресу www.ArtOfUnitTesting.com, а также с сайта издательства www.manning.

com/TheArtofUnitTestingSecondEdition. В корневой папке и во всех папках глав имеются файлы с именем Readme.txt; в них описано, как установить и запустить код.

Весь исходный код в листингах и в тексте выделяется `моноширинным шрифтом`. В листингах **полужирным** шрифтом выделяются изменения по сравнению с предыдущим примером, а также части, которые будут изменены в следующем примере. Многие листинги сопровождаются аннотациями, иллюстрирующими основные идеи, и пронумерованными маркерами, на которые даются ссылки в последующих пояснениях.

Требования к программному обеспечению

Для исполнения приведенных в книге программ потребуется как минимум Visual Studio C# Express (распространяется бесплатно) или более полная версия (которая стоит денег). Также понадобится каркас NUnit (бесплатный и с открытым исходным кодом) и другие инструменты, о которых будет сказано в своем месте. Все упоминаемые инструменты либо бесплатны и распространяются с открытым исходным кодом, либо имеют пробную версию, которой можно бесплатно воспользоваться при чтении этой книги.

Автор в сети

Приобретение книги «Искусство автономного тестирования» открывает бесплатный доступ к закрытому форуму, организованному издательством Manning Publications, где вы можете оставить свои комментарии к книге, задать технические вопросы и получить помощь от автора и других пользователей. Получить доступ к форуму и подписаться на список рассылки можно на странице www.manning.com/TheArtofUnitTestingSecondEdition. Там же написано, как зайти на форум после регистрации, на какую помощь можно рассчитывать, и изложены правила поведения в форуме.

Издательство Manning обязуется предоставлять читателям площадку для общения с другими читателями и автором. Однако это не означает, что автор обязан как-то участвовать в обсуждениях; его присутствие на форуме остается чисто добровольным (и не оплачивается). Мы советуем задавать автору какие-нибудь хитроумные вопросы, чтобы его интерес к форуму не угасал!

Форум автора в сети и архивы будут доступны на сайте издательства до тех пор, пока книга не перестанет печататься.

Другие проекты Роя Ошероува

Рой написал также следующие книги:

- «Beautiful Builds: Growing Readable, Maintainable Automated Build Processes». Доступна на сайте http://BeautifulBuilds.com.
- «Notes to a Software Team Leader: Growing Self-Organizing Teams». Доступна на сайте http://TeamLeadSkills.com.

Другие ресурсы:

- Блог для руководителей групп, относящийся к этой книге, находится по адресу http://5whys.com.
- Видеоролик с мастер-классом Роя по TDD находится по адресу http://TddCourse.Osherove.com.
- Многочисленные бесплатные видеоролики по автономному тестированию имеются на сайтах http://ArtOfUnitTesting.com и http://Osherove.com/Videos.
- Рой постоянно проводит курсы и консультации по всему миру. Заказать проведение курса на территории своей компании вы можете на сайте http://contact.osherove.com.
- Адрес Роя в Твиттере @RoyOsherove.

ОБ ИЛЛЮСТРАЦИИ НА ОБЛОЖКЕ

Рисунок на обложке книги называется «Japonais en costume de cérémonie», или японец в церемониальной одежде. Это раскрашенная литография, взятая из книги «Естественная история человека» Джеймса Причарда, напечатанной в Англии в 1847 году. Наш художник нашел ее в антикварном магазине в Сан-Франциско.

Причард начал свое исследование народов мира в 1813 году. 34 года спустя, когда книга была опубликована, у него накопилось множество материалов о различных народах и нациях; его работа заложила основы современной этнологии. В книгу Причарда были включены портреты представителей различных рас и племен в национальных костюмах, написанные разными художниками, в основном с натуры.

Литографии из коллекции Причарда, как и другие иллюстрации на обложках наших книг, воскрешают богатство и разнообразие одежды и племенных обычаев двухсотлетней давности. С тех пор манера одеваться сильно изменилась, и различия между областями, когда-то столь разительные, сгладились. Теперь трудно отличить друг от друга даже выходцев с разных континентов, что уж говорить о странах или областях. Но можно взглянуть на это и с оптимизмом – мы обменяли культурное и визуальное разнообразие на иное устройство личной жизни – основанное на многостороннем и стремительном технологическом и интеллектуальном развитии.

Издательство Manning откликается на новации и инициативы в компьютерной отрасли обложками своих книг, на которых представлено широкое разнообразие местных укладов быта в позапрошлом веке. Мы возвращаем его в том виде, в каком оно запечатлено на рисунках из разных собраний, в том числе из собрания Причарда.

Часть I.

ПРИСТУПАЯ К РАБОТЕ

В этой части книги мы рассмотрим основы автономного тестирования.

В главе 1 я определю, что такое автономная единица и что понимается под «хорошим» автономным тестированием, а затем сравню автономное и интеграционное тестирование. После этого мы познакомимся с понятием разработки через тестирование и ее связью с автономным тестированием.

Первый автономный тест с применением NUnit мы напишем в главе 2. Вы узнаете о базовом API NUnit, о том, что такое утверждение и как прогонять тест в исполнителе тестов NUnit.

ГЛАВА 1. Основы автономного тестирования

В этой главе:

- Определение автономного теста.
- Сравнение автономного и интеграционного тестирования.
- Разбор простого примера автономного тестирования.
- Что такое разработка через тестирование.

В любом деле есть первый шаг: вы впервые пишете программу, впервые проваливаете проект и впервые доводите до успешного завершения то, что собирались. Вы никогда не забудете свой первый раз, и надеюсь, что свои первые тесты тоже не забудете. Возможно, вам уже доводилось писать тесты и, быть может, вы даже помните, какие они были плохие, неуклюжие, медленные или несопровождаемые (обычно люди такое помнят). Но не исключено, что все было наоборот: ваш первый опыт написания автономных тестов был на удивление удачным, а эту книгу вы читаете, чтобы понять, что упустили из виду.

В этой главе мы сначала проанализируем «классическое» определение автономного теста и сравним его с понятием интеграционного тестирования. Различие между ними многих приводит в замешательство. Затем мы рассмотрим плюсы и минусы автономного тестирования в сравнении с интеграционным и предложим более подходящее определение «хорошего» автономного теста. В заключение мы поговорим о том, что такое разработка через тестирование, поскольку эта методика часто ассоциируется с автономным тестированием. В этой главе я также затрону ряд концепций, которые более подробно будут рассмотрены далее.

Начнем с определения того, каким должен быть автономный тест.

1.1. Определение автономного тестирования, шаг за шагом

Концепция автономного тестирования – не новость в индустрии разработки программного обеспечения. Она зародилась еще на заре создания языка программирования Smalltalk в 1970-х годах, и с тех пор раз за разом оказывается, что это один из лучших способов улучшения кода разработчиком, благодаря которому он еще и начинает глубже понимать функциональные требования к классу или методу.

Кент Бек (Kent Beck) ввел концепцию автономного тестирования в Smalltalk, а оттуда она перекочевала во многие другие языки программирования, превратившись в чрезвычайно полезную практику разработки ПО. Прежде чем двигаться дальше, я хочу дать более удачное определение автономного тестирования. Ниже приведено классическое определение, взятое из википедии. Оно будет постепенно эволюционировать на протяжении этой главы и в разделе 1.4 примет окончательную форму.

Определение 1.0. Автономный тест – это часть кода (обычно метод), которая вызывает другую часть кода и затем проверяет правильность некоторых предположений. Если предположения не подтверждаются, считается, что автономный тест завершился неудачно. Автономной единицей (unit) является метод или функция.

То, для проверки чего пишутся тесты, называется тестируемой системой (system under test – SUT).

Определение. Акроним SUT означает «тестируемая система», некоторые предпочитают использовать акроним CUT (class under test или code under test – тестируемый класс или тестируемый код). Мы будем называть объект тестирования SUT.

Раньше у меня было ощущение (именно ощущение – в этой книге нет науки, только искусство), что это определение автономного теста технически правильно, но за последние два года мое представление о том, что такое *автономная единица*, изменилось. Для меня *единица* означает «единица работы» внутри системы или «вариант использования» системы.

Определение

Единица работы — это совокупность действий от момента вызова какого-то открытого метода в системе до единственного конечного результата, заметного тесту системы. Этот конечный результат можно наблюдать, не исследуя внутреннее состояние системы, а только с помощью открытых API и поведения. Конечный результат может принимать следующие формы:

- вызванный открытый метод возвращает значение (т. е. является функцией, возвращающей не void);
- существует видимое изменение состояния или поведения системы до и после вызова, которое можно обнаружить, не опрашивая внутреннее состояние (примеры: в систему может войти ранее не существовавший пользователь или, если система представляет собой конечный автомат, то изменились ее свойства);
- имеет место обращение к сторонней системе, над которой у теста нет контроля, и эта сторонняя система не возвращает никакого значения либо возвращенное значение системой игнорируется (пример: обращение к сторонней системе протоколирования, которая была написана не вами и исходный код которой вам недоступен).

Идея единицы работы для меня означает, что *автономная единица* может охватывать как один-единственный метод, так и несколько классов и функций.

Возможно, вам кажется, что размер тестируемой автономной единицы следует сводить к минимуму. Мне тоже так казалось. Но больше не кажется. Я полагаю, что если удается создать более крупную единицу работы, конечный результат которой более явственно виден пользователю API, то и тесты окажутся более пригодными для сопровождения. Стараясь минимизировать размер единицы работы, вы в конце концов дойдете до того, что будете тестировать не конечные результаты, видимые пользователю открытого API, а промежуточные *остановки поезда* на пути к *конечному пункту назначения*. Я еще вернусь к этой теме при обсуждении избыточного специфицирования ниже (в основном, в главе 8).

Уточненное определение 1.1. Автономный тест – это часть кода, которая вызывает единицу работы и затем проверяет ее конечный результат. Если предположения о конечном результате не подтверждаются, считается, что автономный тест завершился неудачно. Объектом автономного тестирования может быть как единственный метод, так и совокупность нескольких классов.

Вне зависимости от используемого языка программирования один из самых трудных аспектов заключается в том, чтобы определить, какой автономный тест считать «хорошим».

1.1.1. О важности написания хороших автономных тестов

Понять, что такое единица работы, недостаточно.

Большая часть тех, кто пытается автономно тестировать свой код, либо в какой-то момент сдаются, либо выполняют код, который автономным тестом не является. На самом деле, они либо рассчитывают на проведение комплексных и интеграционных тестов на гораздо более поздней стадии жизненного цикла продукта, либо прибегают к ручному тестированию кода с помощью специально написанных тестовых приложений или конечного разрабатываемого продукта, который используют для вызова своего кода.

Не имеет смысла писать плохой автономный тест, если только это не первый шаг на пути постижения искусства написания хороших тестов. Если вы собираетесь написать автономный тест плохо, не осознавая этого, то лучше уж не писать его вовсе и избавить себя от хлопот, связанных с пригодностью для сопровождения и соблюдением сроков. Определив, что такое хороший автономный тест, мы можем быть уверены, что не отправимся в путь, не зная, куда направляемся.

Чтобы понять, что считать хорошим автономным тестом, нужно приглядеться к тому, что именно делают разработчики, когда что-то тестируют.

Как удостовериться в том, что сегодня код работает?

1.1.2. Все мы писали автономные тесты (или что-то в этом роде)

Возможно, вы удивитесь, услышав, что уже не раз сами писали те или иные автономные тесты. Вы хоть раз встречали разработчика, который не тестирует код до его сдачи? Я не встречал.

Возможно, вы пользовались консольным приложением, из которого вызывали различные методы класса или компонента, или специально написанным приложением с пользовательским интерфейсом на базе WinForms или Web Forms, которое проверяло функциональность класса или компонента. А быть может, вы даже тестировали код вручную, выполняя различные действия прямо из интерфейса реального приложения. Конечный результат всегда один – обретение субъективной уверенности в том, что код работает достаточно хорошо для передачи его кому-то другому.

Рис. 1.1. При классическом тестировании разработчик использует графический интерфейс пользователя (ГИП) для активации некоторого действия в тестируемом классе. Затем проверяет результат.

На рис. 1.1 показано, как тестирует код большинство разработчиков. Пользовательский интерфейс может отличаться, но принцип обычно неизменен: использовать внешнюю программу для повторяющейся проверки чего-то или запустить само приложение и вручную проверить его поведение.

Такие тесты, может быть, и полезны и, возможно, даже почти отвечают классическому определению автономного теста, но они очень далеки от того, что я называю *хорошим автономным тестом* в этой книге. И это подводит нас к первому и самому важному вопросу, с которым разработчик сталкивается при определении свойств хорошего автономного теста: что является автономным тестом, а что – нет?

1.2. Свойства хорошего автономного теста

Хороший автономный тест должен обладать следующими свойствами:

- он должен быть автоматизированным и повторяемым;
- его должно быть просто реализовать;
- он должен сохранять актуальность и завтра;

- любой должен иметь возможность выполнить его одним нажатием кнопки;
- он должен работать быстро;
- его результаты должны быть стабильны (тест всегда должен возвращать один и тот же результат, если между двумя последовательными запусками ничего не менялось);
- он должен полностью контролировать тестируемую автономную единицу;
- он должен быть полностью изолирован (работать независимо от других тестов);
- если тест завершается неудачно, то должно быть легко понять, каков ожидаемый результат и в каком месте искать ошибку.

Многие путают процесс тестирования своей программы с концепцией автономного теста. Для начала задайте себе следующие вопросы о тестах, которые вы уже писали ранее.

- Могу ли я выполнить и получить полезные результаты от автономного теста, который написал две недели, или месяц, или несколько лет назад?
- Может ли любой член моей команды выполнить и получить полезные результаты от автономных тестов, который я написал два месяца назад?
- Могу ли я прогнать все написанные мной автономные тесты максимум за несколько минут?
- Могу ли я прогнать все написанные мной автономные тесты одним нажатием кнопки?
- Могу ли я написать простой тест не более чем за несколько минут?

Если вы ответили отрицательно хотя бы на один вопрос, то с высокой вероятностью написанное вами автономным тестом не является. Это, безусловно, *какой-то* тест, и, возможно, он *не менее важен*, чем автономный, но по сравнению с тестами, для которых ответы на все вышеупомянутые вопросы положительны, у него имеются недостатки.

«Так что же я до сих пор делал?» – спрашиваете вы. Вы занимались интеграционным тестированием.

1.3. Интеграционные тесты

Я называю интеграционными любые тесты, которые не являются быстрыми и стабильными и пользуются одной или несколькими реально существующими зависимостями между тестируемыми автономными

единицами. Так, тесты, в которых используется системное время или реальная файловая система или реальная база данных, попадают в категорию интеграционных.

Например, если тест не контролирует системное время и в его коде используется текущее время DateTime.Now, то при каждом запуске мы на самом деле выполняем новый тест, потому что время изменяется. Такой тест уже нельзя назвать стабильным.

Само по себе это не плохо. Я считаю интеграционные тесты важным дополнением к автономным, просто их надо четко разделять, чтобы складывалось ощущение «безопасной зеленой зоны», о которой я буду говорить ниже.

Если в тесте используется реальная база данных, то он уже работает не только в памяти, а это значит, что его действия труднее аннулировать, чем при использовании одних лишь подставных данных в памяти. Кроме того, тест будет работать дольше, поскольку не способен контролировать реальность. Автономные тесты должны быть быстрыми, интеграционные обычно гораздо медленнее. Когда на руках сотни тестов, роль играют даже доли секунды.

Интеграционные тесты рискуют столкнуться и еще с одной проблемой: тестирование слишком многих аспектов за раз.

Что происходит, когда ваш автомобиль ломается? Как узнать, в чем проблема, не говоря уже о том, чтобы починить? Двигатель состоит из многих подсистем, каждая из них зависит от других, а все вместе они порождают конечный результат: машина едет. Если машина перестает ехать, ошибка может быть в любой подсистеме – и даже не в одной. Только интеграция всех подсистем (или уровней) дает машине возможность двигаться. Можно считать, что движение автомобиля – последний и решающий интеграционный тест всех составляющих ее частей. Если этот тест не проходит, вина лежит на всех частях; если проходит – то проходят тест и все части.

То же самое относится и к программному обеспечению. Большинство разработчиков тестирует функциональность своего кода с помощью окончательного пользовательского интерфейса. Нажатие кнопки запускает последовательность событий – классы и компоненты работают совместно, чтобы породить конечный результат. Если этот тест не проходит, то все программные компоненты, как одна команда, проваливаются, и тогда может быть трудно понять, что привело к ошибке операции в целом (рис. 1.2).

В книге Bill Hetzel «The Complete Guide to Software Testing» (Wiley, 1993) интеграционное тестирование определено как «упорядоченная

последовательность испытаний, в ходе которых программные и (или) аппаратные элементы объединяются и тестируются, до тех пор пока не будет собрана вся система». Это определение не вполне соответствует тому, что многие делают постоянно, считая это частью разработки и автономного тестирования, а не интеграционного тестирования системы.

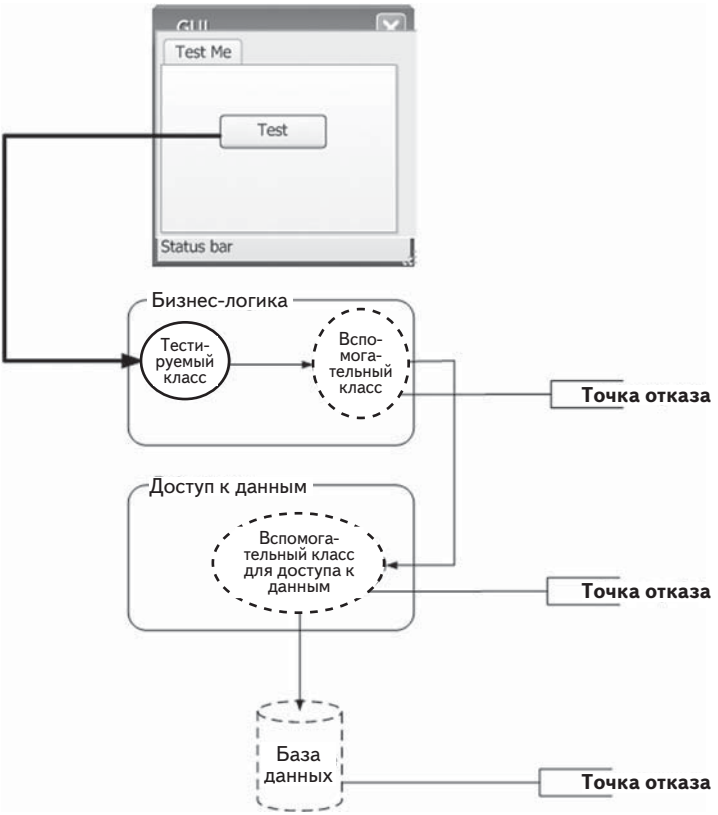


Рис. 1.2. В интеграционном тесте может быть много точек отказа. Все автономные единицы должны работать совместно, и каждый может отказать, поэтому найти источник ошибки не всегда просто.

Приведем более подходящее определение интеграционного тестирования.

Определение. Интеграционное тестирование – это тестирование единицы работы при отсутствии полного контроля над ней и с использованием одной

или нескольких реальных зависимостей, например, от времени, от сети, от базы данных, от потоков, от генераторов случайных чисел и т. д.

Подведем итог: в интеграционном тесте используются реальные зависимости, тогда как в автономных тестах единица работы изолируется от всех зависимостей, поэтому их результаты стабильны, и они способны легко управлять всеми аспектами поведения автономной единицы, моделируя нужное.

Вопросы, перечисленные в разделе 1.2, помогут вам оценить некоторые недостатки интеграционного тестирования. Попробуем определить, каких качеств мы ожидаем от хорошего автономного теста.

1.3.1. Недостатки неавтоматизированных интеграционных тестов по сравнению с автоматизированными автономными тестами

Обратим вопросы, поставленные в разделе 1.2, к интеграционным тестам и посмотрим, чего мы хотим достичь с помощью реальных автономных тестов.

- *Могу ли я выполнить и получить полезные результаты от автономного теста, который написал две недели, или месяц, или несколько лет назад?*

  Если нет, то как узнать, работает ли еще функция, реализованная ранее? В течение времени жизни приложения его код постоянно изменяется и, если вы не можете (или не хотите) прогонять тесты всех ранее работавших функций после каждого изменения, то можете ненароком что-то поломать. Я называю это явление «случайным внесением ошибок», чаще всего такое случается, когда проект подходит к концу и разработчики исправляют имеющиеся ошибки в условиях сильного стресса. Иногда, исправляя старые ошибки, они вносят новые. Правда, было бы неплохо узнавать о поломке в течение трех минут после того, как она произошла? Ниже вы узнаете, как этого добиться.

Определение. *Регрессией* называется одна или несколько единиц работы, которые когда-то работали, а теперь перестали.

- *Может ли любой член моей команды выполнить и получить полезные результаты от автономных тестов, который я написал два месяца назад?*

  Этот вопрос очень похож на предыдущий, но находится на ступеньку выше. Мы хотим быть уверены, что, внося изменения в свой код, не сломаем чей-то еще. Многие разработчики опасаются изменять код в унаследованных системах, поскольку не знают, какой код зависит от изменяемого. По существу, они рискуют перевести систему в неизвестное и, возможно, нестабильное состояние.

  Мало можно назвать вещей, столь же страшных, как неуверенность в том, работает еще приложение или уже нет, особенно если его код писали не вы. Если бы точно знать, что мы ничего не сломаем, то было бы совсем не так страшно приступать к незнакомому коду, потому что страховочная сетка автономных тестов всегда наготове.

  Хорошие тесты может выполнить любой.

Определение. Согласно википедии, *унаследованным* называется «исходный код, который рассчитан на более не поддерживаемую или снятую с производства операционную систему или иную компьютерную технологию», но во многих компаниях так называют просто старую версию приложения, которая в настоящее время поддерживается на правах унаследованного кода. Часто этот термин распространяют на код, с которым трудно работать, который трудно тестировать и даже просто читать.

Один мой клиент как-то определил унаследованный код приземленным образом: «код, который работает». Многим нравится такое определение: «код, для которого нет тестов». В книге Michael Feathers «Working Effectively with Legacy Code» (Prentice Hall, 2004) именно это определение унаследованного кода принято в качестве официального, его мы и будем иметь в виду в этой книге.

- *Могу ли я прогнать все написанные мной автономные тесты максимум за несколько минут?*

  Если вы не можете прогнать свои тесты быстро (секунды лучше минут), то будете прогонять их не так часто (раз в день, а то и раз в неделю или даже месяц). Но дело в том, что мы хотим узнать результат внесения изменений в код как можно быстрее, чтобы понять, не сломалось ли что-нибудь. Чем больше времени проходит между прогонами тестов, тем больше изменений успевают внести в систему и тем больше (много боль-

ше) мест приходится просматривать в поисках ошибки, если обнаруживается, что система перестала работать.

Хорошие тесты должны выполняться *быстро*.

- *Могу ли я прогнать все написанные мной автономные тесты одним нажатием кнопки?*

  Если не можете, то, вероятно, следует правильно сконфигурировать компьютер, на котором прогоняются тесты (например, задать строки соединения с базой данных), либо же ваши автономные не полностью автоматизированы. Если вам не удалось полностью автоматизировать тесты, то, скорее всего, вы будете избегать их частого запуска – как и все остальные члены команды.

  Никто не любит тратить время на настройку машины для запуска тестов только ради того, чтобы убедиться, что система все еще работает. Разработчики предпочитают заниматься более важными вещами, например, добавлять в систему новые функции.

  Хорошие тесты должно быть легко выполнить в том виде, в котором они написаны, – и не вручную.

- *Могу ли я написать простой тест не более чем за несколько минут?*

  Опознать интеграционный тест проще всего по тому, сколько времени уходит на его подготовку и реализацию, а не только на выполнение. Чтобы понять, как его написать, нужно учесть все внутренние, а иногда и внешние зависимости (базу данных можно считать внешней зависимостью) – на все это требуется время. Если вы не автоматизируете тесты, то наличие зависимостей мешает не так сильно, но ведь в этом случае вы утрачиваете все выгоды автоматизированного тестирования. Чем труднее написать тест, тем менее вероятно, что вы вообще будете писать новые тесты, а, если и будете, то только для проверки заботящих вас «существенных вещей». Но одна из сильных сторон автономных тестов состоит в том, что они проверяют все мелочи, которые могут сломаться, а не только «существенных вещи». Удивительно, сколько ошибок программист может найти в коде, который считал простым и безошибочным.

  Если вы концентрируете внимание только на крупных тестах, то покрытие логики программы тестами оказывается меньше.

Многие части базовой логики кода остаются не протестированными (пусть даже покрыто больше компонентов), и многие ошибки ускользают от рассмотрения.

Хорошие тесты системы пишутся легко и быстро, коль скоро вы поняли, какие принципы хотите применить к тестированию своей конкретной объектной модели. Но хочу предупредить: даже у разработчиков, собаку съевших на автономном тестировании, может уйти минут 30, а то и больше, чтобы понять, как должен выглядеть самый первый тест объектной модели, которую они раньше не тестировали. Это часть работы, от которой никуда не деться. Второй и последующие тесты той же объектной модели даются куда проще.

Приняв во внимание все сказанное о том, что не является автономным тестом, и о том, какие черты желательны, чтобы тестирование было полезным, я могу приступить к ответу на главный вопрос этой главы: что такое хороший автономный тест?

1.4. Из чего складывается хороший автономный тест?

Рассмотрев важные свойства, которыми должен обладать автономный тест, я готов дать окончательное определение.

Уточненное и окончательное определение 1.2. *Автономный тест* – это автоматизированная часть кода, которая вызывает тестируемую единицу работы и затем проверяет некоторые предположения о единственном конечном результате этой единицы. Автономный тест почти всегда пишется с помощью того или иного каркаса автономного тестирования. Написать его легко, а выполняется он быстро. Он заслуживает доверия, удобочитаем и пригоден для сопровождения. Он стабилен, т. е. его результаты не меняются, пока остается неизменным тестируемый код.

При первом знакомстве кажется, что это определение слишком высоко поднимает планку, особенно если учесть, сколько разработчиков пишут автономные тесты плохо. Оно заставляет нас критически пересмотреть наш прежний подход к тестированию и сравнить его с тем, как должно быть (заслуживающие доверия, удобочитаемые и пригодные для сопровождения тесты рассматриваются в главе 8).

В предыдущем издании этой книги я дал несколько иное определение автономного теста. Я говорил, что автономный тест «проверяет только код, где имеется управляющая логика». Теперь я так не счи-

таю. В состав единицы работы обычно входит и последовательный код тоже. Даже свойства, в коде которых нет никакой логики, используются в единице работы, пусть даже они не являются специальной мишенью для тестов.

Определение. *Кодом с управляющей логикой* называется любой код, в котором имеются какие-то логические конструкции, даже совсем незначительные, а именно: предложения if, циклы, предложения switch или case, вычисления и прочие элементы принятия решений.

Свойства (методы get и set в Java) – примеры кода, в котором обычно нет никакой логики, поэтому специально подвергать их тестированию необязательно. Скорее всего, этот код используется в тестируемой единице работы, но тестировать его напрямую нет смысла. Однако помните: стоит только добавить в код свойства какую-нибудь проверку, как надо будет эту логику протестировать.

В следующем разделе мы рассмотрим простой автономный тест, написанный без применения какого-либо каркаса тестирования (с каркасами тестирования мы начнем знакомиться в главе 2).

1.5. Пример простого автономного теста

Автоматизированный автономный тест можно написать и без каркаса тестирования. На самом деле, с тех пор как у разработчиков стало входить в привычку автоматизировать тестирование, я не раз видел, как они это делают, не подозревая о существовании каркасов. В этом разделе я покажу, как может выглядеть тест, созданный без использования каркаса, а в главе 2 мы сравним его с написанным для каркаса.

Допустим, у нас есть класс `SimpleParser` (приведен в листинге 1.1), и мы хотим его протестировать. В классе есть метод `ParseAndSum`, который принимает строку, состоящую из нуля или более чисел, разделенных запятыми. Если чисел в строке нет, метод возвращает 0. Если есть только одно число, оно возвращается в виде `int`. Если чисел несколько, они складываются и возвращается сумма (хотя в настоящий момент код умеет обрабатывать только случаи нуля или одного числа). Да, я знаю, что ветвь `else` лишняя, но из того, что ReSharper призывает вас спрыгнуть с моста, вовсе не следует, что так и надо делать. На мой взгляд, `else` делает код более удобочитаемым.

Листинг 1.1. Простой класс разбора, подлежащий тестированию

```
public class SimpleParser
{
  public int ParseAndSum(string numbers)
  {
     if(numbers.Length==0)
     {
        return 0;
     }
     if(!numbers.Contains(",”))
     {
        return int.Parse(numbers);
     }
     else
     {
        throw new InvalidOperationException(
          “Пока умею обрабатывать только 0 или 1 число!”);
     }
  }
}
```

Можно создать проект простого консольного приложения, которое ссылается на сборку с этим классом, и написать класс `SimpleParserTests`, показанный в листинге ниже. Тестовый метод вызывает метод *продуктового класса* (тестируемый) и проверяет возвращенное им значение. Если оно не совпадает с ожидаемым, тестовый метод выводит сообщение на консоль. Он также перехватывает все исключения и тоже печатает их на консоли.

Листинг 1.2. Простой написанный вручную метод для тестирования класса `SimpleParser`

```
class SimpleParserTests
{
  public static void TestReturnsZeroWhenEmptyString()
  {
     try
     {
        SimpleParser p = new SimpleParser();
        int result = p.ParseAndSum(string.Empty);
        if(result!=0)
        {
           Console.WriteLine(
           @”***SimpleParserTests.TestReturnsZeroWhenEmptyString:
           -------
           ParseAndSum должен вернуть 0 для пустой строки”);
        }
```

```
        }
        catch (Exception e)
        {
           Console.WriteLine(e);
        }
    }
}
```

Затем написанные тесты можно вызвать из выполняемого консольным приложением метода `Main`, который показан в следующем листинге. Метод `Main` в данном случае играет роль простого исполнителя тестов, который вызывает тесты один за другим, давая им возможность выводить что-то на консоль. Поскольку приложение выполняемое, оно может работать без вмешательства человека (при условии, что тесты не открывают интерактивных диалоговых окон).

Листинг 1.3. Прогон написанных вручную тестов с помощью простого консольного приложения

```
public static void Main(string[] args)
{
  try
  {
     SimpleParserTests.TestReturnsZeroWhenEmptyString();
  }
  catch (Exception e)
  {
     Console.WriteLine(e);
  }
}
```

Обязанность перехватывать исключения и выводить соответствующие сообщения на консоль (чтобы исключения не мешали выполнять последующие методы) возлагается на сами тестовые методы. По мере увеличения количества тестов мы можем добавлять в `Main` вызовы все новых и новых методов. Каждый тест отвечает за вывод описания проблемы (если таковая имеется) на экран.

Понятно, что приведенный выше тест написан на скорую руку. Если бы у нас было несколько подобных тестов, то, наверное, мы завели бы общий метод `ShowProblem`, которым могли бы пользоваться все тесты для единообразного форматирования сообщений об ошибках. Можно было бы также добавить вспомогательные методы для проверки объекта на null, обнаружения пустых строк и т. д. – чтобы не писать одни и те же длинные строки в нескольких тестах.

В листинге ниже показано, как мог бы выглядеть этот тест при наличии чуть более общего метода `ShowProblem`.

Листинг 1.4. Использование общего метода `ShowProblem`

```
public class TestUtil
{
  public static void ShowProblem(string test,string message )
  {
     string msg = string.Format(@"
     ---{0}---
     {1}
     --------------------
     ", test, message);
     Console.WriteLine(msg);
  }
}

public static void TestReturnsZeroWhenEmptyString()
{
  // Используем API отражения в .NET для получения имени
  // текущего метода.
  // Можно было бы зашить его в код, но этот
  // полезный прием следует знать.
  string testName = MethodBase.GetCurrentMethod().Name;
  try
  {
     SimpleParser p = new SimpleParser();
     int result = p.ParseAndSum(string.Empty);
     if(result!=0)
     {
        // Вызываем вспомогательный метод
        TestUtil.ShowProblem(testName,
           "ParseAndSum должен вернуть 0 для пустой строки");
     }
  }
  catch (Exception e)
  {
     TestUtil.ShowProblem(testName, e.ToString());
  }
}
```

В каркасах автономного тестирования вспомогательные методы носят более общий характер, поэтому и тесты писать проще. Об этом мы будем говорить в главе 2. Но перед тем как отправиться туда, хотелось бы обсудить еще один важный вопрос: не *как* писать автономный тест, а *когда* на протяжении этапа разработки делать это. Вот тут-то в игру и вступает методика разработки через тестирование.

1.6. Разработка через тестирование

Итак, мы знаем, как писать структурированные, пригодные для сопровождения и заслуживающие доверия тесты с помощью каркаса автономного тестирования. Теперь возникает следующий вопрос: когда это делать. Многие считают, что лучшее время для написания автономных тестов – по окончании разработки программы, однако растет число тех, кто предпочитает писать тесты *до* создания продуктового кода. Такой подход получил название «тесты сначала» или «разработка через тестирование» (test-driven development – TDD).

Примечание. Существуют разные точки зрения на точный смысл разработки через тестирование. Одни говорят, что все дело в написании тестов сначала, другие – что в большом количестве тестов. Кто-то считает, что это способ проектирования, а кто-то – что таким образом можно было бы определять поведение программы, уделяя собственно проектированию гораздо меньше внимания. Более полный обзор различных точек зрения на TDD см. в статье «The various meanings of TDD» в моем блоге (http://osherove.com/blog/2007/10/8/the-various-meanings-of-tdd.html). В этой книге под TDD будет пониматься процесс разработки, при котором тесты пишутся сначала, а проектирование играет вторичную роль (и обсуждаться не будет).

На рис. 1.3 и 1.4 показаны различия между традиционным кодированием и TDD.

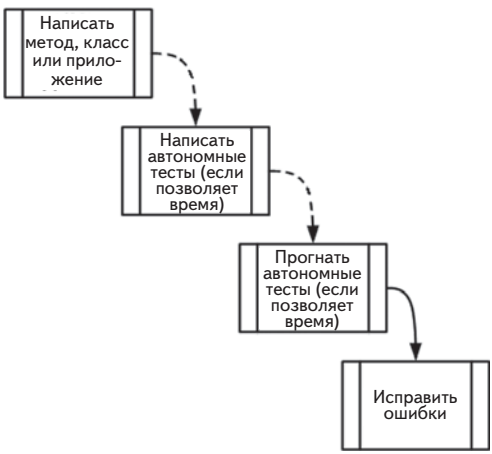


Рис. 1.3. Традиционный способ написания автономных тестов. Пунктирными линиями представлены действия, которые принято считать необязательными

TDD отличается от традиционной разработки, как показывает рис. 1.4. Мы начинаем с теста, который не проходит. Затем переходим к созданию продуктового кода, добиваясь, чтобы тест прошел, после приступаем либо к рефакторингу кода, либо к созданию следующего безуспешного теста.

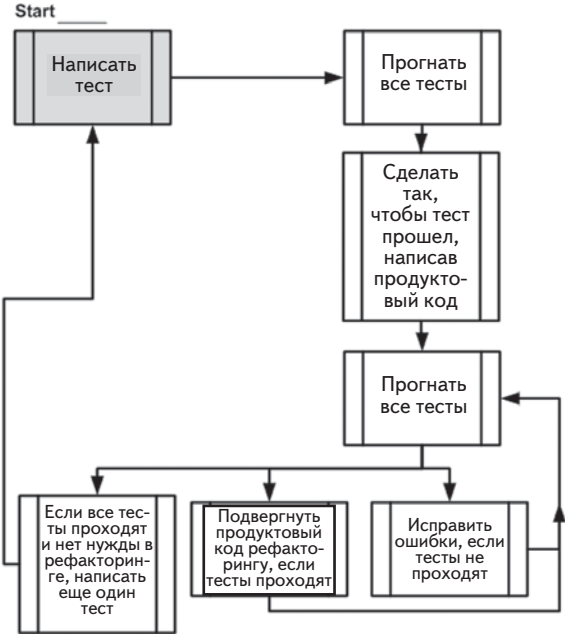


Рис. 1.4. Разработка через тестирование – взгляд с высоты птичьего полета. Обратите внимание на спиральный характер процесса: написать тест, написать код, выполнить рефакторинг, написать следующий тест. Это демонстрация инкрементной природы TDD: мелкие шаги приводят к высококачественному конечному результату

Эта книга посвящена технике написания хороших автономных тестов, а не разработке через тестирование, но я сам большой поклонник TDD. Я написал несколько крупных приложений и каркасов с применением TDD, руководил командами, использующими TDD в работе, и провел более сотни курсов и семинаров по TDD и методикам автономного тестирования. На собственном опыте я убедился, что TDD помогает создавать высококачественный код и высококачественные тесты, причем структура кода оказывается лучше, чем раньше. Я убежден, что вы сможете применить эту методику с выгодой, но

за это придется заплатить (временем на изучение, внедрение и т. п.). Впрочем, затраты окупятся сторицей.

Важно понимать, что TDD ничего не гарантирует: и проект может провалиться, и тесты могут получиться нестабильными или непригодными для сопровождения. Очень легко поддаться обаянию самой методики TDD и перестать обращать внимание на способ написания автономных тестов: выбор имен, акцент на удобочитаемость и пригодность для сопровождения. Легко даже упустить из виду, то ли они тестируют, что нужно, и не содержат ли ошибок. Вот потому-то я и пишу эту книгу.

Методика TDD очень проста.

1. Написать тест, который не проходит, и тем самым доказать, что в конечном продукте отсутствует некий код или функциональность. Тест пишется так, будто продуктовый код уже работает, поэтому отказ теста означает, что в продуктовом коде есть ошибка. Если бы я хотел добавить в класс калькулятора функцию запоминания последней суммы `LastSum`, то написал бы тест, проверяющий, что `LastSum` действительно содержит правильное значение. Этот тест не откомпилировался бы, поэтому я добавил бы ровно столько кода, чтобы компиляция прошла успешно (не реализуя запоминание числа), и снова прогнал бы код. На этот раз он выполнился бы, но с ошибкой, потому что функциональность еще не реализована.
2. Сделать так, чтобы тест прошел, написав продуктовый код, отвечающий ожиданиям теста. Продуктовый код должен быть максимально простым.
3. Подвергнуть его рефакторингу, т. е. переработать его. Добившись успешного завершения теста, мы вправе либо перейти к следующему тесту, либо переработать код: сделать его более удобочитаемым, устранить дублирование и т. д.

Заниматься рефакторингом можно после написания нескольких тестов или после каждого теста. Это важный шаг, он повышает удобочитаемость и сопровождаемость кода, гарантируя, что все ранее написанные тесты проходят.

Определение. Под *рефакторингом* понимается изменение части кода *без* изменения его функциональности. Переименование метода – это рефакторинг. Разбиение длинного метода на несколько более коротких – тоже рефакторинг. В обоих случаях программа делает то же, что и раньше, но ее становится проще сопровождать, читать, отлаживать и изменять.

Формулировка шагов TDD выглядит технической, но за ней стоит обычная житейская мудрость. При правильном применении TDD качество кода устремится ввысь, количество ошибок уменьшится, ваша уверенность в правильности программы повысится, время поиска ошибок сократится, структура кода станет совершеннее, а начальник будет доволен. Напротив, при неправильном применении TDD работа над проектом будет выбиваться из графика, время – тратиться впустую, мотивация – ослабевать, а качество кода – снижаться. Это палка о двух концах, и многие осознали это на собственной шкуре.

Технически, одно из важнейших достоинств TDD, о котором вам никто не расскажет, состоит в том, что, наблюдая, как тест падает, а затем проходит, хотя вы ничего в нем самом не изменяли, вы по сути дела тестируете сам тест. Если вы ожидаете, что тест не должен пройти, а он проходит, то, вероятно, есть ошибка в самом тесте или вы тестируете не то, что нужно. Если ранее падавший тест должен пройти, но все равно не проходит, значит, либо в тесте ошибка, либо вы не того ожидаете.

В этой книге речь пойдет об удобочитаемых, пригодных для сопровождения и заслуживающих доверия тестах, но наибольшую уверенность в своих тестах вы получаете, когда видите, как они падают и проходят в должное время. TDD в этом очень помогает, и это одна из причин, по которой разработчики, практикующие TDD, гораздо меньше занимаются отладкой, чем те, кто автономно тестирует свой код постфактум. Если доверяешь тесту, то нет нужды отлаживать просто «на всякий случай». А доверие возникает, когда видишь обе стороны теста: ту, что падает, и ту, что проходит, – то и другое в свое время.

1.7. Три основных навыка успешного практика TDD

Для успешной разработки через тестирование нужно обладать тремя основными навыками: знать, как писать хорошие тесты, писать их раньше кода и правильно структурировать тесты.

- *Из того, что вы пишете тесты сначала, еще не следует, что они будут удобочитаемыми, пригодными для сопровождения или заслуживающими доверия.* Навыки написания хороших автономных тестов – как раз то, о чем написана эта книга.
- *Из того, что вы пишете удобочитаемые и пригодные для сопровождения тесты, еще не следует, что вы получите те же*

выгоды, что при написании их сначала. Умение писать тесты сначала – это то, чему учат книги по TDD (по крайней мере, большая их часть), хотя о навыках написания хороших тестов в них нет ни слова. Особенно я рекомендовал бы книгу Kent Beck «Test-Driven Development: by Example»[1] (Addison-Wesley Professional, 2002).

- *Из того, что вы пишете тесты сначала, и они являются удобочитаемыми и пригодными для сопровождения, еще не следует, что в итоге получится хорошо структурированная система.* Только навыки проектирования позволяют сделать код красивым и пригодным для сопровождения. По этой теме я рекомендую книги Steve Freeman, Nat Pryce «Growing Object-Oriented Software, Guided by Tests» (Addison-Wesley Professional, 2009) и Robert C. Martin «Clean Code»[2] (Prentice Hall, 2008).

Прагматический подход к изучению TDD состоит в освоении всех трех аспектов по отдельности и по очереди. Я рекомендую такой подход, потому что часто видел, как люди пытались освоить все три навыка одновременно, тратили на это огромные усилия и в конце концов сдавались, потому что гора оказалась слишком крутой.

Приняв постепенный подход к изучению этой области знаний, вы избавите себя от постоянного страха сделать что-то не так в той части, которая в данный момент находится на периферии внимания.

Что же касается конкретного порядка изучения, то у меня нет какой-то определенной схемы. Буду рад, если вы расскажете о своем опыте и дадите свои рекомендации по приобретению этих навыков. О том, как со мной связаться, написано на сайте http://osherove.com.

1.8. Резюме

В этой главе я дал определение хорошего автономного теста как теста, обладающего следующими свойствами.

- Это автоматизированный код, который вызывает какой-то метод и затем проверяет предположения о логике поведения этого метода или класса.
- Он написан с применением каркаса автономного тестирования.

[1] Кент Бек «Экстремальное программирование: разработка через тестирование». Питер, 2003. – *Прим. перев.*

[2] Роберт Мартин «Чистый код». Питер, 2013. – *Прим. перев.*

- Его легко написать.
- Он быстро работает.
- Его может многократно выполнять любой член команды разработчиков.

Чтобы понять, что такое автономная единица, нужно разобраться, какого рода тестированием вы занимались до сих пор. Мы назвали этот тип тестирования интеграционным, потому что тестируется набор автономных единиц, зависящих друг от друга.

Важно понимать разницу между автономными и интеграционными тестами. Этим знанием вы будете пользоваться в повседневной работе, решая, куда поместить тест, какие тесты когда писать и какой вариант лучше подходит в конкретной ситуации. Оно поможет также исправить уже имеющиеся проблемы с тестами, которые стали сильно докучать.

Мы также остановились на минусах интеграционного тестирования без поддерживающего каркаса: такие тесты трудно писать и автоматизировать, они работают медленно и требует предварительного конфигурирования. И хотя мы не отказывается от интеграционных тестов в проекте, автономные тесты гораздо полезнее на ранних этапах, когда ошибки не так серьезны, их проще искать и объем подлежащего просмотру кода меньше.

Наконец, мы рассмотрели методику разработки через тестирование, рассказали, чем она отличается от традиционного кодирования и каковы ее основные преимущества. TDD позволяет достичь очень высокого (близкого к ста процентам для *логического* кода) покрытия кода тестами (какая часть кода выполнена в результате прогона тестов). TDD также вселяет уверенность в то, что тестам можно доверять. TDD «тестирует тесты» в том смысле, что позволяет наблюдать, как они сначала не проходят, а потом проходят. У TDD есть и много других достоинства, например: содействие в проектировании, снижение сложности и помощь в решении сложных проблем шаг за шагом. Но невозможно в полной мере овладеть TDD, не научившись писать хорошие тесты.

В следующей главе мы напишем первые автономные тесты с применением NUnit – каркаса тестирования, ставшего стандартом де факто для разработчиков на платформе .NET.

ГЛАВА 2.
Первый автономный тест

В этой главе:

- Обзор каркасов автономного тестирования для .NET.
- Создание первого теста с помощью NUnit.
- Использование атрибутов NUnit.
- Три типа результатов единицы работы.

Когда я впервые начал писать тесты с применением настоящего каркаса автономного тестирования, документации почти не было, а для каркасов, с которыми я работал, не существовало достойных примеров (в то время я писал в основном на VB 5 и 6). Изучать их было тяжело, и поначалу я писал довольно скверные тесты. К счастью, времена изменились.

В этой главе вы начнете писать тесты, даже если понятия не имеете, с чего начинать. Вы уверенно встанете на путь создания самых что ни на есть реальных автономных тестов с помощью каркаса NUnit для платформы .NET. Это мой любимый каркас автономного тестирования в .NET, потому что с ним легко работать и в нем есть масса полезных возможностей.

Для .NET существуют и другие каркасы, и некоторые из них содержат больше функций, но начинаю я всегда с NUnit. Если возникает необходимость, я иногда перехожу на другой каркас. Мы рассмотрим, как работает NUnit, познакомимся с его синтаксисом, научимся прогонять в нем тесты и смотреть, прошли они или нет. Для этого мы создадим небольшой программный проект, на котором будет изучать различные приемы и передовые практики тестирования на протяжении всей книги.

Возможно, вам кажется, что я грубо навязываю NUnit. Почему не воспользоваться каркасом MSTest, встроенным в Visual Studio? Ответ состоит из двух частей.

- NUnit лучше, чем MSTest, приспособлен к написанию автономных тестов, а нем имеются атрибуты, позволяющие писать более удобочитаемые и пригодные для сопровождения тесты.
- Встроенный в Visual Studio 2012 исполнитель тестов, позволяет прогонять тесты, написанные для других каркасов, в том числе NUnit. Для этого нужно лишь установить адаптер NUnit для Visual Studio с помощью NuGet (о том, как работать с NuGet, я расскажу ниже в этой главе).

Мне этих аргументов достаточно для выбора каркаса.

Для начала рассмотрим, что такое каркас автономного тестирования и что он позволяет делать такого, что без него было бы достичь трудно или невозможно.

2.1. Каркасы автономного тестирования

Ручные тесты – отстой. Вы пишете свой код, запускаете его в отладчике, нажимаете клавиши, заставляющие приложение делать то, что вам надо, а затем повторяете все это снова всякий раз, как добавился новый код. И нужно все время помнить, как новый код может повлиять на старый. Ручной работы все больше. Факт.

Выполнение тестов и регрессионного тестирования полностью вручную, повторяя одни и те же действия, как мартышка, – процесс, отнимающий много времени и чреватый ошибками. Из всего связанного с разработкой ПО это самая ненавистная программистам деятельность. Проблему можно смягчить с помощью инструментальных средств. Каркасы автономного тестирования помогают писать тесты быстрее, используя документированный API, выполнять их автоматически и легко получать наглядное представление результатов. И каркасы ничего не забывают! Посмотрим внимательнее, что они нам предлагают.

2.1.1. Что предлагают каркасы автономного тестирования

Многие читатели этой книги при написании тестов сталкивались со следующими ограничениями.

- *Тесты не были структурированы*. Приходилось изобретать колесо всякий раз, как нужно было протестировать какую-то

функцию. Один тест оформлялся в виде консольного приложения, для другого нужна была форма с графическим интерфейсом, для третьего – веб-форма. На тестирование не хватало времени, тесты не удовлетворяли требованию «простоты реализации».

- *Тесты не были повторяемыми.* Ни вы, ни члены команды не могли прогнать тесты, написанные в прошлом. Тем самым нарушалось требование «повторяемости» и затруднялся поиск регрессионных ошибок. Каркас позволяет просто и автоматически писать повторяемые тесты.
- *Тесты не покрывают все важные части кода.* Тестируются не все существенные участки кода, т. е. участки, содержащие какую-то логику, хотя каждый из них потенциально может содержать ошибку. (Методы чтения и установки свойств не содержат логики, но входят в состав какой-то единицы работы.) Если бы писать тесты было проще, то у вас было бы больше желания этим заниматься и обеспечивать лучшее покрытие.

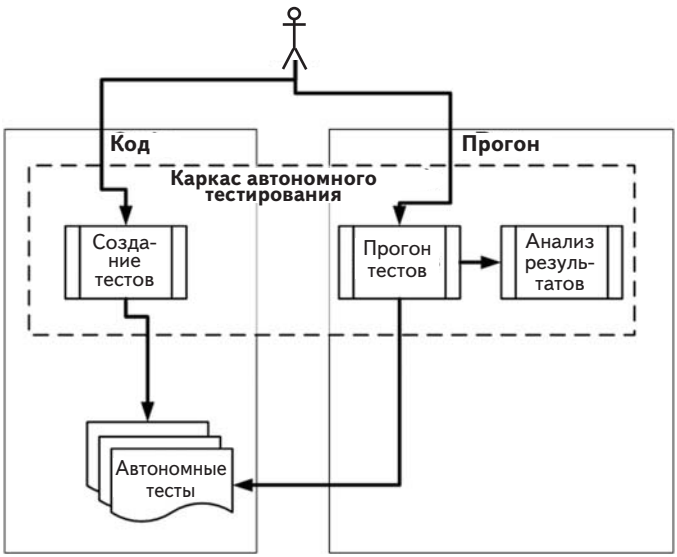


Рис. 2.1. При написании автономных тестов используются библиотеки, входящие в состав каркаса тестирования. Затем тесты прогоняются с помощью специального инструмента или непосредственно в IDE, а результаты (представленные в виде текста или в графическом интерфейсе каркаса) анализируются разработчиком или автоматизированной процедурой сборки

Короче говоря, вам не хватает *каркаса* для создания, прогона и анализа результатов тестов. На рис. 2.1 показаны те этапы разработки программного обеспечения, к которым имеет отношение каркас автономного тестирования.

Каркасы включают библиотеки и модули, помогающие разработчикам проводить автономное тестирование своего кода (см. табл. 2.1). Но у них есть и другая сторона – прогон тестов в составе автоматизированной сборки; об этом я расскажу в последующих главах.

Таблица 2.1. Как каркас автономного тестирования помогает разработчику создавать, прогонять и анализировать результаты тестов

Аспект автономного тестирования	Чем помогает каркас
Простота и упорядоченность написания тестов	Каркас предоставляет разработчику библиотеку классов, которая содержит: • базовые классы и интерфейсы, которым можно унаследовать; • атрибуты, помечающие, какие методы являются тестовыми; • классы утверждений, в которых имеются специальные методы для верификации кода.
Выполнение одного или всех тестов	Каркас включает в себя исполнитель тестов (консольный или графический инструмент), который: • находит в коде тесты; • автоматически выполняет их; • отображает состояние во время выполнения; • допускает автоматизацию путем запуска из командной строки.
Анализ результатов прогона тестов	Исполнитель тестов обычно предоставляет следующую информацию: • сколько тестов было выполнено; • сколько тестов не было выполнено; • сколько тестов не прошло; • какие тесты не прошли; • почему тесты не прошли; • сообщение, указанное вами при вызове метода `ASSERT`; • место в коде, где была обнаружена ошибка; • возможно, полную трассировку стека в случае исключения, приведшего к ошибке; при этом имеется возможность перейти в точку вызова различных методов, перечисленных к стеке.

На момент написания этой книги существовало более 150 каркасов автономного тестирования – практически для любого сколько-

нибудь распространенного языка программирования. Достойный список можно найти по адресу http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks. Кстати, для одной лишь платформы .NET имеется по меньшей мере три активно поддерживаемых каркаса автономного тестирования: MSTest (от Microsoft), xUnit.net и NUnit. При этом NUnit когда-то был стандартом де факто. Сейчас идет борьба между MSTest и NUnit – просто потому, что MSTest уже встроен в Visual Studio. Но если у меня есть выбор, я предпочитаю NUnit ради некоторых возможностей, о которых пойдет речь ниже в этой главе, а также в приложении, посвященном инструментам и каркасам.

Примечание. Само по себе использование каркаса автономного тестирования еще не гарантирует, что написанные вами тесты будут *удобочитаемыми*, *пригодными для сопровождения* и *заслуживающими доверия* или что они будут покрывать всю логику, которую вы хотели бы протестировать. Как добиться, чтобы автономные тесты обладали этими свойствами, мы будем обсуждать в главе 7 и в других местах книги.

2.1.2. Каркасы семейства xUnit

Термин *каркасы xUnit* закрепился за этими каркасами автономного тестирования, потому что их названия обычно начинаются с первой буквы языка программирования, для которого каркас предназначен. Для C++ это CppUnit, для Java – JUnit, для .NET – NUnit, а для Haskell – HUnit. Не все, но большинство каркасов следуют этому соглашению об именовании.

Мы в этой книге будем использовать каркас NUnit для .NET, который упрощает написание, прогон и анализ результатов тестов. NUnit появился на свет в результате прямого переноса широко известного каркаса JUnit для Java, но с тех пор сделал гигантский шаг вперед в части структуры и удобства использования, далеко отошел от своего прародителя и вдохнул новую жизнь в целую экосистему каркасов тестирования, которая все больше и больше изменяется. Обсуждаемые ниже концепции будут понятны также программистам на Java и C++.

2.2. Знакомство с проектом LogAn

Для изучения тестирования мы в этой книге используем проект, который поначалу будет совсем простым, состоящим всего из одного клас-

са. По ходу дела мы будем добавлять в него новые классы и возможности. Проект назовем LogAn («log and notification» – протоколирование и уведомление).

Опишем сценарий. Предположим, что у компании имеется много внутренних продуктов, которые используются для мониторинга ее приложений в местах установки у заказчиков. Все они заносят информацию в файлы журналов, размещенные в специальном каталоге. Журналы пишутся в придуманном компанией закрытом формате, который не может быть разобран имеющимися на рынке инструментами. Ваша задача – написать программу LogAn, которая умеет анализировать файлы журналов и находить в них особые случаи и события. Обнаружив нечто представляющее интерес, программа должна уведомлять соответствующих лиц.

В этой книге я научу вас писать тесты, которые проверяют правильность работы LogAn в части разбора, распознавания событий и уведомления. Но перед тем как приступить к тестированию этого проекта, посмотрим, как вообще пишутся автономные тесты в NUnit. Для начала необходимо каркас установить.

2.3. Первые шаги освоения NUnit

Любой новый инструмент нужно сначала установить. Поскольку NUnit – бесплатная программа с открытыми исходными текстами, то это довольно простая задача. Справившись с ней, мы затем начнем писать тесты в NUnit, научимся пользоваться встроенными атрибутами, прогонять тесты и получать результаты прогона.

2.3.1. Установка NUnit

Проще всего установить NUnit, воспользовавшись NuGet – бесплатным расширением Visual Studio, которое позволяет искать, загружать и устанавливать ссылки на популярные библиотеки прямо из Visual Studio. Для этого достаточно нескольких щелчков мышью или ввода простой текстовой команды.

Я настоятельно рекомендую установить NuGet: перейдите в меню **Tools → Extension Manager** (Сервис → Диспетчер расширений), щелкните по ссылке Online Gallery (Каталог в Интернете) и установите пакет NuGet Package Manager, имеющий один из самых высоких рейтингов. После установки перезапустите Visual Studio и вот – к вашим услугам мощнейший и очень простой в использовании инструмент

для добавления ссылок в проекты. (Читатели, знакомые с Ruby, обратят внимание на сходство NuGet с Ruby Gems и идеей gem-пакета, хотя имеются существенные новации в части функций, относящихся к версионированию и развертыванию в производственной среде.)

Установив NuGet, вы можете открыть меню **Tools → Library Package Manager → Package Manager Console** (Сервис → Диспетчер библиотечных пакетов → Консоль диспетчера пакетов) и ввести команду `Install-Package NUnit` в открывающемся окне (при вводе названий команд и имен библиотечных пакетов можно пользоваться клавишей Tab для автозавершения).

Когда все будет сделано, вы увидите приятное сообщение «NUnit Installed Successfully» (NUnit успешно установлен). NuGet скачал на ваш диск zip-файл, содержащий файлы NUnit, добавил ссылку в проект по умолчанию, установленный в раскрывающемся списке в окне консоли диспетчера пакетов, и, закончив свои дела, сообщил вам об этом. В проекте должна появиться ссылка на NUnit.Framework.dll.

Одно замечание касательно пользовательского интерфейса NUnit – это простой исполнитель тестов, являющийся частью NUnit. Я расскажу о нем ниже, но сам обычно им не пользуюсь. Считайте, что это скорее учебное средство, позволяющее понять, как NUnit работает сам по себе, без интеграции с Visual Studio. Кстати, оно и не включено в дистрибутив NUnit, загруженный NuGet. NuGet устанавливает только необходимые DLL, но не пользовательский интерфейс (и это разумно, потому что проектов, в которых используется NUnit, может быть много, но вряд ли вы хотите, чтобы в каждом присутствовала копия пользовательского интерфейса для запуска тестов). Чтобы получить пользовательский интерфейс NUnit, вы можете установить из NuGet пакет NUnit.Runners или зайти на сайт NUnit.com и скачать оттуда полную версию. В состав полной версии входит также программа NUnit Console Runner, которой можно пользоваться для прогона тестов на сервере сборки.

Если у вас нет доступа к NUnit, можете скачать его с сайта www.NUnit.com и добавить ссылку на nunit.framework.dll вручную.

Ну и поскольку исходный код NUnit открыт, вы можете скачать его, откомпилировать самостоятельно и пользоваться как угодно в рамках лицензии (подробности см. в файле license.txt, который находится в инсталляционном каталоге программы).

Примечание. На момент написания этой книги последняя версия NUnit имела номер 2.6.0. Приведенные примеры должны быть совместимы с будущими версиями этого каркаса.

Если вы решите устанавливать NUnit вручную, запустите скачанную программу установки. Установщик поместит на рабочий стол ярлык, указывающий на пользовательский интерфейс исполнителя тестов, а основные части программы будут находиться в каталоге с именем вида C:\Program Files\NUnit-Net-2.6.0. Дважды щелкнув по значку NUnit на рабочем столе, вы увидите окно исполнителя тестов, показанное на рис. 2.2.

Примечание. Для проработки примеров из этой книги достаточно экспресс-выпуска Visual Studio C# Express Edition (или более полной версии).

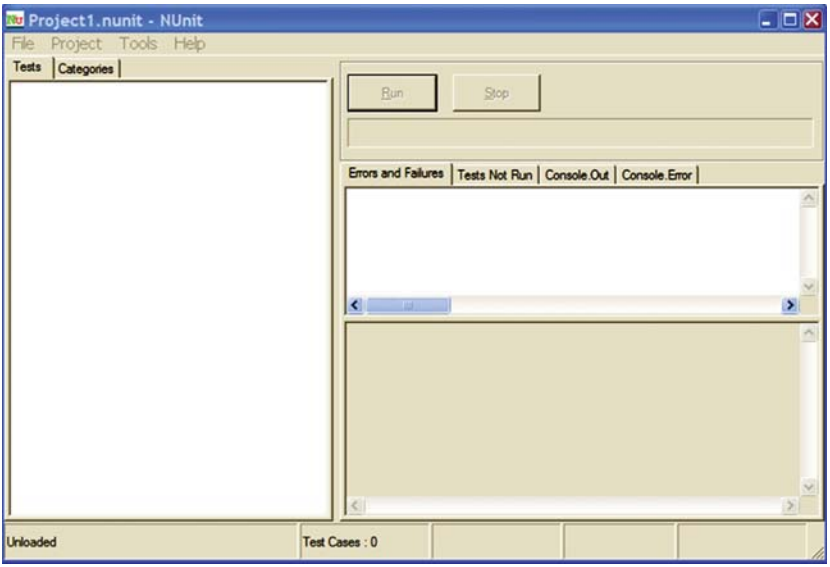


Рис. 2.2. Пользовательский интерфейс NUnit состоит из трех основных частей: дерева тестов слева, области сообщений и ошибок справа вверху и трассировки стека справа внизу

2.3.2. Загрузка решения

Если вы скачали на свою машину исходный код для этой книги, то загрузите в Visual Studio 2010 (или более поздней версии) решение ArtOfUnitTesting2ndEd.Samples.sln из папки Code.

Мы начнем с тестирования следующего простого класса, содержащего всего один метод (тестируемая автономная единица):

```
public class LogAnalyzer
{
  public bool IsValidLogFileName(string fileName)
  {
     if(fileName.EndsWith(".SLF"))
     {
        return false;
     }
     return true;
  }
}
```

Обратите внимание, что я сознательно опустил знак `!` в условии `if`, из-за чего этот метод работает неправильно – возвращает `false` вместо `true`, если имя файла кончается строкой `.SLF`. Я сделал это для того, чтобы вы увидели, что показывает исполнитель тестов, когда тест не проходит.

Метод не кажется сложным, но мы все же протестируем его работоспособность – главным образом, для того чтобы познакомиться с процедурой тестирования. В реальной программе желательно тестировать все методы, содержащие логику, даже если она совсем простая. В логику могут вкрасться ошибки, и мы хотим знать об этом. В следующих главах мы будем тестировать более сложные сценарии и логику.

Этот метод решает, является ли некий файл допустимым файлом журнала, анализируя расширение имени. В первом тесте мы зададим допустимое имя и проверим, что метод возвращает `true`.

Ниже перечислены первые шаги написания автоматизированного теста метода `IsValidLogFileName`.

1. Добавьте в решение новый проект библиотеки классов, в котором будут храниться тестовые классы. Назовите проект LogAn.UnitTests (предполагается, что основной проект называется LogAn.csproj).
2. Добавьте в эту библиотеку новый класс, который будет содержать тестовые методы. Назовите его `LogAnalyzerTests` (в предположении, что тестируемый класс называется `LogAnalyzer`).
3. Добавьте в этот класс метод `IsValidLogFileName_BadExtension_ReturnsFalse()`.

О стандартах именования тестов и организации файлов мы еще будем говорить ниже, но основные правила приведены в табл. 2.2.

Таблица 2.2. Основные правила размещения и именования тестов

Тестируемый объект	Объект, создаваемый для тестирования
Проект	Создать тестовый проект с именем `[ProjectUnderTest].UnitTests`
Класс	Для класса из проекта `ProjectUnderTest` создать класс с именем `[ClassName]Tests`
Единица работы (метод или логическая группа нескольких методов или нескольких классов)	Для каждой единицы работы создать тестовый метод с именем `[UnitOfWorkName]_[ScenarioUnderTest]_ [ExpectedBehavior]`. Именем единицы работы (`UnitOfWorkName`) может быть как имя метода (если метод и представляет собой законченную единицу работы), так и нечто более абстрактное, если это сценарий, охватывающий несколько методов и классов, например: `UserLogin` или `RemoveUser` или `Startup`. Можно начать с имен методов и переходить к более абстрактным именам позже. Но это должны быть открытые методы, иначе они не будут представлять начало единицы работы.

Наш тестовый проект называется LogAn.UnitTests. А класс для тестирования `LogAnalyzer` называется `LogAnalyzerTests`.

Опишем подробнее три части имени тестового метода.

- `UnitOfWorkName` – имя тестируемого метода либо группы методов или классов.
- `Scenario` – условия, при которых тестируется автономная единица, например: «bad login» (неверное имя входа) или «invalid user» (несуществующий пользователь) или «good password» (правильный пароль). Можно описать параметры, передаваемые открытому методу, или начальное состояние системы в момент вызова единицы работы, например: «system out of memory» (не хватает памяти) или «no users exist» (нет ни одного пользователя) или «user already exists» (пользователь уже существует).
- `ExpectedBehavior` – что должен делать метод при заданных условиях. Существует три возможности: вернуть результат в виде значения (или исключения), изменить состояние системы (например, добавить в систему нового пользователя, так что при следующем входе поведение системы изменится) или обратиться к сторонней системе (например, внешней вебслужбе).

В нашем тесте метода `IsValidLogFileName` сценарий заключается в том, что методу передается допустимое имя файла, а ожидаемое поведение – в том, что метод должен вернуть `true`. Тестовый метод можно было бы назвать `IsValidFileName_BadExtension_ReturnsFalse()`.

Включать ли тесты в проект с продуктовым кодом? Или лучше вынести их в отдельный проект? Я предпочитаю второе, потому что при этом упрощаются все остальные вещи, относящиеся к тестам. Кроме того, многие разработчики терпеть не могут включать тесты в продуктовый код, потому что это ведет к уродливым схемам с условной компиляцией и прочим осложнениям, которые делают код неудобочитаемым.

Я не склонен воевать по этому поводу. Но мне нравится размещать тесты в стороне от продуктового кода, чтобы можно было проверить его работоспособность после развертывания. Это, конечно, требует тщательного планирования, но *не* требует хранения кода и тестов в одном проекте. Так что *можно* и рыбку съесть, и косточкой не подавиться.

Мы еще не приступили к использованию каркаса NUnit, но близки к этому. Еще нужно добавить в тестовый проект ссылку на тестируемый проект. Для этого щелкните правой кнопкой мыши по тестовому проекту и выберите команду **Add Reference** (Добавить ссылку). Затем перейдите на вкладку **Projects** (Проекты) и выберите проект LogAn.

Далее мы научимся помечать тестовые методы, чтобы NUnit автоматически загружал и выполнял их. Но сначала проверьте, что ссылка на NUnit добавлена автоматически NuGet или вручную, как описано в разделе 2.3.1.

2.3.3. Использование атрибутов NUnit

В NUnit для опознания и загрузки тестов применяются атрибуты. Как закладки в книге, атрибуты позволяют каркасу находить интересующие его элементы в загруженной сборке и решать, какие тесты следует вызывать.

В состав NUnit входит сборка, содержащая эти специальные атрибуты. Нужно лишь добавить в тестовый (не продуктовый!) проект ссылку на сборку NUnit.Framework. Эту сборку вы найдете на вкладке .NET в диалоговом окне **Add Reference** (Добавление ссылки) (если для установки NUnit использовался NuGet, то этот шаг необязателен). После ввода строки NUnit вы увидите несколько сборок, имена которых начинаются этим словом.

Добавьте в тестовый проект ссылку на nunit.framework.dll (если устанавливали NUnit вручную, а не через NuGet).

Чтобы знать, какие методы вызывать, исполнителю NUnit нужны по меньшей мере два атрибута.

- Атрибут `[TestFixture]` помечает класс, содержащий автоматизированные тесты (было бы понятнее, если бы вместо `Fixture` фигурировало слово `Class`, но при такой замене код не откомпилируется). Поместите этот атрибут в начало класса `LogAnalyzerTests`.
- Атрибутом `[Test]` помечаются методы, которые следует вызывать при автоматизированном прогоне тестов. Поместите этот атрибут в начало тестового метода.

По завершении тестовый код должен выглядеть следующим образом:

```
[TestFixture]
public class LogAnalyzerTests
{
  [Test]
  public void IsValidFileName_BadExtension_ReturnsFalse()
  {
  }
}
```

Совет. NUnit требует, чтобы все тестовые методы были открыты, возвращали void и, как правило, не принимали параметров. Впрочем, мы увидим, что иногда тесты могут принимать параметры!

Итак, мы пометили класс и подлежащий выполнению метод. Теперь NUnit по первому требованию выполнит код, который мы поместим в тестовый метод.

2.4. Создание первого теста

Как тестировать свой код? Автономный тест обычно состоит из трех частей.

1. Подготовка (Arrange) объектов, то есть создание и настройка.
2. Воздействие (Act) на объект.
3. Утверждение (Assert) об ожидаемом результате.

В приведенном ниже фрагменте кода присутствуют все три части, причем для утверждения используется класс `Assert` из каркаса NUnit:

```
[Test]
public void IsValidFileName_BadExtension_ReturnsFalse()
{
  LogAnalyzer analyzer = new LogAnalyzer();

  bool result = analyzer.IsValidLogFileName("filewithbadextension.foo");

  Assert.False(result);
}
```

Прежде чем двигаться дальше, необходимо узнать кое-что о классе `Assert`, поскольку это важный компонент автономных тестов.

2.4.1. Класс Assert

Класс `Assert` состоит из статических методов и находится в пространстве имен NUnit.Framework. Это мост между вашим кодом и каркасом NUnit, а его назначение – объявить о том, что должно выполняться некоторое предположение. Если переданные классу `Assert` аргументы отличаются от ожидаемых, то NUnit считает, что тест не прошел и выдает соответствующее уведомление. При желании можно задать сообщение, которое должно выдаваться в случае, когда утверждение оказалось ложно.

В классе `Assert` много методов, но основным является метод `Assert.True (булево выражение)`, который проверяет, выполнено ли некоторое булево условие. Существует и много других методов, играющих роль синтаксической глазури, которая проясняет смысл утверждений (например, использованный выше метод `Assert.False`).

Следующий метод проверяет, что фактический объект (или значение) совпадает с ожидаемым:

```
Assert.AreEqual(expectedObject, actualObject, message);
```

Например:

```
Assert.AreEqual(2, 1+1, "Арифметическая ошибка");
```

А вот метод, который проверяет, что оба аргумента ссылаются на один и тот же объект:

```
Assert.AreSame(expectedObject, actualObject, message);
```

Например:

```
Assert.AreSame(int.Parse("1"),int.Parse("1"),
  "это тест не должен пройти")
```

Синтаксис `Assert` просто понять, запомнить и использовать.

Отметим также, что все методы утверждений в качестве последнего параметра типа `string` принимают сообщение, отображаемое в дополнение к тому, что выводит каркас в случае отказа теста. Умоляю вас, *никогда* не пользуйтесь этим параметром (он необязателен). Просто называйте тесты так, чтобы из самого имени было понятно, что должно произойти. Часто разработчики задают тривиальные сообщения типа «тест не прошел» или «ожидалось *x*, а не *y*», хотя каркас уже и так предоставляет эту информацию. Тут дело обстоит так же, как с комментариями в коде: если вы вынуждены использовать этот параметр, значит, следовало бы придумать более подходящее имя метода.

Познакомившись с основами API, прогоним тест.

2.4.2. Прогон первого теста в NUnit

Настало время прогнать первый тест и посмотреть, пройдет ли он.

Существует по меньшей мере четыре способа прогнать тест:

- с помощью пользовательского интерфейса NUnit;
- с помощью исполнителя тестов в Visual Studio 2012 с расширением NUnit Runner, которое в каталоге NUget называется NUnit Test Adapter;
- с помощью исполнителя тестов ReSharper (хорошо известного коммерческого подключаемого модуля для VS);
- с помощью исполнителя тестов TestDriven.NET (еще одного хорошо известного коммерческого подключаемого модуля для VS).

И хотя в этой книге рассказывается только о пользовательском интерфейсе NUnit, лично я предпочитаю NCrunch – быстрый автоматический исполнитель, который, однако, стоит денег (этот и другие инструменты описаны в приложении). Он отображает результаты в окне редактора Visual Studio. Я считаю, что этот исполнитель является органичным дополнением к методике разработки через тестирование в реальных проектах. Дополнительные сведения о нем можно найти на сайте www.ncrunch.net/.

Чтобы прогнать тест с помощью пользовательского интерфейса NUnit, нужно сначала построить сборку (в данном случае DLL-файл), которую можно передать NUnit для инспектирования. Построив проект, посмотрите, в каком каталоге была создана сборка.

Затем откройте пользовательский интерфейс NUnit. (Если вы устанавливали NUnit вручную, найдите значок на рабочем столе. Если

же пакет NUnit.Runners устанавливался через NuGet, то нужный EXE-файл находится в папке Packages корневого каталога решения.) Выполните команду **File → Open** (Файл → Открыть). Введите имя тестовой сборки. Слева появится ваш единственный тест и иерархия пространств имен и классов проекта (рис. 2.3). Нажмите кнопку **Run** для прогона тестов. Тесты автоматически группируются по пространству имен (сборке, имени типа), так что можно выбрать для прогона только тесты одного типа или из одного пространства имен. (Обычно прогоняются все тесты, чтобы информация в случае отказов была более полной.)

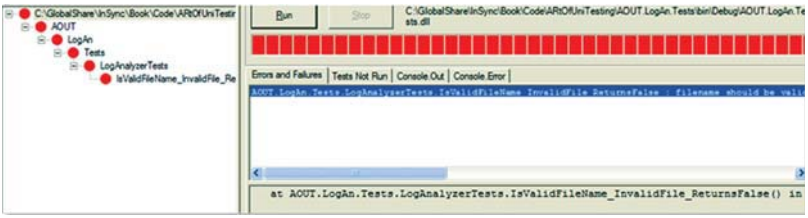


Рис. 2.3. Информация о том, что некоторые тесты не прошли, в NUnit показывается в трех местах: иерархия тестов слева и индикатор хода выполнения сверху становятся красными, а в область справа выводятся сообщения об ошибках

Наш тест не прошел, что может свидетельствовать о наличие ошибки в коде. Время исправить код и убедиться, что тест проходит. Добавьте недостающий `!` в условие `if`:

```
if(!fileName.EndsWith(".SLF"))
{
  return false;
}
```

2.4.3. Добавление положительных тестов

Мы видели, что метод распознает файлы с неправильным расширением, но кто сказал, что для файлов с правильным расширением он тоже ведет себя, как положено? Если бы мы вели разработку через тестирование, то было бы сразу понятно, что теста не хватает, но поскольку мы пишем тесты после кода, то приходится следить, чтобы были покрыты все пути. В листинге ниже мы добавили еще два метода, чтобы посмотреть, что происходит, когда передается имя файла с правильным расширением. В одном случае расширение состоит из прописных букв, в другом – из строчных.

Листинг 2.1. Тестирование логики проверки имени файла в классе LogAnalyzer

```
[Test] public void
IsValidLogFileName_GoodExtensionLowercase_ReturnsTrue()
{
  LogAnalyzer analyzer = new LogAnalyzer();
  bool result = analyzer
        .IsValidLogFileName("filewithgoodextension.slf");

  Assert.True(result);
}

[Test]public void IsValidLogFileName_GoodExtensionUppercase_ReturnsTrue()
{
  LogAnalyzer analyzer = new LogAnalyzer();

  bool result =
     analyzer
     .IsValidLogFileName("filewithgoodextension.SLF");

  Assert.True(result);
}
```

Если сейчас перестроить решение, то выяснится, что NUnit умеет обнаруживать изменение сборки и автоматически перезагружает ее в пользовательском интерфейсе. Еще раз прогнав тесты, мы увидим, что тест с расширением, записанным строчными буквами, не прошел. Необходимо исправить продуктовый код, воспользовавшись нечувствительным к регистру сравнением строк:

```
public bool IsValidLogFileName(string fileName)
{
  if (!fileName.EndsWith(".SLF",
        StringComparison.CurrentCultureIgnoreCase))
  {
     return false;
  }
  return true;
}
```

Теперь все тесты проходят, и в пользовательском интерфейсе NUnit снова отображается радующая глаз зеленая полоса.

2.4.4. От красного к зеленому: тесты должны проходить

В основе пользовательского интерфейса NUnit лежит простая идея: чтобы зажегся зеленый свет и можно было двигаться дальше, все тес-

ты должны пройти. Если хотя бы один тест не проходит, то полоса индикатора хода выполнения сверху становится красной в знак того, что с системой (или с тестами) не все в порядке.

Концепция красный–зеленый преобладает в мире автономного тестирования вообще и при разработке через тестирование в особенности. Мантра этой методики – «красный–зеленый–рефакторинг» – означает, что мы начинаем с теста, который не проходит, затем добиваемся, чтобы он прошел, а затем вносим в код изменения, делая его удобочитаемым и пригодным для сопровождения.

Тест может падать также из-за неожиданного исключения. Такие тесты считаются не прошедшими в большинстве каркасов тестирования – если не во всех. И это понятно – иногда ошибки принимают форму исключения, которого вы не ожидали.

И раз уж зашла речь об исключениях, то ниже в этой главе мы встретим тест, который ожидает, что код возбудит исключение, считая это правильным поведением. Такие тесты не проходят, если исключения не было.

2.4.5. Стилистическое оформление тестового кода

Обратите внимание, что все написанные мной тесты оформлены в стиле, отличающемся от «стандартного» кода. Имя теста может быть очень длинным, а подчерки помогают не забыть о включении всех важных элементов. Кроме того, части «подготовка», «действие» и «утверждение» отделены друг от друга пустой строкой. Это помогает мне гораздо быстрее читать тесты и находить в них ошибки.

Я также стараюсь как можно рельефнее отделить утверждение от действия. Я предпочитаю формулировать утверждение о значении, а не о результате вызова функции. Так код получается гораздо понятнее.

Удобочитаемость – одна из самых важных характеристик теста. Следует всемерно стремиться к тому, чтобы тест читался без усилий – даже человеком, который раньше его никогда не видел, – чтобы не возникало слишком много вопросов, а лучше – чтобы их вообще не возникало. Мы еще вернемся к этой теме в главе 8. Теперь посмотрим, нельзя ли уменьшить количество повторов в этих тестах, сделав их более лаконичными, но все же удобочитаемыми.

2.5. Рефакторинг – параметризованные тесты

Всем написанным выше тестам свойственны некоторые проблемы с удобством сопровождения. Представьте, что в конструктор класса `LogAnalyzer` добавлен параметр. Теперь все три теста не откомпилируются. Исправление трех тестов, быть может, и не такая большая проблема, но что, если их 30 или 100? В реальных проектах разработчикам есть чем заняться и кроме выспрашивания у компилятора, куда внести изменения. Если из-за тестов оказывается под угрозой текущий забег[1], то вы, скорее всего, не станете их запускать или вообще удалите те, что мешаются.

Переработаем тесты, так чтобы с этой проблемой никогда не сталкиваться.

В NUnit есть «крутая фишка», которая может помочь в этом деле, – параметризованные тесты. Чтобы ей воспользоваться, нужно просто взять один из уже имеющихся тестов, похожий на все остальные, и проделать следующее.

1. Заменить атрибут `[Test]` атрибутом `[TestCase]`.
2. Сделать все зашитые в тест значения параметрами тестового метода.
3. Поместить все параметры, выявленные на предыдущем шаге, внутрь квадратных скобок в атрибуте `[TestCase(param1, param2,...)]`.
4. Придумать для теста более общее имя.
5. Добавить к этому методу атрибут `[TestCase(...)]` для каждого теста, объединяемого в один метод, указывая в качестве параметров значения, присутствующие в этих тестах.
6. Удалить тесты, для которых на шаге 5 добавлен атрибут, оставив единственный метод с несколькими атрибутами `[TestCase]`.

Выполним эти действия шаг за шагом. Последний тест после шага 4 будет выглядеть так:

```
[TestCase("filewithgoodextension.SLF")]
public void
IsValidLogFileName_ValidExtensions_ReturnsTrue(string file)
{
  LogAnalyzer analyzer = new LogAnalyzer();
```

Атрибут TestCase передает параметр методу в следующей строке

Параметр, с которым атрибуты TestCase могут связать значение

[1] Sprint (забег) — одна итерация разработки в гибких методологиях. – *Прим. перев.*

```
    bool result = analyzer.IsValidLogFileName(file);

    Assert.True(result);
}
```

Параметр используется обобщенным образом

Во время выполнения исполнитель тестов сопоставляет параметр, указанный в атрибуте `TestCase`, с первым параметром самого тестового метода. Количество параметров в тестовом методе и в атрибуте `TestCase` может быть произвольным.

А теперь самое интересное: у одного и того же тестового метода может быть *несколько* атрибутов `TestCase`. Поэтому после шага 6 тест будет выглядеть так:

Дополнительный атрибут означает наличие еще одного теста с другим значением, передаваемым в качестве того же параметра

```
[TestCase("filewithgoodextension.SLF")]
[TestCase("filewithgoodextension.slf")]
public void
IsValidLogFileName_ValidExtensions_ReturnsTrue(string file)
{
    LogAnalyzer analyzer = new LogAnalyzer();

    bool result = analyzer.IsValidLogFileName(file);

    Assert.True(result);
}
```

И теперь можно *удалить* предыдущий тестовый метод, в котором проверялось расширение, записанное строчными буквами, потому что он уже учтен с помощью атрибута `TestCase` текущего метода. Прогнав тесты, мы убедимся, что их количество не изменилось, но зато код стал более удобным для сопровождения и удобочитаемым.

Можно пойти еще дальше и включить отрицательный тест (в котором утверждение ожидает значения `false`) в текущий тестовый метод. Я покажу, как это делается, но предупреждаю, что получающийся метод, скорее всего, окажется малопонятным, потому что придется придумывать *еще более* общее имя. Поэтому считайте этот пример демонстрацией синтаксиса, но имейте в виду, что это слишком большой шаг пусть даже в правильном направлении, так как без внимательного изучения кода тесты становится труднее понять.

Вот как можно свести все тесты в этом классе в один – добавив в атрибут `TestCase` и в тестовый метод один еще один параметр и заменив утверждение на `Assert.AreEqual`:

```
[TestCase("filewithgoodextension.SLF",true)]
[TestCase("filewithgoodextension.slf",true)]
[TestCase("filewithbadextension.foo",false)]
```

Добавляем еще один параметр в атрибут `TestCase`

```
public void
IsValidLogFileName_VariousExtensions_ChecksThem(string file,
                bool expected)
{
  LogAnalyzer analyzer = new LogAnalyzer();

  bool result = analyzer.IsValidLogFileName(file);

  Assert.AreEqual(expected,result);
}
```

Добавляем соответствующий параметр в тестовый метод

Используем значение второго параметра

Один этот тестовый метод позволил избавиться от всех остальных методов в классе, однако имя теста стало таким общим, что уже трудно понять, в чем разница между допустимым и недопустимым. Это должно легко выводиться из передаваемых значений параметров, поэтому старайтесь, чтобы они отражали ваше намерение, будучи при этом максимально простыми и очевидными. Подробнее о критерии удобочитаемости мы будем говорить в главе 8.

Если же говорить о пригодности для сопровождения, то обратите внимание, что теперь имеется всего один вызов конструктора. Это лучше, но все же недостаточно хорошо, потому что невозможно же заменить одним гигантским параметризованным тестовым методом все вообще тесты. Подробнее об удобстве сопровождения речь пойдет ниже (точно, в главе 8 – да вы просто телепат).

Сейчас можно подвергнуть рефакторингу и продуктовый код – изменить вид условного предложения `if`. Можно было бы свести его к одному предложению `return`. Если вам такой стиль нравится, флаг в руки. Мне не нравится. Я предпочитаю показаться многословным, но не заставлять читателя напряженно думать, что означает код. Я не люблю чрезмерно заумный код, а предложения `return` с условными операторами меня раздражают. Но это не книга о проектировании, помните? Делайте, как хотите. А я отсылаю вас к книге Роберта Мартина (дядюшки Боба) о «чистом коде».

2.6. Другие атрибуты в NUnit

Поняв, как легко создавать автономные тесты, которые можно выполнить автоматически, посмотрим, как задать начальное состояние теста и как прибраться после его выполнения.

В жизненном цикле автономного теста есть несколько точек, которые мы хотели бы контролировать. Прогон теста – лишь одна из них, а еще существуют специальные методы подготовки, которые выполняются перед прогоном каждого теста.

2.6.1. Подготовка и очистка

При прогоне автономных тестов важно, чтобы все данные и объекты, оставшиеся после предыдущих тестов, уничтожались, и чтобы для каждого нового теста воссоздавалось такое окружение, как будто до него никакие тесты не запускались. Если состояние будет сохраняться, то может оказаться, что некий тест падает, когда выполняется сразу после какого-то другого теста, но проходит в остальных случаях. Поиск ошибок, вызванных зависимостями между тестами, может занять много времени, и я никому этого не пожелаю. Обеспечение полной независимости тестов – одна из рекомендаций, о которых я рассажу во второй части книги.

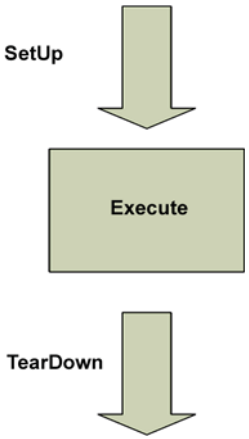


Рис. 2.4. NUnit производит подготовку и очистку соответственно до и после выполнения каждого метода

В NUnit существуют специальные атрибуты, которые упрощают контроль над подготовкой и очисткой состояния до и после тестов. Они называются `[SetUp]` и `[TearDown]`. На рис. 2.4 показан процесс выполнения теста с подготовкой и очисткой.

Пока что запомните, что любой написанный вами тест должен создавать новый экземпляр тестируемого класса, чтобы остаточное состояние не влияло на ход выполнения последующих тестов.

Чтобы взять на себя контроль над тем, что происходит во время подготовки и очистки, мы воспользуемся двумя атрибутами NUnit:

- `[SetUp]` – этот атрибут можно применить к методу, как и атрибут `[Test]`; помеченный так метод NUnit будет вызывать перед запуском любого теста в классе.
- `[TearDown]` – этим атрибутом помечается метод, который должен вызываться после выполнения любого теста в классе.

В листинге 2.2 показано, как с помощью атрибутов `[SetUp]` и `[TearDown]` можно гарантировать, что любой тест получит новый экземпляр `LogAnalyzer`, и при этом немного уменьшить дублирование.

Но имейте в виду, что чем активнее вы пользуетесь атрибутом `[SetUp]`, тем менее понятными становятся тесты, потому что читатель будет вынужден уделять внимание сразу двум местам в файле, чтобы

понять, как тест получает объекты для тестирования и какого типа эти объекты. Своим студентам я говорю: «Представьте, что читатель вашего теста никогда вас не видел и не увидит. Он столкнулся с вашими тестами спустя два года после того, как вы уволились. Каждая мелочь, которая поможет ему понять код, не задавая вопросов, будет высоко оценена. Очень может быть, что рядом не окажется никого, кто мог бы ответить на эти вопросы, так что вы – его единственная надежда». Заставлять читателя попеременно смотреть на разные участки кода, чтобы понять, как работает тест, – не самая лучшая мысль.

Листинг 2.2. Использование атрибутов `[SetUp]` и `[TearDown]`

```
using NUnit.Framework;

[TestFixture] public class LogAnalyzerTests
{
  private LogAnalyzer m_analyzer=null;

  [SetUp]                                          <-- Атрибут SetUp
  public void Setup()
  {
     m_analyzer = new LogAnalyzer();
  }

  [Test]
  public void IsValidFileName_validFileLowerCased_ReturnsTrue()
  {
     bool result = m_analyzer
            .IsValidLogFileName("whatever.slf");

     Assert.IsTrue(result, "имя файла должно быть правильным!");
  }

  [Test]
  public void IsValidFileName_validFileUpperCased_ReturnsTrue()
  {
     bool result = m_analyzer
            .IsValidLogFileName("whatever.SLF");

     Assert.IsTrue(result, "имя файла должно быть правильным!");
  }

  [TearDown]
  public void TearDown()                           <-- Атрибут TearDown
  {
     // Следующая строка включена для демонстрации антипаттерна.
     // На самом деле, она не нужна. Не поступайте так.
     m_analyzer = null;                            <-- Типичный антипаттерн – эта строка не нужна
```

```
    }
}
```

Методы подготовки и очистки можно рассматривать как конструкторы и деструкторы тестов в данном классе. В любом тестовом классе может быто только по одному методу каждого вида, и они будут выполняться для каждого тестового метода. В листинге 2.2 имеется два автономных теста, поэтому поток выполнения в NUnit будет таким, как показано на рис. 2.5.

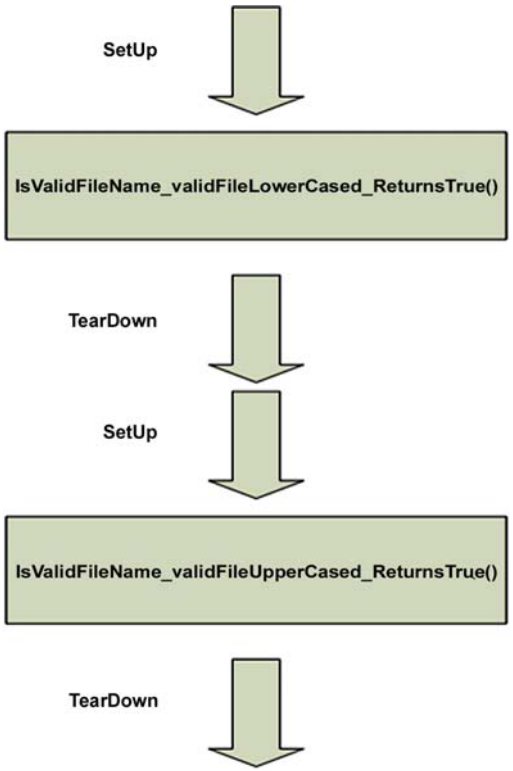


Рис. 2.5. Как NUnit вызывает методы с атрибутами `SetUp` и `TearDown` при наличии нескольких тестов в одном классе: до начала выполнения каждого теста вызывается `SetUp`, а после его завершения – `TearDown`

На практике я *не* пользуюсь методами подготовки для инициализации экземпляров. Я описал этот способ, только для того чтобы вы знали о его существовании и избегали его. На первый взгляд, идея

прекрасная, но очень скоро тесты, расположенные после метода подготовки, становится трудно читать. Поэтому для инициализации тестируемых экземпляров я пользуюсь фабричными методами. О них я расскажу в главе 7.

В NUnit имеется еще несколько атрибутов для подготовки и очистки состояния. Например, атрибуты `[TestFixtureSetUp]` и `[TestFixtureTearDown]` позволяют однократно инициализировать состояние до выполнения всех тестов в одном прогоне *класса* и после завершения прогона в целом (один раз на фикстуру). Это полезно, когда подготовка или очистка занимают много времени, и желательно производить их только один раз для всех тестов в составе фикстуры. Но пользоваться этими атрибутами следует с осторожностью, иначе можно невзначай создать состояние, общее для всех тестов.

Я призываю вас *никогда* (или почти никогда) не пользоваться методами с атрибутами `TearDown` и `TestFixture` в автономных тестах. Поступая иначе, вы, скорее всего, пишете интеграционный тест, который обращается к файловой системе или к базе данных, так что после прогона нужно почистить диск или базу. На мой взгляд, единственный случай, когда применение атрибута `TearDown` в автономных тестах оправдано, – «сброс» состояния статической переменной или объекта-одиночки (синглтона) в памяти между тестами. Во всех остальных случаях речь, вероятно, идет об интеграционных тестах. Само по себе, это неплохо, но для интеграционных тестов следует завести отдельный проект.

Далее мы узнаем, как проверить, что код действительно возбуждает ожидаемое исключение.

2.6.2. Проверка ожидаемых исключений

При тестировании часто требуется проверить, что тестируемый метод возбуждает исключение в тех случаях, когда это необходимо.

Допустим, что наш метод должен возбуждать исключение `ArgumentException`, когда ему передается пустое имя файла. Если код не возбуждает исключение, то тест не должен пройти. Логика этого метода показана в следующем листинге.

Листинг 2.3. Подлежащий тестированию метод проверки имени файла в классе `LogAnalyzer`

```
public class LogAnalyzer
{
  public bool IsValidLogFileName(string fileName)
```

```
    {
        …
        if (string.IsNullOrEmpty(fileName))
        {
            throw new ArgumentException(
              "имя файла должно быть задано");
        }
        …
    }
}
```

Протестировать этот метод можно двумя способами. Начнем с того, которым пользоваться не следует, хотя он очень широко распространен и когда-то был единственным. В NUnit имеется специальный атрибут для проверки исключений: `[ExpectedException]`. Вот как мог бы выглядеть тест для проверки возникновения исключения:

```
[Test]
[ExpectedException(typeof(ArgumentException),
 ExpectedMessage = "имя файла должно быть задано")]
public void IsValidFileName_EmptyFileName_ThrowsException()
{
  m_analyzer.IsValidLogFileName(string.Empty);
}

private LogAnalyzer MakeAnalyzer()
{
  return new LogAnalyzer();
}
```

Отметим следующие важные особенности.

- Ожидаемое в исключении сообщение передается в виде параметра атрибута `[ExpectedException]`.
- В тесте нет вызова `Assert`. Утверждением является само наличие атрибута `[ExpectedException]`.
- Не имеет смысла анализировать булево значение, возвращаемое методом, т. к. предполагается, что метод возбудит исключение.

Безотносительно к этому примеру я забежал вперед и вынес в фабричный метод код создания экземпляра `LogAnalyzer`. Такого рода методы я использую во всех своих тестах, чтобы в процессе сопровождения конструктора не нужно было править много тестов.

Для метода, приведенного в листинге 2.3, этот тест должен пройти. Если бы метод *не* возбуждал исключение `ArgumentException` или сообщение в этом исключении отличалось бы от ожидаемого, то тест не прошел бы, а каркас напечатал бы соответствующее сообщение.

Так почему я сказал, что этим способом пользоваться не нужно? Потому что атрибут `ExpectedException` по существу означает, что исполнитель тестов должен погрузить весь этот метод в большой блок `try-catch` и считать тест успешным, если исключение не было перехвачено. Проблема в том, что мы не знаем, в *какой* строке было возбуждено исключение. Вполне может случиться, что в конструкторе имеется ошибка, из-за которой он возбуждает исключение, хотя не должен был бы, а тест при этом проходит! Таким образом, при использовании этого атрибут тест может лгать, а потому использовать его не стоит.

Взамен в NUnit сравнительно недавно появился новый метод `Assert.Catch<T>(delegate)`. И вот как можно переписать приведенный выше тест:

```
[Test]   <— Атрибут ExpectedException не нужен
public void IsValidFileName_EmptyFileName_Throws()
{
  LogAnalyzer la = MakeAnalyzer();

  var ex =Assert.Catch<Exception>(() => la.IsValidLogFileName(""));

  StringAssert.Contains("имя файла должно быть задано",
                        ex.Message);
}
```

Используется метод Assert.Catch

Используется объект Exception, возвращенный методом Assert.Catch

Как видим, изменений немало.

- Мы больше не используем атрибут `[ExpectedException]`.
- Мы используем метод `Assert.Catch` и лямбда-выражение без аргументов, в теле которого вызывается метод `la.IsValidLogFileName("")`.
- Если код внутри лямбда-выражения возбуждает исключение, то тест проходит. Если исключение возбуждает какая-то строка вне лямбда-выражения, то тест не проходит.
- Функция `Assert.Catch` возвращает объект исключения, которое было возбуждено внутри лямбда-выражения. Это позволяет впоследствии делать утверждения относительно сообщения в этом исключении.
- Мы используем входящий в состав NUnit класс `StringAssert`, с которым еще не встречались. Он содержит вспомогательные методы, упрощающие проверку различных условий, в которые входят строки.
- Мы не утверждаем (с помощью метода `Assert.AreEqual`), что строки должны буквально совпадать, а пользуемся методом

`StringAssert.Contains`. Сообщение должно *содержать* искомую строку. В результате тест будет проще сопровождать, потому что строки имеют обыкновение изменяться, когда добавляются новые функциональные возможности. Строки – это часть пользовательского интерфейса, поэтому в них могут быть разрывы, несущественная дополнительная информация и т. д. Если бы мы утверждали, что строки полностью совпадают, то тест пришлось бы изменять при каждом добавлении в начало или в конец строки чего-то, что нас в данном случае совершенно не интересует (например, дополнительных строк или форматирования).

Этот тест «солжет» вам с меньшей вероятностью, и я рекомендую использовать `Assert.Catch`, а не `[ExpectedException]`.

Есть также возможность воспользоваться текучим синтаксисом NUnit для проверки сообщения в исключении. Мне он не нравится, но это вопрос вкуса. Узнать о текучем синтаксисе вы можете на сайте NUnit.com.

2.6.3. Игнорирование тестов

Иногда складывается ситуация, когда некоторые тесты перестали работать, но исходный код все равно нужно поместить в основную ветвь системы управления версиями. В таких редких случаях (а они действительно должны быть редкими!) можно снабдить «сломавшиеся» (по причине ошибки в тесте, а не в самом коде) тесты атрибутом `[Ignore]`. Выглядит это так:

```
[Test]
[Ignore("в этом тесте имеется ошибка")]
public void IsValidFileName_ValidFile_ReturnsTrue()
{
  /// ...
}
```

При прогоне этого теста в NUnit получится результат, показанный на рис. 2.6.

А что, если мы захотим прогонять тесты, сгруппировав их не по пространству имен, а по какому-то другому критерию? Тут в игру вступают категории тестов. Что это такое, я объясню в разделе 2.6.5.

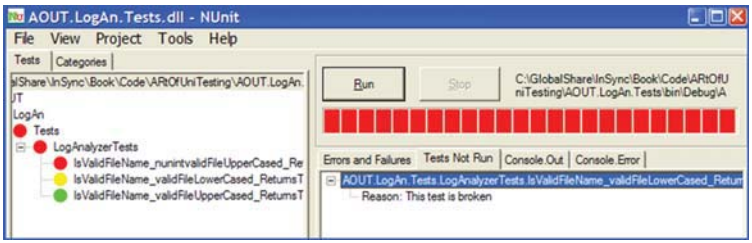


Рис. 2.6. В NUnit игнорируемый тест обозначается желтым цветом (средний тест), а причина, по которой он не выполняется, отображается на вкладке Tests Not Run в правой области

2.6.4. Текучий синтаксис в NUnit

В NUnit имеется альтернативный текучий синтаксис, которым можно пользоваться вместо простых методов `Assert.*`. Текучее выражение всегда начинается с вызова `Assert.That(..)`.

Вот как выглядит предыдущий тест, будучи записан в текучем синтаксисе:

```
[Test]
public void IsValidFileName_EmptyFileName_ThrowsFluent()
{
  LogAnalyzer la = MakeAnalyzer();

  var ex = Assert.Catch<ArgumentException>(() =>
              la.IsValidLogFileName(""));

  Assert.That(ex.Message,
       Is.StringContaining("имя файла должно быть задано"));
}
```

Лично мне нравится более лаконичный и простой синтаксис `Assert.something()`, а не `Assert.That`. И хотя текучий синтаксис на первый взгляд кажется более ясным, требуется больше времени, чтобы понять, что же все-таки тестируется (т. к. эта информация находится в конце строки). Выбирайте, что вам больше по душе, только придерживайтесь принятого решения во всем тестовом проекте, потому что разнобой сильно вредит удобочитаемости.

2.6.5. Задание категорий теста

Можно распределить тесты по категориям, например, медленные и быстрые. Для этого предназначен атрибут NUnit `[Category]`:

```
[Test]
[Category("Быстрые тесты")]
public void IsValidFileName_ValidFile_ReturnsTrue()
{
  /// ...
}
```

Снова загрузив тестовую сборку в NUnit, вы увидите, что тесты организованы по категориям, а не по пространствам имен. Перейдите на вкладку **Categories** и дважды щелкните по категории, тесты из которой хотите прогнать; ее название появится снизу в области **Selected Categories**. Затем нажмите кнопку **Run**. На рис. 2.7 показано, как выглядит экран после перехода на вкладку **Categories**.

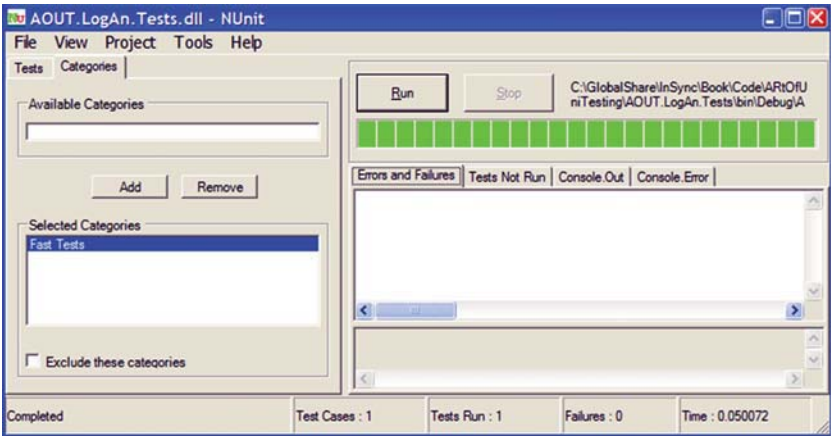


Рис. 2.7. Мы можем распределить тесты по категориям, а затем в пользовательском интерфейсе NUnit указать, из какой категории прогонять тесты

До сих пор мы прогоняли простые тесты методов, возвращающих значение в качестве результата своей работы. Но что, если метод не возвращает значение, а изменяет состояние объекта?

2.7. Проверка изменения состояния системы, а не возвращаемого значения

Пока что рассматривалась проверка простейшего вида результата, производимого единицей работы: возвращаемого значения (см. гла-

ву 1). Здесь и в следующей главе мы обсудим результат второго вида: изменение состояния системы; требуется проверить, что поведение системы изменилось после выполнения действия, указанного в тесте.

Определение. *Тестирование по состоянию* (state-based testing) (называемое также *верификацией состояния*) устанавливает правильность работы метода путем исследования изменившегося состояния тестируемой системы и взаимодействующих с ней компонентов (зависимостей) после выполнения метода.

Если система ведет себя в точности, как раньше, то либо ее состояние не изменилось, либо имеется ошибка.

Если вам встречалось определение тестирования по состоянию в других местах, то вы, наверное, обратили внимание, что я определяю это понятие иначе. Дело в том, что я рассматриваю этот вопрос с несколько иной точки зрения – удобства сопровождения тестов. Непосредственную проверку состояния (которое иногда делается доступным извне, чтобы его можно было протестировать) я обычно не одобряю, потому что это ведет к неудобным для сопровождения и малопонятным тестам.

Рассмотрим простой пример тестирования по состоянию на основе классе `LogAnalyzer`, когда для тестирования недостаточно вызвать какой-то один метод класса. В листинге 2.4 приведен код самого класса. Мы ввели новое свойство `WasLastFileNameValid`, в котором будем хранить результат последнего обращения к методу `IsValidLogFileName`. Напомню, что я привожу код сначала, потому что учу вас не разработке через тестирование, а умению писать хорошие тесты. Путем применения TDD можно было бы создать более качественные тесты, но это следующий шаг после того, как вы научитесь писать тесты *после* кода.

Листинг 2.4. Изменение значения свойства при вызове метода `IsValidLogFileName`

```
public class LogAnalyzer
{
  public bool WasLastFileNameValid { get; set; }

  public bool IsValidLogFileName(string fileName)
  {
     WasLastFileNameValid = false;          <-- Состояние системы изменяется

     if (string.IsNullOrEmpty(fileName))
```

```
        {
            throw new ArgumentException("имя файла должно быть задано");
        }
        if (!fileName.EndsWith(".SLF",
            StringComparison.CurrentCultureIgnoreCase))
        {
            return false;
        }

        WasLastFileNameValid = true;
        return true;
    }
}
```

Состояние системы изменяется

Как видим, `LogAnalyzer` запоминает результат последней проверки. Поскольку значение свойства `WasLastFileNameValid` зависит от предварительного вызова другого метода, мы не можем проверить эту функциональность на основе анализа возвращаемого значения какого-то метода. Нужны другие средства.

Прежде всего, необходимо решить, какую единицу работы мы тестируем. Новое свойство `WasLastFileNameValid`? Да, отчасти. Но также и метод `IsValidLogFileName`, поэтому имя теста должно начинаться с имени метода, так как именно эту единицу работы мы вызываем из открытого интерфейса, чтобы изменить состояние системы. В следующем листинге показан тест, проверяющий, что результат действительно запомнен.

Листинг 2.5. Тестирование класса путем вызова метода и последующей проверки значения свойства

```
[Test]
public void
IsValidFileName_WhenCalled_ChangesWasLastFileNameValid()
{
  LogAnalyzer la = MakeAnalyzer();

  la.IsValidLogFileName("badname.foo");

  Assert.False(la.WasLastFileNameValid);
}
```

Утверждение о состоянии системы

Отметим, что функциональность метода `IsValidLogFileName` тестируется с помощью утверждения, относящегося к другой части тестируемого класса.

Ниже приведен переработанный пример, в который добавлен еще один тест, где ожидается прямо противоположное состояние системы.

```
[TestCase("badfile.foo", false)]
[TestCase("goodfile.slf", true)]
public void
IsValidFileName_WhenCalled_ChangesWasLastFileNameValid(string file,
     bool expected)
{
  LogAnalyzer la = MakeAnalyzer();

  la.IsValidLogFileName(file);

  Assert.AreEqual(expected, la.WasLastFileNameValid);
}
```

В следующем листинге приведен пример иного рода. Здесь исследуется функциональность калькулятора в памяти.

Листинг 2.6. Методы `Add()` и `Sum()`

```
public class MemCalculator
{
  private int sum=0;

  public void Add(int number)
  {
     sum+=number;
  }

  public int Sum()
  {
     int temp = sum;
     sum = 0;
     return temp;
  }
}
```

Класс `MemCalculator` работает так же, как обычный карманный калькулятор. Нужно ввести число, нажать кнопку **Add**, набрать следующее число, снова нажать **Add** и т. д. По завершении нажимаем кнопку **Equals** и получаем текущую сумму.

С чего начать тестирование метода `Sum()`? Начинать всегда нужно с простейшего теста, например, с проверки того, что по умолчанию `Sum()` возвращает `0`. Такой тест показан в листинге ниже.

Листинг 2.7. Простейший тест метода калькулятора `Sum()`

```
[Test]
public void Sum_ByDefault_ReturnsZero()
{
```

```
    MemCalculator calc = new MemCalculator();

    int lastSum = calc.Sum();

    Assert.AreEqual(0,lastSum);
}
```

Утверждение о значении, возвращаемом по умолчанию

Снова отметим важность имени метода. Оно читается, как связное предложение.

Ниже приведен перечень соглашений об именовании, которые я предпочитаю использовать в подобных случаях.

- `ByDefault` используется, когда ожидается возврат определенного значения без каких-либо предварительных действий (как в примере выше).
- `WhenCalled` или `Always` используется для проверки результатов единицы работы второго или третьего вида (изменение состояния или обращение к третьей стороне) в случае, когда изменение состояния производится без предварительной инициализации или третья сторона вызывается без предварительного конфигурирования, например: `Sum_WhenCalled_CallsTheLogger` или `Sum_Always_CallsTheLogger`.

Больше никаких тестов без вызова метода `Add()` написать нельзя, поэтому в следующем тесте мы вызовем `Add()` и сделаем утверждение относительно значения, возвращаемого `Sum()`.

Листинг 2.8. Два теста, во втором вызывается метод `Add()`

```
[Test]
public void Sum_ByDefault_ReturnsZero()
{
  MemCalculator calc = MakeCalc();

  int lastSum = calc.Sum();

  Assert.AreEqual(0, lastSum);
}

[Test]
public void Add_WhenCalled_ChangesSum()
{
  MemCalculator calc = MakeCalc();

  calc.Add(1);
  int sum = calc.Sum();

  Assert.AreEqual(1, sum);
```

Поведение и состояние системы можно считать изменившимися, если метод `Sum()` возвращает не то же число, что раньше

```
}

private static MemCalculator MakeCalc()
{
  return new MemCalculator();
}
```

Обратите внимание, что на этот раз для инициализации `MemCalculator` мы воспользовались фабричным методом. Это правильно, потому что экономит время на написании тестов, уменьшает размер каждого теста и делает код более понятным, а также гарантирует единообразную инициализацию `MemCalculator`. Это также лучше с точки зрения удобства сопровождения, потому что если конструктор `MemCalculator` изменится, нам нужно будет изменить инициализацию только в одном месте, а не модифицировать вызов `new` в каждом тесте.

Пока все хорошо. Но что, если тестируемый метод зависит от внешнего ресурса, например, файловой системы, базы данных, веб-службы или еще чего-то, что трудно контролировать? И как организовать тестирование единицы работы третьего вида – обращение к стороннему компоненту? Понадобятся тестовые заглушки, поддельные и подставные объекты. Все это мы будем обсуждать в нескольких следующих главах.

2.8. Резюме

В этой главе мы рассмотрели, как использовать NUnit для написания простых тестов простого кода. Мы узнали об атрибутах `[TestCase]`, `[SetUp]` и `[TearDown]`, которые гарантируют, что в каждом тесте состояние объекта инициализируется заново. Чтобы сделать тесты более удобными для сопровождения, мы пользовались фабричными методами. Для пропуска тестов, нуждающихся в исправлении, мы указывали атрибут `[Ignore]`. Категории позволяют создавать логические группы тестов, а не просто объединять их по классу и пространству имен. Метод `Assert.Catch()` дает возможность надежно проверить, что код возбуждает исключения именно тогда, когда должен. Мы рассмотрели также ситуацию, когда проверять нужно не просто значение, возвращенное методом, а конечное состояние объекта.

Но всего этого недостаточно. В тестах по большей части приходится иметь дело с куда более сложным кодом. В следующих двух главах мы познакомимся с дополнительными инструментами написания ав-

тономных тестов. Сталкиваясь с различными непростыми сценариями, вы должны будете выбирать подходящий инструмент.

И напоследок повторим несколько важных моментов.

- Общепринято создавать по одному тестовому классу на каждый тестируемый, по одному проекту автономных тестов на каждый тестируемый проект (для интеграционных тестов создается отдельный проект) и по крайней мере по одному тестовому методу на каждую единицу работы (которая может состоять как из одного-единственного метода, так и из нескольких классов).
- Давайте тестам понятные имена, устроенные по образцу `[ЕдиницаРаботы]_[Сценарий]_[ОжидаемоеПоведение]`.
- Применяйте фабричные методы для повторного использования кода в тестах, например, для создания и инициализации объектов, необходимых всем тестам.
- Не используйте атрибуты `[SetUp]` и `[TearDown]`, если можете без них обойтись. Из-за них тесты становятся менее понятными.

В следующей главе мы будем рассматривать более реалистичные сценарии, где тестируемый код больше похож на настоящий. В реальном коде имеются зависимости, он не всегда тестопригоден, поэтому мы начнем знакомиться с заглушками, поддельными и подставными объектами и научимся использовать их при тестировании такого кода.

Часть II.

ОСНОВНЫЕ ПРИЕМЫ

Рассмотрев в предыдущих главах простейшие элементы, мы теперь познакомимся с основными приемами тестирования и рефакторинга, необходимыми для написания тестов в реальном мире.

В главе 3 мы начнем рассмотрение с изучения заглушек и того, как они помогают разрывать зависимости. Затем мы обсудим приемы рефакторинга, позволяющие сделать код более тестопригодным, и попутно расскажем о зазорах.

В главе 4 мы перейдем к подставным объектам и тестированию взаимодействий, узнаем, чем подставные объекты отличаются от заглушек, и познакомимся с идеей поддельных объектов.

Глава 5 посвящена изолирующим каркасам (их еще называют «каркасы подстановки» – mocking framework) и их применению к устранению некоторых видов дублирования кода в написанных вручную подставных объектах и заглушках. В главе 6 проводится сравнение наиболее популярных изолирующих каркасов для .NET и демонстрируются примеры применения каркаса FakeItEasy в типичных случаях.

ГЛАВА 3. Использование заглушек для разрыва зависимостей

В этой главе:

- Определение заглушки.
- Рефакторинг кода с целью использования заглушек.
- Добавляем текст и название.
- Решение проблемы инкапсуляции.
- Рекомендации по использованию заглушек.

В предыдущей главе мы написали первый автономный тест с применением NUnit и рассмотрели несколько тестовых атрибутов. Мы также подготовили тесты для простых случаев, когда нужно было лишь проверить возвращаемое значение метода или состояние тестируемой автономной единицы в тривиальной системе.

В этой главе мы рассмотрим более реалистичные примеры, когда тестируемый объект зависит от других объектов, которые мы не контролируем (или еще не реализовали). Таким объектом может быть веб-служба, текущее время, потоки и многое другое. Важно, что тест не может управлять тем, что возвращает зависимость и как она себя ведет (например, если вы захотите смоделировать исключение). Тутто и приходят на помощь *заглушки*.

3.1. Введение в заглушки

Полет человека в космос ставит ряд интересных задач перед инженерами и космонавтами, и одна из самых сложных – как удостовериться, что космонавт готов к полету и способен эксплуатировать всю технику, находясь на орбите. Для проведения полного *интеграционного*

теста космический челнок следовало бы запустить в космос, но, очевидно, что это не самый безопасный способ испытания космонавтов. Поэтому НАСА разработала полнофункциональные имитационные установки, копирующие окружение и панель управления космического корабля, чтобы устранить внешнюю зависимость от космического пространства.

Определение. *Внешней зависимостью* называется объект в системе, с которым тестируемый код взаимодействует, но над которым у него нет контроля (типичные примеры: файловая система, потоки, память, время и т. д.).

Управление внешними зависимостями из вашей программы и есть тема настоящей главы и большей части книги. В программировании для обхода проблемы внешних зависимостей применяются *заглушки*.

Определение. *Заглушкой* называется управляемая замена существующей зависимости (или *компаньона*) в системе. Используя заглушку, мы можем протестировать код, не взаимодействуя с зависимостью непосредственно.

В главе 4 мы более точно определим, что такое заглушки, подставные и поддельные объекты и как они взаимосвязаны. А пока запомните, что подставные объекты (mocks) – это те же заглушки, только о подставном объекте можно высказывать утверждения, а о заглушке – нет.

Рассмотрим пример и усложним класс `LogAnalyzer`, введенный в предыдущей главе. Нам предстоит устранить зависимость от файловой системы.

Названия паттернов тестирования

Классическим справочником по паттернам автономного тестирования является книга Gerard Meszaros «xUnit Test Patterns: Refactoring Test Code» (Addison-Wesley, 2007)[1]. В ней есть по меньшей мере пять определений сущностей, имитируемых в тестах, и мне кажется, что это только вносит путаницу (хотя все определения детализированы). В этой книге я использую только три определения имитируемых сущностей: поддельные объекты, или подделки (fakes), заглушки (stubs) и подставные объекты, или подставки (mocks)[2].

[1] Д. Мезарош «Шаблоны тестирования xUnit. Рефакторинг кода тестов». Вильямс, 2009. – *Прим. перев.*

[2] В русскоязычной литературе и в Интернете повсеместно встречаются «mock-объекты» и «fake-объекты» (см., например, https://code.google.com/p/googletest-translations/wiki/GoogleMockForDummiesRussian). Не считая возможным оставлять эти понятия без перевода, я взял за основу терминологию из весьма содержательной статьи по адресу http://club.shelek.ru/viewart.php?id=354. – *Прим. перев.*

Я полагаю, что при такой упрощенной терминологии читателю будет легче освоить паттерны, а для того чтобы начать писать отличные тесты, ничего больше знать и не нужно. Впрочем, в некоторых местах я буду ссылаться на названия паттернов, употребляющиеся в «xUnit Test Patterns», чтобы при желании вы могли найти определение, данное Мезарошем.

3.2. Выявление зависимости от файловой системы в LogAn

Класс `LogAnalyzer` можно настроить так, что он будет обрабатывать файлы журналов с разными расширениями, пользуясь адаптерами для каждого типа. Для простоты предположим, что допустимые расширения хранятся где-то на диске в конфигурационном файле приложения и что метод `IsValidLogFileName` выглядит следующим образом:

```
public bool IsValidLogFileName(string fileName)
{
   // читать конфигурационный файл
   // вернуть true, если конфигурация
   // говорит, что расширение
   // поддерживается
}
```

Беда (рис. 3.1) в том, что коль скоро тест зависит от файловой системы, то мы имеем интеграционный тест со всеми вытекающими проблемами: медленная работа, зависимость от конфигурации, тестирование сразу многих сторон и т. д.

В этом и состоит сущность *нетестопригодной* структуры: код зависит от некоторого внешнего ресурса, и эта зависимость может стать причиной отказа теста, когда логика кода совершенно корректна. В унаследованных системах одна единица работы (*действие* в системе) может зависеть от многочисленных внешних ресурсов, которые тест может контролировать в слабой степени или вообще не может. Тема унаследованного кода подробнее обсуждается в главе 10.

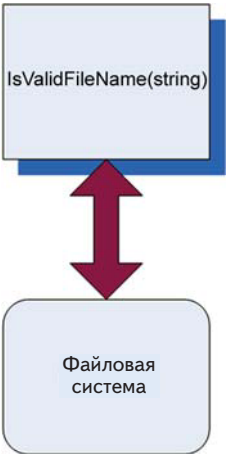


Рис. 3.1. Наш метод напрямую зависит от файловой системы. Структура тестируемой объектной модели не дает возможности протестировать его автономно, мы вынуждены перейти к интеграционному тестированию

3.3. Как можно легко протестировать LogAnalyzer

«Любую объектно-ориентированную задачу можно решить, добавив уровень косвенности, если, конечно, число уровней косвенности не слишком велико». Мне нравится эта цитата (из статьи в википедии по адресу http://en.wikipedia.org/wiki/Abstraction_layer), поскольку немалая доля «искусства» в искусстве автономного тестирования как раз и состоит в том, чтобы найти нужное место для добавления или использования уровня косвенности, позволяющее протестировать продуктовый код.

Не удается что-то протестировать? Добавьте уровень косвенности, обертывающий обращения к этому «чему-то», а затем имитируйте этот уровень в своих тестах. Или сделайте это «что-то» подменяемым (так что оно само представляет собой уровень косвенности). Искусство состоит также в том, чтобы понять, что нужный уровень косвенности уже существует, и не изобретать его заново, а также в том, чтобы вовремя воздержаться от использования косвенности, потому что она слишком усложняет дело. Но давайте двигаться шаг за шагом.

Единственный способ протестировать показанный выше код в таком виде – создать конфигурационный файл в файловой системе. Но мы хотим избежать подобных зависимостей и сделать код тестопригодным, не прибегая к интеграционному тестированию.

Обратившись к исходной аналогии с космонавтом, мы обнаруживаем четкий принцип разрыва зависимости.

1. Найти *интерфейс* или *API*, через который работает тестируемый объект. В случае космонавта это ручки управления и мониторы космического корабля, изображенные на рис. 3.2.
2. Заменить *истинную реализацию* этого интерфейса чем-то, что мы можем контролировать. Это означает, что различные мониторы, джойстики и кнопки следует подключить к пультовой, откуда инженеры-испытатели могут управлять тем, что интерфейс корабля показывает испытуемым космонавтам.

Для переноса этого принципа на наш код потребуются дополнительные шаги.

1. Найти *интерфейс*, через который работает начало тестируемой единицы работы. (В данном случае слово «интерфейс» употребляется не в строго объектно-ориентированном смысле, имеется в виду метод или класс, с которым взаимодейст-

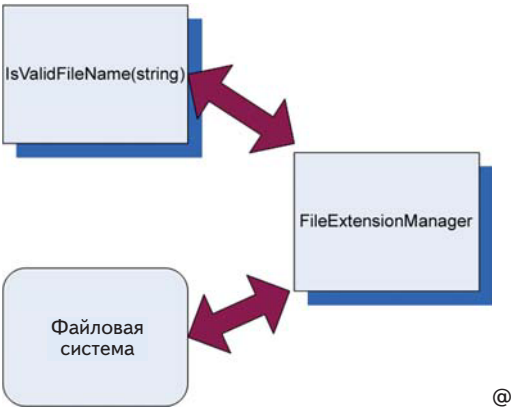
вует единица работы.) В проекте LogAn таковым является конфигурационный файл в файловой системе.

2. Если интерфейс соединен с тестируемой единицей работы *напрямую* (как в данном случае — мы непосредственно обращаемся к файловой системе), сделать код тестопригодным, добавив уровень косвенности, скрывающий интерфейс. В нашем примере это можно было бы сделать, переместив обращение к файловой системе в отдельный класс (например, `FileExtensionManager`). Позже мы рассмотрим и другие способы (на рис. 3.3 показано, как могла бы выглядеть структура программы после этого шага).
3. Заменить *истинную реализацию* интерфейса чем-то, что мы можем контролировать. В данном случае мы заменяем класс, к которому обращается метод (`FileExtensionManager`) классом-заглушкой, который можем контролировать (`StubExtensionManager`), и тем самым даем тесту возможность управлять внешними зависимостями.

Рис. 3.2. Имитатор космического челнока с реалистичными джойстиками и экранами, позволяющий моделировать внешний мир (фотография публикуется с разрешения НАСА)

Подменный класс вообще не обращается к файловой системе, т. е. мы разорвали зависимость от файловой системы. Но поскольку мы

тестируем не класс, общающийся с файловой системой, а код, который его вызывает, то вполне можно смириться с тем, что класс-заглушка не делает ничего полезного, а лишь довольно лепечет, когда к нему обращаются из теста. На рис. 3.4 показано, во что превратилась структура программы после такого изменения.



@

Рис. 3.3. Добавление уровня косвенности, чтобы избежать прямой зависимости от файловой системы. Код, обращающийся к файловой системе, вынесен в класс `FileExtensionManager`, который в тесте будет заменен заглушкой

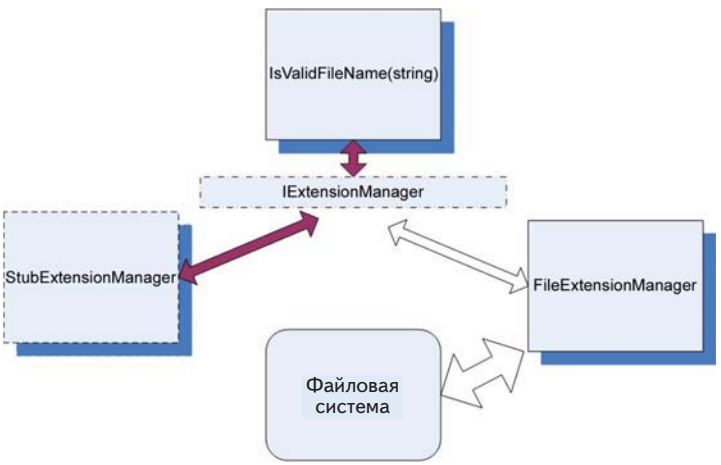


Рис. 3.4. Добавление заглушки для разрыва зависимости. Теперь классу ничего не нужно знать о том, с какой реализацией диспетчера расширений он работает

На рис. 3.4 я включил в общую картину интерфейс в смысле C#. Он позволяет объектной модели абстрагировать операции класса `FileExtensionManager`, а тесту – создать заглушку, которая выглядит в точности, как `FileExtensionManager`. В следующем разделе мы рассмотрим эту технику более детально.

Это был один из способов сделать продуктовый код тестопригодным – создание нового интерфейса. Теперь обсудим идею *рефакторинга* кода и включения в него *зазоров*.

3.4. Рефакторинг проекта с целью повышения тестопригодности

Настало время ввести еще два термина, которые будут постоянно встречаться в этой книге: *рефакторинг* и *зазоры*.

Определение. *Рефакторингом* называется операция изменения кода без изменения его функциональности. Иными словами, код делает в точности то же, что и раньше. Не больше и не меньше. Просто выглядит по-другому. В качестве примеров рефакторинга можно назвать переименование метода или разбиение одного длинного метода на несколько более коротких

Определение. *Зазорами* (seam) называются места программы, куда можно подключить иную функциональность взамен существующей, например: классы-заглушки, добавление параметра конструктора, добавление открытого свойства, допускающего установку, преобразование метода в виртуальный, чтобы его можно было переопределить, или вынесение наружу делегата в виде параметра или свойства, чтобы его можно было задавать вне класса. Зазор – это то, что получается в результате следования принципу открытости-закрытости, когда функциональность класса открыта для расширения, но его исходный код закрыт для прямой модификации (о зазорах см. книгу Michael Feathers «Working Effectively with Legacy Code»[3], а о принципе открытости-закрытости – книгу Роберта Мартина «Чистый код»).

Мы можем подвергнуть код рефакторингу, включив новый зазор без изменения исходной функциональности, что я и сделал, добавив новый интерфейс `IExtensionManager`.

И обязательно подвергнем.

Но прежде хочу напомнить, что бесшабашный рефакторинг кода *без* каких-нибудь автоматизированных тестов (интеграционных или иных) может завести вас в кроличью нору, которая станет концом ка-

[3] Майкл Физерс «Эффективная работа с унаследованным кодом». Вильямс, 2009. – *Прим. перев.*

рьеры. Прежде чем вносить любые изменения в существующий код, подстрахуйтесь интеграционными тестами или хотя бы составьте план отступления – сделайте сначала копию кода, предпочтительно в системе управления версиями, снабдив ее четко видным примечанием «до начала рефакторинга», чтобы потом легко найти. В этой главе я предполагаю, что какие-то интеграционные тесты у вас уже есть и что вы прогоняете их после любого рефакторинга, дабы убедиться, что программа все еще работает. Но больше мы к этому возвращаться не будем, потому что книга посвящена автономному тестированию.

Чтобы разорвать зависимости между тестируемым кодом и файловой системой, мы можем внести в код один или несколько *зазоров*. Нужно только помнить о том, что новый код должен делать в точности то же, что и раньше. Существует два взаимосвязанных типа рефакторинга для разрыва зависимостей. Я буду называть их рефакторинг типа А и рефакторинг типа Б.

- *Тип А* – абстрагирование конкретных объектов путем преобразования их в *интерфейсы* или *делегаты*.
- *Тип Б* – внедрение *поддельных реализаций* этих делегатов или интерфейсов.

В перечне ниже только первый способ является рефакторингом типа А, остальные – рефакторинг типа Б.

- *Тип А* – выделение интерфейса с целью подмены истинной реализации.
- *Тип Б* – внедрение реализации заглушки в тестируемый класс.
- *Тип Б* – внедрение подделки на уровне конструктора.
- *Тип Б* – внедрение подделки в виде чтения либо установки свойства.
- *Тип Б* – внедрение подделки непосредственно перед вызовом метода.

Рассмотрим все эти способы поочередно.

3.4.1. Выделение интерфейса с целью подмены истинной реализации

При таком подходе необходимо вынести весь код, относящийся к работе с файловой системой, в отдельный класс. Тогда его можно будет легко обособить и впоследствии подменить при обращении из тестов (как было показано на рис. 3.3). В следующем листинге показаны места кода, которые нужно изменить.

Листинг 3.1. Выделение класса, содержащего операции с файловой системой, и обращение к нему

```
public bool IsValidLogFileName(string fileName)
{
  FileExtensionManager mgr = new FileExtensionManager();
  return mgr.IsValid(fileName);
}

class FileExtensionManager
{
  public bool IsValid(string fileName)
  {
     // здесь читается файл
  }
}
```

Использование выделенного класса

Определение выделенного класса

Затем мы можем изменить тестируемый класс, так чтобы вместо конкретного класса `FileExtensionManager` использовалась какая-то форма `ExtensionManager`, о конкретной реализации которой класс ничего не знает. В .NET для этого применяется базовый класс или интерфейс, которому наследует `FileExtensionManager`.

В следующем листинге показано, как использовать новый интерфейс, чтобы сделать класс более тестопригодным. Структура такой реализации показана на рис. 3.4.

Листинг 3.2. Extracting an interface from a known class

```
public class FileExtensionManager : IExtensionManager
{
  public bool IsValid(string fileName)
  {
    ...
  }
}

public interface IExtensionManager
{
  bool IsValid (string fileName);
}

// тестируемая единица работы
public bool IsValidLogFileName(string fileName)
{
  IExtensionManager mgr = new FileExtensionManager();
  return mgr.IsValid(fileName);
}
```

Реализуем интерфейс

Определяем новый интерфейс

Определяем переменную типа интерфейса

Мы создаем интерфейс с одним методом `IsValid(string)`. Класс `FileExtensionManager` реализует этот интерфейс. Работает он так же, как и раньше, но теперь мы можем заменить «настоящий» `Manager` «поддельным», который ниже создадим для поддержки тестирования.

Мы еще не создали класс-заглушку для диспетчера расширений, так не будем откладывать в долгий ящик. Код показан в следующем листинге.

Листинг 3.3. Простой метод-заглушка, который всегда возвращает `true`

```
public class AlwaysValidFakeExtensionManager:IExtensionManager
{
  public bool IsValid(string fileName)
  {
     return true;
  }
}
```

Реализует IExtensionManager

Прежде всего, обратите внимание на имя класса. Оно очень важно. Не `StubExtensionManager` и не `MockExtensionManager`, а именно `FakeExtensionManager`. Слово *fake* (поддельный) означает, что объект выглядит, как другой объект, но может использоваться как *подставка* или *заглушка* (подставным объектам посвящена следующая глава).

Говоря, что объект или переменная является подделкой, мы откладываем решение о том, как назвать этот объект-имитацию, и избегаем путаницы, которая могла бы возникнуть, если бы мы назвали его диспетчером-подставкой или диспетчером-заглушкой.

Услышав слово «подставка» или «заглушка», человек ожидает определенное поведение, которое мы обсудим позже. Мы не хотим уточнять имя этого класса, потому что он создан таким образом, что может выступать в обеих ролях, чтобы в будущем его можно было использовать в разных тестах.

Этот поддельный диспетчер расширений всегда возвращает `true`, поэтому мы назвали класс `AlwaysValidFakeExtensionManager`, чтобы читатель будущего теста, даже не заглядывая в исходный код, понимал, какого поведения ожидать от поддельного объекта.

Это лишь одно из возможных решений, и оно может привести к лавинообразному росту числа таких «рукописных» подделок. «Рукописными» мы называем подделки, написанные вручную, без использования порождающего каркаса. Ниже в этой главе я покажу другой способ конфигурирования подделок.

Эту подделку можно и в таком виде использовать в тестах, устранив тем самым зависимость от файловой системы, но можно также добавить в нее код, который позволит имитировать возбуждение любого исключения. Об этом тоже чуть ниже.

Итак, у нас есть интерфейс и два реализующих его класса, однако тестируемый метод по-прежнему вызывает настоящую реализацию напрямую:

```
public bool IsValidLogFileName(string fileName)
{
  IExtensionManager mgr = new FileExtensionManager();
  return mgr. IsValid (fileName);
}
```

Нужно как-то сказать этому методу, чтобы он использовал нашу, а не исходную реализацию `IExtensionManager`. Необходимо ввести в код *зазор*, в который можно будет вставить нашу заглушку.

3.4.2. Внедрение зависимости: внедрение поддельной реализации в тестируемую единицу работы

Существует несколько проверенных способов создания зазоров на основе интерфейсов – мест, куда можно внедрить реализацию интерфейса, чтобы ей пользовались методы класса. Перечислим наиболее распространенные:

- получить интерфейс в конструкторе и сохранить его в поле для последующего использования;
- получить интерфейс в свойстве и сохранить его в поле для последующего использования;
- получить интерфейс непосредственно перед вызовом в тестируемом методе одним из следующих способов:
  - в виде параметра метода (*внедрение через параметр*);
  - с помощью фабричного класса;
  - с помощью локального фабричного метода;
  - варианты вышеупомянутых способов.

Метод внедрения через параметр тривиален: передаем экземпляр (поддельной) зависимости рассматриваемому методу, добавив в его сигнатуру дополнительный параметр.

Рассмотрим остальные решения и подумаем, когда какое применять.

3.4.3. Внедрение подделки на уровне конструктора (внедрение через конструктор)

В этом случае мы добавляем новый конструктор (или параметр в существующий конструктор), который будет принимать объект типа созданного ранее интерфейса (`IExtensionManager`). Этот конструктор инициализирует поле класса типа интерфейса, которое впоследствии сможет использовать наш или любой другой метод. На рис. 3.5 показана схема внедрения заглушки.

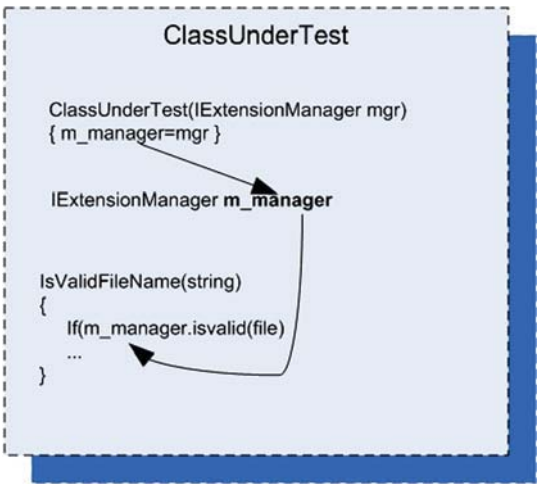


Рис. 3.5. Схема внедрения через конструктор

В листинге ниже показано, как можно было бы написать тест класса `LogAnalyzer` с применением техники внедрения через конструктор.

Листинг 3.4. Внедрение заглушки через конструктор

```
public class LogAnalyzer                                  <-- Продуктовый код
{
  private IExtensionManager manager;

  public LogAnalyzer(IExtensionManager mgr)               <-- Определяем конструктор, который можно вызывать из тестов
  {
     manager = mgr;
  }

  public bool IsValidLogFileName(string fileName)
```

```
    {
        return manager.IsValid(fileName);
    }
}

public interface IExtensionManager
{
    bool IsValid(string fileName);
}

[TestFixture]                                    ← Тест
public class LogAnalyzerTests
{
    [Test]
    public void
    IsValidFileName_NameSupportedExtension_ReturnsTrue()
    {
        FakeExtensionManager myFakeManager =
                    new FakeExtensionManager();
        myFakeManager.WillBeValid = true;

        LogAnalyzer log =
            new LogAnalyzer (myFakeManager);

        bool result = log.IsValidLogFileName("short.ext");
        Assert.True(result);
    }
}

internal class FakeExtensionManager : IExtensionManager
{
    public bool WillBeValid = false;

    public bool IsValid(string fileName)
    {
      return WillBeValid;
    }
}
```

Настраиваем заглушку, так чтобы она возвращала true

Передаем заглушку

Определяем заглушку самым простым способом

Примечание. Поддельный диспетчер расширений находится в том же файле, что тестовый код, потому что в данный момент подделка используется только в этом тестовом классе. Если рукописная подделка находится в том же файле, то найти, прочитать и сопровождать ее будет гораздо проще. Но если впоследствии эта подделка понадобится еще в каком-то классе, то мы сможем вынести ее в отдельный файл с помощью инструмента типа ReSharper (который я горячо рекомендую). См. приложение.

Отметим также, что поддельный объект в листинге 3.4 отличается от того, что мы видели раньше. В тесте его можно настроить, указав,

какое булево значение следует возвращать при вызове метода. Благодаря возможности настройки мы сможем использовать класс заглушки в нескольких тестах – тесту нужно будет сконфигурировать заглушку, перед тем как передавать тестируемому объекту. Это также делает код теста более понятным, потому что читатель находит все необходимое в одном месте. Удобочитаемость – важная характеристика автономных тестов, о которой мы будем подробно говорить позже, особенно в главе 8.

Еще следует отметить, что, используя параметры конструктора, мы по существу намеренно делаем их обязательными зависимостями (в предположении, что это единственный конструктор). Пользователь этого типа будет обязан задать в аргументах все необходимые ему конкретные зависимости.

Недостатки внедрения через конструктор

Использование конструкторов для внедрения зависимостей иногда чревато неприятностями. Если тестируемому коду для корректной работы без зависимостей необходимо несколько заглушек, то добавление все новых и новых конструкторов (или новых параметров конструктора) может стать проблемой. Код даже может оказаться неудобочитаемым или плохо сопровождаемым.

Допустим, что класс `LogAnalyzer` зависит не только от диспетчера расширений файлов, но еще от веб-службы и службы протоколирования. Тогда конструктор может выглядеть следующим образом:

```
public LogAnalyzer(IExtensionManager mgr, ILog logger, IWebService service)
{
  // этот конструктор будет вызываться из тестов
  manager = mgr;
  log= logger;
  svc= service;
}
```

Для решения проблемы можно было бы создать специальный класс, содержащий все необходимое для инициализации данного класса и передавать конструктору только один параметр: объект этого класса. Этот подход иногда называют *рефакторингом параметрического объекта*. Правда, если у такого объекта окажется несколько десятков свойств, то целесообразность решения будет сомнительна, но тем не менее так поступить можно.

Другое решение – воспользоваться контейнерами инверсии управления (inversion of control – IoC). IoC-контейнер можно рассматри-

вать как «умную фабрику» объектов (хотя это далеко не все, на что они способны). Из популярных контейнеров такого рода упомянем Microsoft Unity, StructureMap и Castle Windsor. Они предоставляют специальные фабричные методы, которые принимают тип создаваемого объекта и все необходимые ему зависимости, после чего инициализируют объект, применяя специальные настраиваемые правила, например: какой конструктор вызывать, какие свойства устанавливать и в каком порядке и т. д. Они умеют работать со сложными иерархиями объектов, когда для создания одного объекта требуется создать и инициализировать несколько других, лежащих ниже него в иерархии. Например, если конструктору нашего объекта необходимо передать интерфейс `ILogger`, то контейнер можно настроить так, что, пытаясь удовлетворить это требование, он будет возвращать один и тот же объект типа `ILogger`. Как правило, применение контейнеров упрощает порождение нужных объектов и причиняет меньше хлопот в части зависимостей и сопровождения конструкторов.

Совет. Существуют и другие удачные реализации контейнеров, например Autofac или Ninject; обратите на них внимание, если захотите еще что-то почитать на эту тему. Работа с контейнерами выходит за рамки этой книги, но начать их изучение стоит со списка, который Скотт Ханселманн опубликовал на странице www.hanselman.com/blog/ListOfNETDependencyInjectionContainersIOC.aspx. Если же вы желаете по-настоящему глубоко исследовать этот вопрос, то я рекомендую книгу Mark Seeman «Dependency Injection in .NET»[4] (Manning Publications, 2011). Прочитав ее, вы сможете создать собственный контейнер с нуля. Я редко пользуюсь контейнерами в реальном коде, так как считаю, что они обычно лишь усложняют структуру и делают код неудобочитаемым. Если вы испытываете нужду в контейнере, может быть, имеет смысл пересмотреть дизайн? Как вы думаете?

Когда использовать внедрение через конструктор

Мой опыт показывает, что использование аргументов конструктора для инициализации объектов может сделать тестовый код более громоздким, если не пользоваться для создания объектов вспомогательными каркасами типа IoC-контейнеров. Но все же я предпочитаю именно этот способ, потому что это наименьшее из зол в плане понятности и удобочитаемости API.

Кроме того, использование параметров конструктора – лучший способ дать понять пользователю API, что параметры обязательны. Их необходимо задать, чтобы получить объект.

[4] Марк Симан «Внедрение зависимостей в .NET». Питер, 2013. – *Прим. перев.*

Если вы хотите, чтобы зависимости были необязательными, обратитесь к разделу 3.4.5. Там обсуждаются методы чтения и установки свойств – способ задания необязательных зависимостей, не столь суровый, как, скажем, добавление в класс конструкторов для каждой зависимости.

Это книга не по проектированию и не по TDD. Тем, кто хочет понять, когда использовать параметры конструктора, еще раз рекомендую прочитать книгу «Чистый код» Боба Мартина – но либо после того, как вы уверенно овладеете автономным тестированием, либо еще до начала изучения автономного тестирования. Изучение сразу двух или более дисциплин, таких серьезных, как TDD, проектирование и автономное тестирование, только затруднит процесс обучения. Изучайте каждую тему отдельно – и тогда точно все усвоите.

Совет. Скоро вы поймете, что вопрос о том, какую технику или подход к проектированию выбрать в конкретной ситуации, возникает в мире автономного тестирования на каждом шагу. И это прекрасно. Всегда подвергайте сомнению свои предположения, так можно узнать что-то новое.

Если вы остановитесь на внедрении через конструктор, то, наверное, захотите использовать IoC-контейнер. Это решение было бы замечательно, если бы все программы в мире пользовались IoC-контейнерами, однако большинство людей ничего не знают о принципе инверсии управления, не говоря уже об инструментах, позволяющих воплотить его в жизнь. В будущем автономное тестирование, наверное, все чаще будет опираться на такие каркасы. И тогда появятся более ясные и четкие рекомендации по проектированию классов с зависимостями. Либо мы увидим инструменты, которые решают проблему внедрения зависимости вообще без конструкторов.

Как бы то ни было, параметры конструктора – лишь одно из возможных решений. Не менее часто используются и свойства.

3.4.4. Имитация исключений от подделок

Ниже приведен простой пример того, как сделать класс-подделку настраиваемым, чтобы вызванный метод мог возбуждать любое исключение. Для определенности предположим, что тестируется следующее требование: если диспетчер расширений файлов возбуждает исключение, то нужно вернуть `false` и воспрепятствовать дальнейшему распространению исключения (да, в реальной программе так делать не следует, но не ругайте меня – это всего лишь пример).

```
[Test]
public void
IsValidFileName_ExtManagerThrowsException_ReturnsFalse()
{
  FakeExtensionManager myFakeManager = new FakeExtensionManager();
  myFakeManager.WillThrow = new Exception("это подделка");

  LogAnalyzer log = new LogAnalyzer (myFakeManager);
  bool result = log.IsValidLogFileName("anything.anyextension");
  Assert.False(result);
}

internal class FakeExtensionManager : IExtensionManager {
  public bool WillBeValid = false;
  public Exception WillThrow = null ;

  public bool IsValid(string fileName)
  {
     if(WillThrow !=null) {
         throw WillThrow;
     }

     return WillBeValid;
  }
}
```

Чтобы этот тест прошел, мы должны написать код, который вызывает диспетчер расширений внутри блока `try-catch` и возвращает `false`, если мы попали в `catch`.

3.4.5. Внедрение подделки через установку свойства

В этом случае мы устанавливаем свойство с методами get и set для каждой зависимости, которую хотим внедрить. Затем тестируемый код использует эту зависимость, когда она ему понадобится. На рис. 3.6 показана схема внедрения с помощью свойств.

Тестовый код, в котором применяется эта техника (она тоже называется внедрением зависимости, как и все остальные описанные в этой главе приемы), будет выглядеть почти так же, как приведенный в разделе 3.4.3, где зависимость внедрялась через конструктор. Однако этот код (см. листинг ниже) проще писать и читать.

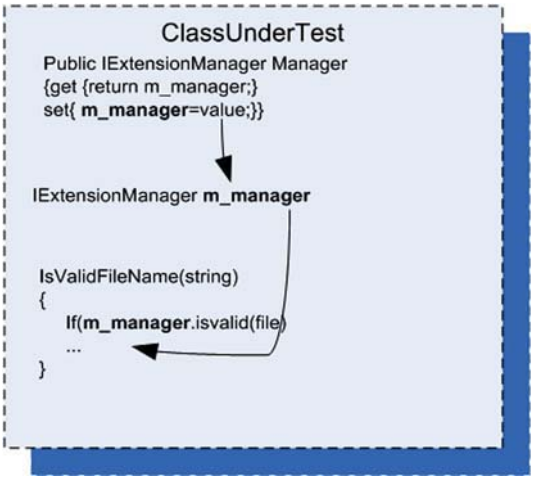


Рис. 3.6. Внедрение зависимостей с помощью свойств. Это гораздо проще, чем использовать конструктор, потому что каждый тест может устанавливать только те свойства, которые ему нужны

Листинг 3.5. Внедрение подделки путем добавления установщиков свойств в тестируемый класс

```
public class LogAnalyzer
{
  private IExtensionManager manager;

  public LogAnalyzer ()
  {
     manager = new FileExtensionManager();
  }

  public IExtensionManager ExtensionManager
  {
     get { return manager; }
     set { manager = value; }
  }

  public bool IsValidLogFileName(string fileName)
  {
     return manager.IsValid(fileName);
  }
}

[Test]
```

Допускается задание зависимости через свойство

```
public void IsValidFileName_SupportedExtension_ReturnsTrue()
{
  // задать используемую заглушку, настроить ее на возврат true
  ...

  // создать анализатор и внедрить заглушку
  LogAnalyzer log = new LogAnalyzer ();
  log.ExtensionManager=someFakeManagerCreatedEarlier;

  // сделать утверждение в предположении, что расширение поддерживается
  ...
}
```

Внедряем заглушку

Внедрение через свойства, как и через конструктор, влияет на дизайн API в плане решения, какие зависимости обязательны, а какие – нет. Использование свойств означает, что зависимость не обязательна для работы типа.

Когда использовать внедрение через свойство

Эта техника применяется, чтобы подчеркнуть, что зависимость тестируемого класса необязательна или может быть создана по умолчанию, не порождая проблем во время тестирования.

3.4.6. Внедрение подделки непосредственно перед вызовом метода

В этом разделе мы рассмотрим случай получения объекта непосредственно перед началом операций с ним, а не через конструктор или свойство. Разница в том, что запрос заглушки инициирует тестируемый код, тогда как в предыдущих разделах объект-подделка создавался вне тестируемого кода до начала тестирования.

Использование фабричного класса

В этом случае мы возвращаемся к исходному варианту, когда класс инициализирует диспетчер в своем конструкторе, однако экземпляр получаем от фабричного класса. Паттерн проектирования «Фабрика» (Factory) описывает ситуацию, когда за создание объектов отвечает отдельный класс.

В тестах необходимо будет сконфигурировать фабричный класс (в данном случае он будет содержать статический метод, который возвращает объект, реализующий интерфейс `IExtensionManager`), так чтобы он возвращал заглушку вместо настоящей реализации. Эта схема показана на рис. 3.7.

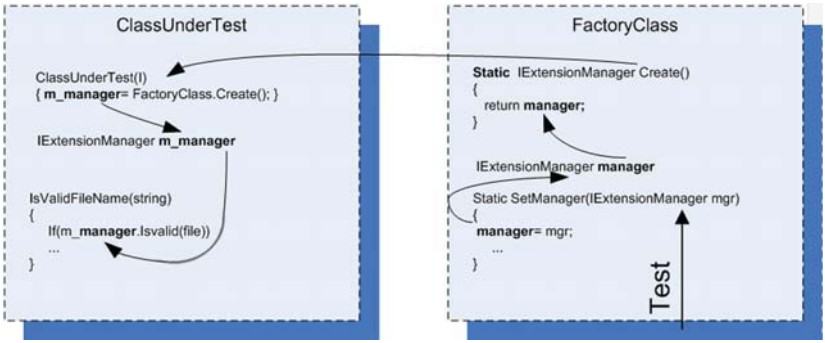


Рис. 3.7. Тест настраивает фабричный класс, так чтобы тот возвращал объект-заглушку. Тестируемый класс обращается к фабрике за экземпляром и в продуктовом коде получит настоящий объект, а не заглушку.

Дизайн на основе фабричных классов отличается чистотой, поэтому во многих объектно-ориентированных системах он применяется для порождения объектов. Но в большинстве систем никто, кроме самого фабричного класса, не может решать, какой экземпляр возвращать, чтобы не нарушать инкапсуляцию фабрики.

Однако в данном случае я добавил в фабричный класс метод установки (еще один зазор), чтобы у тестов было больше контроля над тем, что возвращает фабрика. Коль скоро в тестах появились статические данные, вероятно, понадобится восстанавливать начальное состояние фабрики до или после выполнения каждого теста, чтобы избежать влияния на последующие.

При таком подходе получается тест, код которого легко читать, а обязанности четко разделены между классами. Каждый отвечает за свое действие.

В следующем листинге показано использование фабричного класса в `LogAnalyzer` (тест тоже включен).

Листинг 3.6. Настройка фабричного класса, так чтобы тот возвращал заглушку при вызове из теста

```
public class LogAnalyzer
{
  private IExtensionManager manager;

  public LogAnalyzer ()
  {
     manager = ExtensionManagerFactory.Create();
```

Используем фабрику в продуктовом коде

```
    }

    public bool IsValidLogFileName(string fileName)
    {
       return manager.IsValid(fileName)
          && Path.GetFileNameWithoutExtension(fileName).Length>5;
    }
}

[Test]
public void IsValidFileName_SupportedExtension_ReturnsTrue()
{
    // задать используемую заглушку, настроить ее на возврат true
    ...
    ExtensionManagerFactory.SetManager(myFakeManager);

    // создать анализатор и внедрить заглушку
    LogAnalyzer log = new LogAnalyzer ();

    // сделать утверждение в предположении, что расширение поддерживается
    ...
}

class ExtensionManagerFactory
{
    private IExtensionManager customManager=null;

    public IExtensionManager Create()
    {
       if(customManager!=null)
         return customManager;
       return new FileExtensionManager();
    }

    public void SetManager(IExtensionManager mgr)
    {
       customManager = mgr;
    }
}
```

В этом тесте настраиваем фабрику на возврат заглушки

Определяем фабрику, которая умеет возвращать специальный диспетчер

Существует много различных реализаций фабрики, в этой книге показана лишь простейшая иллюстрация. О других примерах читайте главы, посвященные паттернам «Фабричный метод» и «Абстрактная фабрика», в классической книге «Design Patterns» «банды четырех» (Addison-Wesley, 1994) (Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides)[5].

[5] Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования». ДМК Пресс, Питер, 2013. – *Прим. перев.*

Единственное, о чем нужно помнить при использовании этих паттернов, – необходимость добавить зазор, чтобы фабрика могла возвращать заглушку вместо настоящей реализации. Во многих системах имеется глобальный переключатель `#debug` – если он включен, то в зазоры автоматически вставляются поддельные или тестопригодные объекты вместо реализаций по умолчанию. Соответствующая настройка может оказаться муторным делом, но затраты окупятся сторицей, когда придет время тестировать систему.

Сокрытие зазоров в выпускном режиме

Что, если вы не хотите показывать зазоры в выпускном режиме? Есть несколько способов решить эту задачу. Например, в .NET можно поместить относящийся к зазорам код (дополнительный конструктор, установщик свойства или настройку фабрики) в условно компилируемую секцию. Я расскажу об этом в разделе 3.6.2.

Различные уровни косвенности

В этом разделе мы имеем дело с более глубокой иерархией уровней, чем в предыдущих. На каждом уровне можно подделывать (или заглушать) различные объекты. В табл. 3.1 показаны три уровня программы, на которых можно возвращать заглушки.

Таблица 3.1. Уровни кода, на которых возможна подделка объектов

Тестируемый код	Возможное действие
Уровень 1: переменная типа `FileExtensionManager` внутри класса	Добавить в конструктор аргумент, который будет использоваться для передачи зависимости. Теперь член тестируемого класса – подделка, больше ничего не меняется.
Уровень 2: тестируемый класс получает зависимость от фабричного класса	С помощью установки некоторого свойства сообщить фабричному классу о необходимости возвращать зависимость. Подделкой является член фабричного класса, а в тестируемом классе ничего не меняется.
Уровень 3: фабричный класс, который возвращает зависимость	Подменить экземпляр фабричного класса поддельной фабрикой, которая возвращает поддельную зависимость. Подделкой является фабрика, которая возвращает подделку; в тестируемом классе ничего не меняется.

Говоря об уровнях косвенности, следует понимать, что чем глубже мы спускаемся по кроличьей норе (то бишь по стеку вызовов в программе), тем больше возможностей манипулировать тестируемым

кодом, поскольку создаваемые заглушки имеют более широкую сферу ответственности. Но у такого подхода есть и обратная сторона: чем глубже уровень, тем труднее понять тест и тем сложнее найти подходящее место для зазора. Искусство состоит в том, чтобы отыскать разумный баланс между сложностью и разнообразием способов манипулирования, так чтобы тесты оставались удобочитаемыми, но мы при этом полностью контролировали ситуацию в тестируемом коде.

Для случая, показанного в листинге 3.6 (использование фабрики), добавление аргумента в конструктор только усложнило бы код, в котором и так есть вполне подходящий уровень для размещения зазора – фабрика на уровне 2. Здесь воспользоваться уровнем 2 проще всего, так как это требует минимального изменения кода.

- *Уровень 1 (подделка члена тестируемого класса)* – потребуется добавить конструктор, инициализировать член класса в конструкторе, задать параметры конструктора в тесте и думать о будущем использовании этого API в продуктовом коде. При таком подходе изменяется семантика работы с тестируемым классом, а этого лучше избегать, если нет основательной причины.
- *Уровень 2 (подделка члена фабричного класса)* – здесь все просто. Нужно лишь добавить установщик свойства в фабрику и записать в это свойство желаемую поддельную зависимость. Семантика продуктового кода не изменяется, все остается, как было, и код прост, как правда. Единственный минус состоит в том, что мы должны понимать, кто и когда вызывает фабрику, а это значит, что до начала реализации нужно провести некоторые изыскания. Разобраться в продуктовом коде, который вы раньше в глаза не видели, – задача не из простых, но все равно этот вариант представляется лучше прочих.
- *Уровень 3 (подделка фабричного класса)* – мы должны создать собственный вариант фабричного класса, для которого может и не быть интерфейса. В таком случае придется создать и интерфейс тоже. После этого предстоит создать экземпляр поддельной фабрики, которая будет возвращать класс поддельной зависимости (подделка возвращает подделку – супер!), и настроить поддельный фабричный класс в тестируемом классе. Идея подделки, возвращающей подделку, вообще плохо укладывается в голове, поэтому такого подхода лучше избегать, если мы хотим, чтобы тест оставался понятным.

Поддельный метод – использование локального фабричного метода (выделить и переопределить)

Этого способа нет ни на одном уровне в табл. 3.1; он создает совершенно новый уровень косвенности близко к поверхности тестируемого кода. Чем ближе к поверхности, тем меньше придется возиться с изменением зависимостей. В данном случае тестируемый класс сам является в некотором роде зависимостью, которой требуется манипулировать.

При таком подходе мы используем *виртуальный* метод самого тестируемого класса как фабрику, которая возвращает экземпляр диспетчера расширений. Поскольку метод виртуальный, его можно переопределить в производном классе, создав тем самым зазор. Внедрение в класс заглушки сводится к *наследованию* тестируемого класса ради *переопределения* виртуального фабричного метода. Тестированию затем подвергается производный класс. Фабричный метод в этом случае можно также назвать методом-заглушкой, который возвращает объект-заглушку. На рис. 3.8 показана соответствующая схема создания объектов.

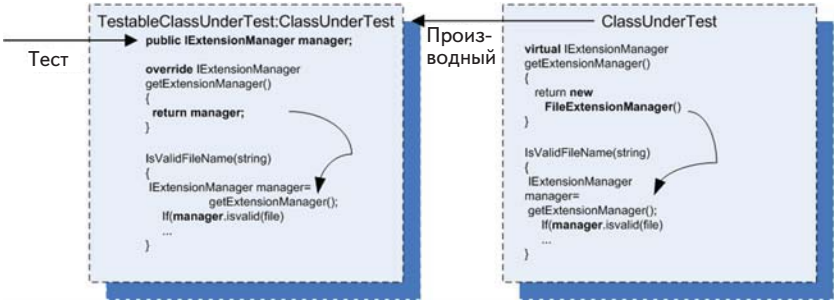


Рис. 3.8. Мы наследуем тестируемому классу и переопределяем его виртуальный метод, возвращая из него нужный нам объект, который реализует интерфейс `IExtensionManager`. Затем тестированию подвергается производный класс

Ниже перечислены шаги использования фабричного метода при тестировании.

- В тестируемом классе:
  - добавить виртуальный фабричный метод, который возвращает настоящий экземпляр;
  - воспользоваться этим фабричным методом, как обычно.

- В тестовом проекте:
  - создать новый класс;
  - унаследовать новый класс от тестируемого;
  - создать открытое поле (необязательно делать его свойством, допускающим чтение и установку) типа интерфейса, который должна реализовывать подделка (`IExtensionManager`);
  - переопределить виртуальный фабричный метод;
  - вернуть открытое поле.
- В коде теста:
  - создать экземпляр класса-заглушки, который реализует требуемый интерфейс (`IExtensionManager`);
  - создать экземпляр производного класса вместо тестируемого;
  - записать в вышеупомянутое открытое поле этого экземпляра ссылку на созданный экземпляр класса-заглушки.

Теперь благодаря переопределенному фабричному методу при тестировании продуктового класса будет использоваться подделка.

Вот как может выглядеть код, в котором применяется описанный способ.

Листинг 3.7. Подделка фабричного метода

```
public class LogAnalyzerUsingFactoryMethod
{
  public bool IsValidLogFileName(string fileName)
  {
     return GetManager().IsValid(fileName);
  }

  protected virtual IExtensionManager GetManager()
  {
     return new FileExtensionManager();
  }
}

[TestFixture]
public class LogAnalyzerTests
{
  [Test]
  public void overrideTest()
  {
     FakeExtensionManager stub = new FakeExtensionManager();
     stub.WillBeValid = true;

     TestableLogAnalyzer logan =
            new TestableLogAnalyzer(stub);
```

Используется виртуальный метод GetManager()

Возвращается зашитое значение

Создаем экземпляр класса, производного от тестируемого

```
      bool result = logan.IsValidLogFileName("file.ext");
      Assert.True(result);
    }
  }

  class TestableLogAnalyzer : LogAnalyzerUsingFactoryMethod
  {
    public TestableLogAnalyzer(IExtensionManager mgr)
    {
      Manager = mgr;
    }

    public IExtensionManager Manager;

    protected override IExtensionManager GetManager()
    {
      return Manager;
    }
  }

  internal class FakeExtensionManager : IExtensionManager
  {
    // то же, что в предыдущих примерах
    ...
  }
```

Возвращает то, что ему было сказано

Описанная техника называется «*выделить и переопределить*»; воспользовавшись ей пару раз, вы поймете, насколько это просто. В этой книге я буду применять ее и для других целей.

Совет. Об этом и других способах разрыва зависимостей можно прочитать в книге Майкла Фэзерса «Эффективная работа с унаследованным кодом», которую я ценю на вес золота.

Мощь техники выделения и переопределения проистекает из того, что она позволяет непосредственно подменить зависимость, не спускаясь в кроличью нору (то есть без подмены зависимости в глубоко вложенном коде). Поэтому ее можно реализовать быстро и чисто, хотя это может прийтись не по вкусу объектно-ориентированному эстету, т. к. получается код, в котором меньше интерфейсов, но больше виртуальных методов. Я люблю называть этот способ «выдернуть и переопределить» (ex-crack and override), и считаю, что, приобретя эту привычку, от нее уже трудно избавиться.

Когда использовать эту технику

Техника «выделить и переопределить» хороша при моделировании *входных данных* для тестируемого кода, но оказывается громоздкой,

если требуется проверить, как зависимость реагирует на данные, *исходящие* из тестируемого кода.

Например, она очень удобна, если код обращается к веб-службе, получает от нее *возвращаемое значение*, а мы хотим смоделировать эту ситуацию, вернув значение сами. Но все становится куда хуже, если мы хотим проверить, правильно ли код *отвечает* на запрос к веб-службе. В этом случае потребуется многое кодировать вручную, и для таких задач больше подходят изолирующие каркасы (как мы убедимся в следующей главе). Техника «выделить и переопределить» хороша, когда требуется имитировать возврат значений или целых интерфейсов, возвращающих значения, но не годится для проверки взаимодействий между объектами.

Я часто применяю эту технику, когда хочу имитировать входные данные для тестируемого кода, поскольку она позволяет несколько уменьшить количество изменений в семантике продуктового кода (добавление интерфейсов, конструкторов и т. д.), необходимых только для того, чтобы сделать его тестопригодным. А отказываюсь я от нее в том единственном случае, когда в продуктовом коде уже проложен для меня очевидный путь: существует интерфейс, который только и ждет, чтобы его подделали, или есть место, куда можно внедрить зазор. Если ничего такого нет, и сам класс не является запечатанным (или его можно сделать таковым, не вызывая гнева у коллег), то я первым делом изучаю возможность применения этой техники, а уже потом перехожу к более сложным альтернативам.

3.5. Варианты рефакторинга

У описанных выше простых методик введения *зазоров* в исходный код существует много вариантов. Например, параметр можно добавлять не в конструктор, а непосредственно в тестируемый метод. Вместо передачи интерфейса можно передавать базовый класс. И так далее. У каждого варианта есть свои плюсы и минусы.

Но использовать базовый класс вместо интерфейса не всегда желательно, в частности потому, что в продуктовом коде базовый класс, возможно, уже имеет зависимости, о которых вы должны знать и которые должны подменить. Поэтому реализовывать для тестирования производные классы сложнее, чем интерфейсы, так как в последнем случае мы точно знаем, что представляет собой истинная реализация, и имеем полный контроль над ней.

В главе 4 мы увидим, как можно уйти от самостоятельного написания подделок, реализующих интерфейсы, прибегнув к помощи каркасов, которые делают это во время выполнения.

А пока рассмотрим другой способ получить контроль над тестируемым кодом *без* использования интерфейсов. Выше мы уже встречались с подобным примером, но эта методика настолько эффективна, что заслуживает отдельного обсуждения.

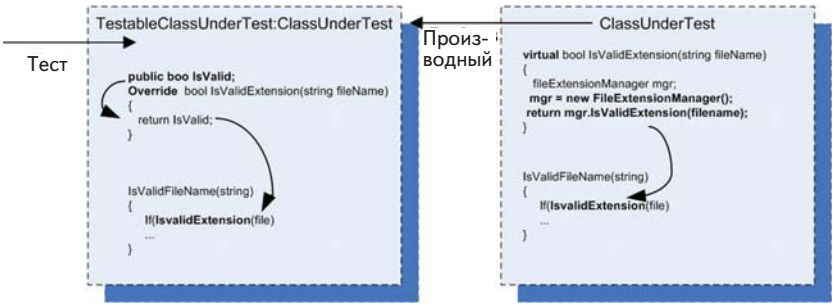


Рис. 3.9. Использование техники выделения и переопределения для возврата логического результата вместо фактического вызова зависимости. Здесь вместо вызова заглушки просто отдается поддельный результат

3.5.1. Использование выделения и переопределения для создания поддельных результатов

Мы уже видели пример выделения и переопределения в разделе 3.4.5. Идея в том, чтобы унаследовать тестируемому классу и переопределить в нем виртуальный метод, так чтобы он возвращал нашу заглушку.

Но почему надо на этом останавливаться? Что, если мы не можем или не хотим добавлять новый интерфейс, всякий раз как требуется получить контроль над каким-то поведением тестируемого класса? В таких случаях техника выделения и переопределения может упростить дело, потому что не требует добавления новых интерфейсов – достаточно унаследовать тестируемому классу и переопределить его поведение.

На рис. 3.9 показан еще один способ заставить тестируемый код всегда возвращать `true` в ответ на вопрос о допустимости расширения файла.

В тестируемом классе *виртуализируется* не фабричный метод, а результат вычисления. Это означает, что в производном классе переопределенный метод возвращает нужное нам значение без необходимости создавать интерфейс или новую заглушку.

В листинге ниже продемонстрировано использование этой техники.

Листинг 3.8. Выделенный метод возвращает результат, а не объект-заглушку

```
public class LogAnalyzerUsingFactoryMethod
{
  public bool IsValidLogFileName(string fileName)
  {
     return this.IsValid(fileName);
  }

  protected virtual bool IsValid(string fileName)
  {
     FileExtensionManager mgr = new FileExtensionManager();
     return mgr.IsValid(fileName);  <-- Возвращается результат истинной зависимости
  }
}

[Test]
public void overrideTestWithoutStub()
{
  TestableLogAnalyzer logan = new TestableLogAnalyzer();
  logan.IsSupported = true;  <-- Задаем значение поддельного результата
  bool result = logan.IsValidLogFileName("file.ext");
  Assert.True(result,"...");
}

class TestableLogAnalyzer: LogAnalyzerUsingFactoryMethod
{
  public bool IsSupported;

  protected override bool IsValid(string fileName)
  {
     return IsSupported;  <-- Возвращается поддельное значение, заданное в тесте
  }
}
```

Когда использовать технику выделения и переопределения

Основной побудительный мотив использования этой техники тот же, что в случае способа, обсуждавшегося в разделе 3.4.5. Если есть

возможность, то я использую именно эту технику, потому что она гораздо проще.

Наверное, сейчас вы думаете, что добавление всех этих конструкторов, установщиков свойств и фабрик ради удобства тестирования – решение спорное. Тем самым мы нарушаем некоторые фундаментальные принципы объектно-ориентированного проектирования и, в первую очередь, принцип инкапсуляции, который гласит: «Скрывай все, что пользователю классу не нужно видеть». Этим вопросом мы и займемся далее. (В главе 11 также обсуждается связь тестопригодности с проектированием.)

3.6. Преодоление проблемы нарушения инкапсуляции

Некоторые полагают, что приоткрывать внутреннюю структуру кода ради удобства тестирования плохо, потому что при этом нарушаются принципы объектно-ориентированного проектирования. Со всей определенностью могу им сказать: «Не валяйте дурака». Объектно-ориентированные методики придуманы, для того чтобы наложить ограничения на конечного пользователя API (каковым является программист, пользующийся вашей объектной моделью) и защитить объектную модель от неправильного и непредусмотренного использования. Объектная ориентация подразумевает также повторное использование кода и принцип единственной обязанности (требующий, чтобы у каждого класса была только одна обязанность).

Разрабатывая автономные тесты для своего кода, вы добавляете еще одного пользователя объектной модели (тест). Этот пользователь не менее важен, чем первоначальный, но у него другие цели. Тест предъявляет к объектной модели требования, которые, на первый взгляд, вступают в противоречие с несколькими принципами объектно-ориентированного проектирования и, прежде всего, с инкапсуляцией. Инкапсуляция внешних зависимостей без возможности их изменять, наличие закрытых конструкторов, запечатанных классов и невиртуальных методов, которые нельзя переопределить, – все это классические признаки перестраховочного дизайна (вопросы безопасности – это особая статья, тут я не возражаю). Проблема в том, что второму пользователю API, тесту, эти зависимости нужны не меньше, чем полезные функции кода. Я называю проектирование с учетом необходимости тестирования «тестопригодным объектно-

ориентированным проектированием» (ТООП). Мы еще вернемся к ТООП в главе 11.

Идея тестопригодного проектирования, по мнению некоторых, противоречит идее объектно-ориентированного проектирования. Для тех, кому просто жизненно необходимо примирить оба мира (и рыбку съесть, и косточкой не подавиться), я опишу несколько приемов, позволяющих скрыть дополнительные конструкторы и установщики свойств в выпускной версии или, по крайней мере, уменьшить их значимость.

Совет. Познакомиться со взглядом на цели проектирования, который более привержен идее тестпригодного дизайна, можно в книге «Чистый код» Боба Мартина.

3.6.1. internal и [InternalsVisibleTo]

Если вам не нравится, что все пользователи класса могут видеть открытый конструктор, поставьте модификатор `internal` вместо `public`. Затем все внутренние члены и методы можно сделать видимыми тестовой сборке, добавив атрибут `[InternalsVisibleTo]` на уровне сборки. Детали показаны в следующем листинге.

Листинг 3.9. Раскрытие внутренних членов тестовой сборке

```
public class LogAnalyzer
{
  ...
  internal LogAnalyzer (IExtensionManager extentionMgr)
  {
     manager = extentionMgr;
  }
  ...
}

using System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("AOUT.CH3.Logan.Tests")]
```

Такой код обычно помещается в файл AssemblyInfo.cs. Использование `internal` — хорошее решение в тех случаях, когда нет другого способа открыть внутреннюю структуру класса тестовому коду.

3.6.2. Атрибут [Conditional]

Атрибут `System.Diagnostics.ConditionalAttribute` печально известен своим интуитивно неочевидным поведением. Если этот атрибут добавляется к методу, то его аргументом служит строка, содержа-

щая имя условного параметра, передаваемого компилятору на этапе сборки (наиболее известны параметры `DEBUG` и `RELEASE`, которые в Visual Studio используются по умолчанию в соответствии с типом сборки).

Если на этапе сборки соответствующий параметр *не* задан, то все *вызовы* аннотированного метода не включаются в исполняемый файл. Например, при наличии в программе показанного ниже метода все обращения к нему удаляются из выпускной версии[6], но сам этот метод в ней остается.

```
[Conditional ("DEBUG")]
public void DoSomething()
{
}
```

Этот атрибут можно применять к методам (но не к конструкторам), которые должны вызываться только в некоторых отладочных режимах.

Примечание. Сами аннотированные методы по-прежнему видны в продуктовом коде – в отличие от приема, обсуждаемого ниже.

Важно отметить, что использование условной компиляции в продуктовом коде делает его менее понятным и повышает степень «спагеттиподобия». Остерегайтесь!

3.6.3. Использование директив #if и #endif для условной компиляции

Если поместить методы или специальные конструкторы, предназначенные только для тестирования, между директивами `#if` и `#endif`, то они будут компилироваться, только когда при сборке задан соответствующий флаг.

Листинг 3.10. Использование специальных флагов при сборке

```
#if DEBUG
public LogAnalyzer (IExtensionManager extensionMgr)
{
  manager = extensionMgr;
}
```

[6] Одно из ограничений атрибута `Conditional` состоит в том, что метод, к которому он применяется, должен возвращать `void` (иначе было бы невозможно корректно удалить все обращения). – *Прим. перев.*

```
#endif
...
#if DEBUG
[Test]
public void IsValidFileName_SupportedExtension_True()
{
  ...
  // создать анализатор и внедрить заглушку
  LogAnalyzer log = new LogAnalyzer (myFakeManager);
  ...
}
#endif
```

Этот метод применяется часто, но при неумеренном употреблении приводит к неразборчивому коду. Подумайте об использовании атрибута `[InternalsVisibleTo]`, так код станет яснее.

3.7. Резюме

В первых двух главах мы начали с написания простых тестов, но в них были зависимости, которые нужно было как-то обойти. В этой главе мы научились заглушать зависимости, применяя интерфейсы и наследование.

Заглушку можно внедрить в код разными способами. Хитрость в том, чтобы найти или создать правильный уровень косвенности, а затем использовать его как *зазор*, позволяющий вставить в работающий код заглушку.

Мы называем такие классы *подделками*, поскольку не хотим поручать им только роль заглушек или подставок.

Чем глубже уровень взаимодействия, тем труднее понять тест, сам тестируемый код и его взаимодействия с другими объектами. Чем ближе к поверхности тестируемого объекта, тем проще разобраться в тесте и настроить его, но, возможно, придется отказаться от некоторых средств контроля над окружением тестируемого объекта.

Изучайте различные пути внедрения заглушек в код. Овладев ими в совершенстве, вы сумеете с большей уверенностью выбрать наиболее подходящий метод из арсенала имеющихся.

Техника «выделить и переопределить» прекрасно подходит, когда нужно имитировать входные данные для тестируемого кода, но если требуется также протестировать взаимодействия между объектами (тема следующей главы), то возвращайте интерфейс, а не просто значение. Так вы упростите себе жизнь.

ТООП обладает рядом интересных преимуществ по сравнению с классическим объектно-ориентированным проектированием, например, оно обеспечивает удобство сопровождения, не ограничивая возможность писать тесты продуктового кода.

В главе 4 мы рассмотрим другие проблемы, связанные с зависимостями, и покажем, как они решаются; обсудим, как уйти от рукописных подделок интерфейсов и как тестировать взаимодействия между объектами в автономных тестах.

ГЛАВА 4. Тестирование взаимодействий с помощью подставных объектов

В этой главе:

- Что такое тестирование взаимодействий.
- Что такое подставные объекты.
- Различия между подделками, подставками и заглушками.
- Рекомендации по работе с подставными объектами.

В предыдущей главе мы решили проблему тестирования кода, зависящего от других объектов. Благодаря заглушкам мы обеспечили тестируемый код нужными ему входными данными, и, как следствие, смогли независимо протестировать его логику.

Но до сих пор мы писали только тесты единиц работы, результаты которых относятся к первым двум типам: возврат значения и изменение состояния системы.

В этой главе мы рассмотрим, как тестировать результаты третьего типа – обращение к стороннему объекту. Мы проверим, правильно ли объект вызывает другие объекты. Тестируемый объект может не возвращать значение и не сохранять состояние, но при этом иметь сложную логику, правильное выполнение которой должно приводить к вызовам других объектов, не находящихся под вашим контролем и не являющихся частью тестируемой единицы работы. Применение изученных выше приемов здесь ничего не даст, потому что не

существует никакого API, который можно было бы вынести наружу и использовать для проверки изменений в состоянии тестируемого объекта. Как убедиться, что наш объект правильно взаимодействует с другими? С помощью подставных объектов.

Первым делом следует определить, что такое тестирование взаимодействий и чем оно отличается от уже знакомых нам видов тестирования – на основе значений и состояния.

4.1. Сравнение тестирования взаимодействий с тестированием на основе значений и состояния

В главе 1 я определил три типа конечных результатов, порождаемых единицей работы. Теперь я определю тестирование взаимодействий, которое имеет непосредственное отношение к результатам третьего типа: обращение к стороннему компоненту. В случае тестирования на основе значений мы проверяем значение, возвращенное функцией. В случае тестирования на основе состояния проверяются видимые изменения в состоянии тестируемой системы – после выполнения действия, изменяющего состояние.

Определение. *Тестированием взаимодействий* называется проверка того, как объект посылает сообщения другим объектам (вызывает их методы). Такой вид тестирования применяется, когда конечным результатом единицы работы является обращение к другому объекту.

Можно считать, что тестирование взаимодействий *управляется действиями*. Это означает, что проверяется, какое действие предпринимает объект (например, отправляет сообщение другому объекту).

Тестирование взаимодействий *всегда* следует оставлять на крайний случай. Это очень важная мысль. Сначала убедитесь, что никак не можете протестировать единицу работы по результатам первых двух типов (значение и состояние), потому что тесты взаимодействий оказываются многократно сложнее. Но иногда, как, например, при обращении к стороннему средству протоколирования, взаимодействия между объектами и являются конечным результатом. В таком случае делать нечего – придется тестировать само взаимодействие.

Хочу отметить, что не все согласны с тем, что подставные объекты следует использовать, только когда нет никаких других способов про-

тестировать код. В книге «Growing Object-Oriented Software, Guided by Tests» ее авторы Стив Фримэн (Steve Freeman) и Нат Прайс (Nat Pryce) пропагандируют идеологию так называемой «лондонской школы TDD», в которой подставки и заглушки используются как средство унификации проектирования ПО. Не то чтобы я был полностью не согласен с таким способом проектирования кода. Но эта книга *не* о проектировании, а, если говорить только об удобстве сопровождения, то в моих тестах использование подставок создает больше проблем, чем отказ от них. Повторяю, это мой личный опыт, но я постоянно изучаю новые веяния. Не исключено, что в следующем издании этой книги моя точка зрения на этот предмет будет прямо противоположной.

Тестирование взаимодействий в том или ином виде существовало с первых дней автономного тестирования. Тогда для него еще не было ни названия, ни паттернов, но знать-то, что один объект правильно вызывает другой, все равно было необходимо. Однако при реализации этой идеи часто либо перебарщивали, либо плохо старались, поэтому тесты получались несопровождаемыми и неудобочитаемыми. Потому-то я и рекомендую по возможности тестировать результаты двух других типов.

Чтобы разобраться в плюсах и минусах тестирования взаимодействий, рассмотрим пример. Допустим, имеется система орошения, и вы настроили время полива дерева у себя в саду: сколько раз в день и каким количеством воды. Есть два способа проверить правильность работы системы:

- *Интеграционный тест на основе состояния* (именно интеграционный, а не автономный). Запустить систему на 12 часов, в течение которых она должна полить дерево несколько раз. В конце периода проверить состояние политого дерева. Достаточно ли влажная земля, хорошо ли дерево себя чувствует, зеленые ли листья и т. д. Такой тест может оказаться довольно трудным, но если вы в состоянии его выполнить, то сможете определить, работает ли система орошения. Я называю этот тест интеграционным, потому что он о-о-очень медленный и для его проведения необходимо все окружение системы орошения.
- *Тестирование взаимодействий.* На конце шланга закрепить устройство, которое регистрирует время полива и количество прошедшей через него воды. В конце дня проверить, что устройство срабатывало нужное число раз и всякий раз регистри-

ровало ожидаемый объем воды. А о дереве вообще не думать. На самом деле, для проверки работоспособности системы дерево и не нужно. Можно пойти дальше и модифицировать синхрогенератор в блоке орошения (подставив вместо него заглушку), чтобы система думала, что время полива настало в момент, когда вам будет удобно. Тогда, чтобы проверить ее работу, не нужно будет ждать (в данном примере 12 часов).

Как видите, в этом случае тест взаимодействия может заметно упростить жизнь.

Но иногда лучше тестировать на основе состояния, потому что организовать тестирование взаимодействий слишком сложно.

Так обстоит дело с манекенами для аварийных испытаний: автомобиль врезается в неподвижное препятствие на определенной скорости, а после столкновения результат испытания определяется по состоянию автомобиля и манекенов. Проведение такого испытания в лаборатории в виде теста взаимодействия может оказаться слишком сложным, поэтому предпочтение отдается физическим испытаниям с последующей оценкой состояния. (Ученые и инженеры работают над компьютерными моделями столкновений, но пока они еще очень далеки от натурных испытаний.)

Но вернемся к системе орошения. Что это за устройство, которое регистрирует информацию о поливе? Это поддельный шланг, заглушка, если хотите. Только это очень умная заглушка – она запоминает обращения к себе. Мы используем ее, чтобы понять, прошел тест или нет. Это и есть одна из характеристик подставного объекта. А синхрогенератор, который мы заменили подделкой? Это заглушка, потому что всего лишь моделирует время, чтобы было удобнее тестировать другую часть системы.

Определение. *Подставным объектом* называет такой поддельный объект, который определяет, прошел автономный тест или нет. Для этого он проверяет, что тестируемый объект вызывал поддельный так, как ожидается. Обычно в каждом тесте участвует не более одного подставного объекта.

На первый взгляд, подставной объект не слишком отличается от заглушки, но на самом деле различия настолько велики, что заслуживают отдельного обсуждения и даже специального синтаксиса в различных каркасах, как мы убедимся в главе 5. Мы уточним, в чем они состоят.

Сейчас, когда у вас уже есть представление о подделках, подставках и заглушках, я могу привести формальное определение поддельного объекта.

Определение. *Поддельный объект*, или *подделка* – это общий термин для обозначения заглушек и подставок (рукописных и прочих), поскольку те и другие имитируют настоящий объект. Считать ли объект заглушкой или подставкой, зависит от способа его использования в тесте. Если объект применяется для проверки взаимодействия (и относительно него высказывается утверждение), то это подставка. В противном случае – заглушка.

Копнем глубже и рассмотрим внимательнее разницу между двумя типами подделок.

4.2. Различия между подставками и заглушками

Заглушки подменяют объект, чтобы можно было протестировать другой объект. На рис. 4.1 показано взаимодействие между заглушкой и тестируемым классом.

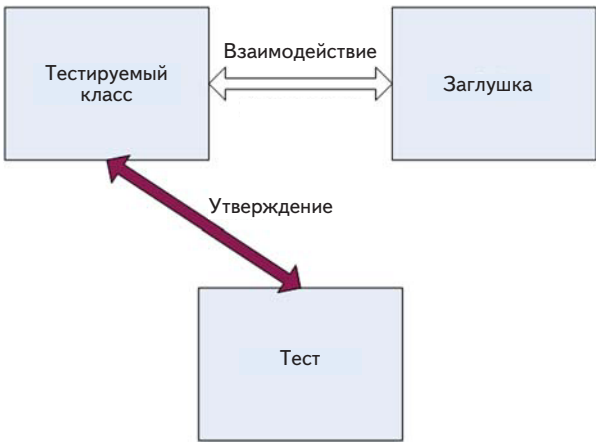


Рис. 4.1. При использовании заглушки утверждение высказывается относительно тестируемого класса. Заглушка – лишь вспомогательное средство для выполнения теста

Различие между подставками и заглушками важно, потому что во многих современных инструментах и каркасах (а также в статьях) этими терминами обозначаются разные вещи. Важно оно и потому, что при изучении тестов, написанных другими людьми, необходимо понимать, что присутствует более одного подставного объекта, – это существенное умение, о котором еще будет речь ниже. Относительно

смысла этих терминов существует путаница, и многие, похоже, считают их синонимами. Но, поняв, в чем разница, вы сможете более осознанно оценивать инструменты, каркасы и API и лучше понимать, что они делают.

На первый взгляд, различие между подставками и заглушками может показаться несущественным или даже не существующим. Разница тонкая, но она важна, т. к. во многих изолирующих каркасах, с которыми мы будем иметь дело в следующих главах, эти термины употребляются для описания разных видов поведения каркаса. Основное различие в том, что заглушка не может стать причиной отказа теста. А подставка может.

Основополагающее свойство заглушки – тот факт, что она не влияет на исход теста. Утверждения высказываются только относительно тестируемого класса, но не заглушки.

С другой стороны, подставной объект анализируется, чтобы узнать, прошел тест или нет. На рис. 4.2 показано взаимодействие между тестом и подставным объектом. Обратите внимание, что утверждение высказано относительно подставки.

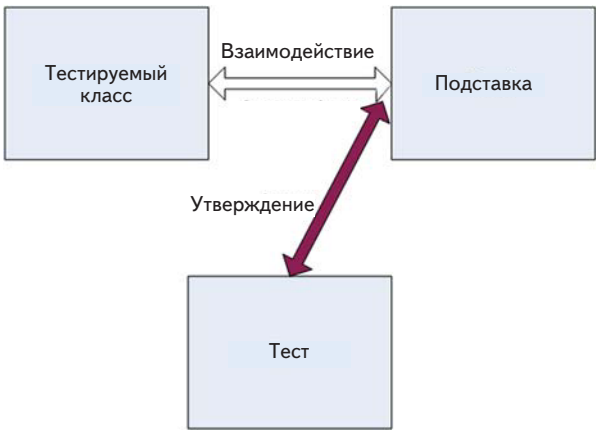


Рис. 4.2. Тестируемый класс взаимодействует с подставным объектом и взаимодействие регистрируется внутри подставки. Подставной объект используется, чтобы узнать, прошел тест или нет

Еще раз повторим, что подставка – это объект, с помощью которого мы узнаем, прошел тест или нет. Проиллюстрируем эти идеи в действии, построив собственный подставной объект.

4.3. Пример простой рукописной подставки

Подставной объект создается и используется практически так же, как заглушка, только делает он чуть побольше заглушки: сохраняет историю взаимодействия, которую впоследствии можно проверить в виде *ожиданий*.

Введем еще одно требование к классу `LogAnalyzer`. Теперь он должен передавать внешней веб-службе сообщение об ошибке всякий раз, как встретит слишком короткое имя файла.

К сожалению, веб-служба, взаимодействие с которой требуется протестировать, еще не готова, а даже если бы она и работала в полном объеме, все равно обращаться к ней из тестов было бы слишком долго. Поэтому мы подвергнем проект рефакторингу и создадим новый интерфейс, для реализации которого впоследствии можно будет написать подставной объект. В интерфейсе будут объявлены методы, необходимые для вызова веб-службы, и ничего более.

На рис. 4.3 показано место подставного класса `FakeWebService` в тесте.

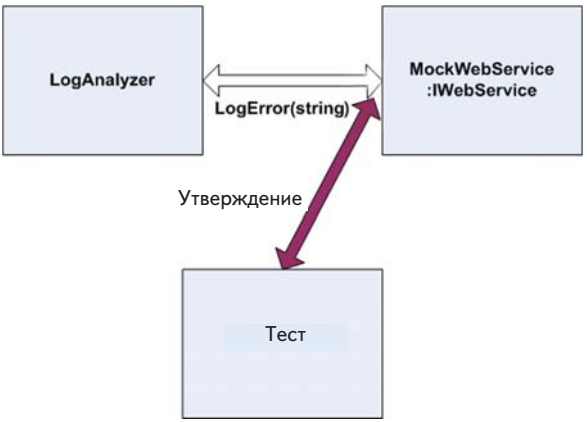


Рис. 4.3. В тесте создается объект `FakeWebService` для регистрации сообщений, полученных от `LogAnalyzer`. Затем мы сможем высказать утверждение относительно `FakeWebService`

Первым делом выделим простой интерфейс, которым можно будет воспользоваться в тестируемом коде вместо прямого обращения к веб-службе:

```
public interface IWebService
{
  void LogError(string message);
}
```

Этот интерфейс пригодится и в том случае, если мы захотим создавать не только подставки, но и заглушки. Он позволяет подменить внешнюю зависимость, которую мы не контролируем.

Затем создаем сам подставной объект. Выглядит он, как заглушка, но содержит дополнительный код, превращающий его в подставку:

```
public class FakeWebService:IWebService
{
  public string LastError;

  public void LogError(string message)
  {
     LastError = message;
  }
}
```

Этот написанный вручную класс реализует интерфейс, как и заглушка, но еще сохраняет некоторое состояние, чтобы впоследствии тест мог сделать утверждение и проверить, что подставка вызывалась правильно. Однако пока это еще не подставной объект. Таковым он станет, лишь когда мы *используем его в этом качестве* в тесте.

Примечание. В книге Джерарда Месароша «Шаблоны тестирования xUnit. Рефакторинг кода тестов» это называется тестовым шпионом.

В следующем листинге приведен код теста.

Листинг 4.1. Тестирование класса `LogAnalyzer` с помощью подставного объекта

```
[Test]
public void Analyze_TooShortFileName_CallsWebService()
{
  FakeWebService mockService = new FakeWebService();
  LogAnalyzer log = new LogAnalyzer(mockService);
  string tooShortFileName=”abc.ext”;

  log.Analyze(tooShortFileName);

  StringAssert.Contains(“Слишком короткое имя файла: abc.ext”,
        mockService.LastError);
}

public class LogAnalyzer
{
```

Утверждение относительно подставного объекта

```
    private IWebService service;

    public LogAnalyzer(IWebService service)
    {
       this.service = service;
    }

    public void Analyze(string fileName)
    {
       if(fileName.Length<8)
       {
          service.LogError("Слишком короткое имя файла: "
             + fileName);
       }
    }
}
```

В продуктовом коде протоколируется ошибка

Обратили внимание, что утверждение высказано относительно подставного объекта, а не объекта `LogAnalyzer`? Это потому, что мы тестируем взаимодействие между `LogAnalyzer` и веб-службой. Мы применяем те же приемы внедрения зависимости, что и в главе 3, но на этот раз исход теста определяется подставным объектом (который используется вместо заглушки).

Также отметим, что проверка вынесена за пределы кода подставного объекта. Тому есть две причины.

- Мы хотели бы иметь возможность повторно использовать этот подставной объект в других тестах, где относительно сообщения делаются другие утверждения.
- Если бы утверждение было помещено внутрь рукописного класса подделки, то читатель теста не смог бы понять, что утверждается. Мы убрали из кода теста существенную информацию, сделав его менее удобочитаемым и удобным для сопровождения.

Возможно, в своих тестах вы встретитесь с необходимостью подменять сразу несколько объектов. Далее мы рассмотрим комбинирование заглушек и подставок. Вы увидите, что несколько заглушек в одном тесте – совершенно обычное дело, а вот наличие нескольких подставок может свидетельствовать о методической ошибке, поскольку означает, что тестируется более одного аспекта.

4.4. Совместное использование заглушки и подставки

Рассмотрим более сложную задачу. На этот раз `LogAnalyzer` должен обратиться к веб-службе, а если та вернет ошибку, то записать эту

ошибку в другую внешнюю зависимость, отправив по электронной почте сообщение администратору веб-службы (рис. 4.4).

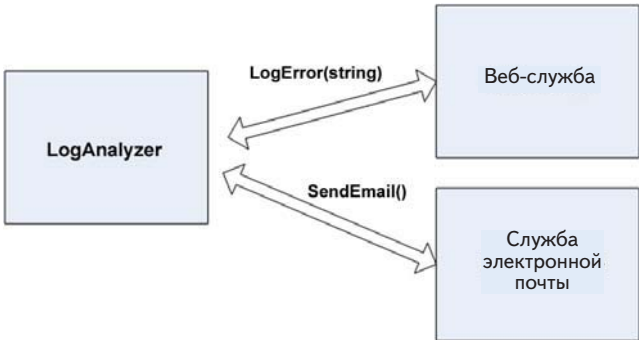


Рис. 4.4. У класса `LogAnalyzer` две внешних зависимости: веб-служба и служба электронной почты. Мы должны протестировать логику `LogAnalyzer` при обращении к ним

Вот как выглядит логика класса `LogAnalyzer`:

```
if(fileName.Length<8)
{
  try
  {
     service.LogError("Слишком короткое имя файла: " + fileName);
  }
  catch (Exception e)
  {
     email.SendEmail("a","subject",e.Message);
  }
}
```

Обратите внимание, что здесь имеет место только взаимодействие с внешними объектами; никакие значения не возвращаются и состояние системы не изменяется. Как проверить, что `LogAnalyzer` правильно вызывает почтовую службу, если веб-служба возбуждает исключение?

Вот на какие вопросы мы должны ответить:

- как подменить веб-службу?
- как имитировать исключение от веб-службы, чтобы можно было протестировать обращение к почтовой службе?
- как узнать, что почтовая служба была вызвана правильно, да и вообще вызвана?

Для ответа на первые два вопроса можно воспользоваться заглушкой веб-службы. Для решения третьей проблемы понадобится подставка почтовой службы.

В нашем тесте будет два поддельных объекта. Один – подставная почтовая служба, с помощью которой мы проверим, что почтовой службе были переданы правильные параметры. Второй – заглушка, которая позволит имитировать исключение, возбужденное веб-службой. Это именно заглушка, потому что мы не станем использовать поддельную веб-службу для проверки исхода теста, она нужна только лишь для обеспечения его работы. Почтовая же служба является подставкой, потому что мы высказываем утверждение относительно правильности обращения к ней. Это наглядно показано на рис. 4.5.

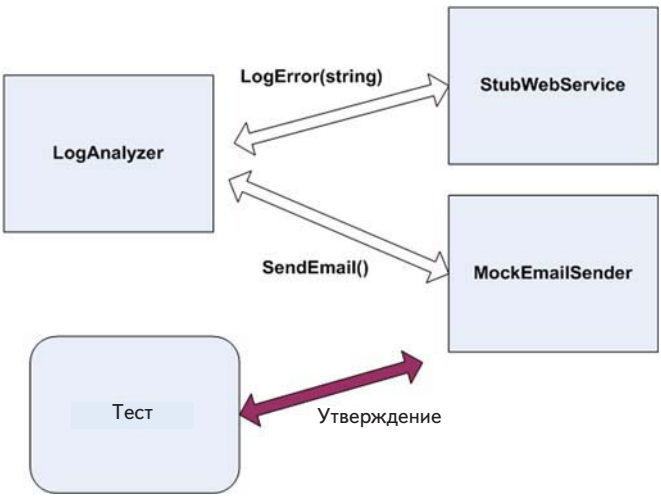


Рис. 4.5. Мы заглушаем веб-службу, чтобы имитировать исключение. Затем используем подставную почтовую службу, чтобы проверить, правильно ли она вызывается. Тест в целом проверяет взаимодействие `LogAnalyzer` с другими объектами

В следующем листинге показана реализация схемы, изображенной на рис. 4.5.

Листинг 4.2. Тестирование `LogAnalyzer` с помощью заглушки и подставки

```
public interface IEmailService
{
```

```
    void SendEmail(string to, string subject, string body);
}

public class LogAnalyzer2
{
    public LogAnalyzer2(IWebService service, IEmailService email)
    {
        Email = email,
        Service = service;
    }

    public IWebService Service
    {
        get ;
        set ;
    }

    public IEmailService Email
    {
        get ;
        set ;
    }

    public void Analyze(string fileName)
    {
        if(fileName.Length<8)
        {
            try
            {
                Service.LogError("Слишком короткое имя файла: " + fileName);
            }
            catch (Exception e)
            {
                Email.SendEmail("someone@somewhere.com",
                    "can't log",e.Message);
            }
        }
    }
}

[TestFixture]
public class LogAnalyzer2Tests
{
    [Test]
    public void Analyze_WebServiceThrows_SendsEmail()
    {
        FakeWebService stubService = new FakeWebService();
        stubService.ToThrow= new Exception("fake exception");

        FakeEmailService mockEmail = new FakeEmailService();

        LogAnalyzer2 log = new LogAnalyzer2(stubService,mockEmail);
```

```
      string tooShortFileName="abc.ext";
      log.Analyze(tooShortFileName);

      StringAssert.Contains("someone@somewhere.com",mockEmail.To);
      StringAssert.Contains("fake exception",mockEmail.Body);
      StringAssert.Contains("can't log",mockEmail.Subject);
   }
}

public class FakeWebService:IWebService
{
   public Exception ToThrow;
   public void LogError(string message)
   {
      if(ToThrow!=null)
      {
         throw ToThrow;
      }
   }
}

public class FakeEmailService:IEmailService
{
   public string To;
   public string Subject;
   public string Body;

   public void SendEmail(string to,
         string subject,
         string body)
   {
      To = to;
      Subject = subject;
      Body = body;
   }
}
```

Стоит отметить несколько интересных моментов в этом коде.

- Иногда наличие нескольких утверждений представляет проблему, потому что первое ложное утверждение возбуждает специальное исключение, перехватываемое исполнителем тестов. Это означает, что все строки после этого утверждения вообще не будут выполнены. В данном конкретном случае все хорошо, так как если какое-то утверждение оказалось ложно, то остальные нас уже не интересуют, поскольку относятся к тому же объекту и являются частями одной и той же «функции».

- Если вам необходимо, чтобы последующие утверждения были проверены, даже если первое оказазалось ложным, то это верный признак того, что данный тест следует разбить на несколько. Или можно было бы завести новый класс `EmailInfo` с тремя атрибутами и передавать объект этого класса методу `SendMail`, а затем в тесте создать экземпляр этого класса с ожидаемыми свойствами. Тогда удалось бы обойтись одним утверждением.

Вот как это выглядело бы:

```
class EmailInfo
{
  public string Body;
  public string To;
  public string Subject;
}

[Test]
public void Analyze_WebServiceThrows_SendsEmail()
{
  FakeWebService stubService = new FakeWebService();
  stubService.ToThrow= new Exception("fake exception");

  FakeEmailService mockEmail = new FakeEmailService();

  LogAnalyzer2 log = new LogAnalyzer2(stubService,mockEmail);

  string tooShortFileName="abc.ext";
  log.Analyze(tooShortFileName);

  EmailInfo expectedEmail = new EmailInfo {
     Body = "fake exception",
     To = "someone@somewhere.com",
     Subject = "can't log"
  }

  Assert.AreEqual(expectedEmail, mockEmail.email);
}

public class FakeEmailService:IEmailService
{
  public EmailInfo email = null;
  public void SendEmail(EmailInfo emailInfo)
  {
     email = emailInfo;
  }
}
```

Если имеется ссылка на *сам* объект, который ожидается в качестве результата, то можно было бы записать такое утверждение:

```
Assert.AreSame(expectedEmail, mockEmail.email);
```

Еще один важный вопрос – сколько заглушек и подставок можно использовать в одном тесте.

4.5. Одна подставка на тест

В тесте, где проверяется что-то одно (а именно так я рекомендую писать тесты), не должно быть более одного подставного объекта. Все остальные поддельные объекты должны выступать в роли заглушек. Наличие нескольких подставок в одном тесте обычно означает, что тестируется более одного аспекта, такие тесты часто оказываются слишком сложными или хрупкими (подробнее об этом см. главу 8).

Если вы будете следовать этой рекомендации, то при переходе к более сложным тестам всегда сможете спросить себя: «Который тут объект-подставка?» Выявив подставной объект, вы сможете оставить остальные заглушками и не высказывать относительно них утверждений. (Если вы все же обнаружите утверждения относительно подделок, которые явно используются как заглушки, насторожитесь. Это признак избыточного специфицирования.)

Избыточное специфицирование имеет место, когда вы говорите, что должно произойти слишком много вещей, о которых ваш тест и знать-то не должен, например, что вызываются заглушки. Из-за таких лишних спецификаций тест может не пройти по причинам, не имеющим отношения к делу: вы внесли изменения в продуктовый код, но хотя его конечный результат по-прежнему правилен, тест ругается: «Я не прошел! Ты сказал, что этот метод должен вызываться, а он не вызывался! Хнык!»

Тогда вам придется постоянно изменять тесты, приводя их в соответствие с внутренней реализацией кода. В конечном итоге вам это надоест, вы зададитесь вопросом «И за каким чертом я все это делаю?» и начнете удалять докучливые тесты.

Игра окончена.

Специфицируйте только один из трех возможных типов результата единицы работы, иначе окажетесь в аду, смерть будет собирать свою жатву и ангелы низринутся с небес. Я вас предупредил.

Далее мы рассмотрим более сложный сценарий: использование заглушки для возврата подделки (подставки или заглушки), которая и будет использоваться в приложении.

4.6. Цепочки подделок: заглушки, порождающие подставки или другие заглушки

Один из самых распространенных в наши дни видов мошенничества устроен очень просто. Большому количеству получателей отправляется поддельное сообщение электронной почты. В этом сообщении от мошеннического банка или онлайновой службы говорится, что клиент должен проверить свой баланс или внести какие-то изменения в учетную запись на сайте.

Все ссылки в сообщении ведут на поддельный сайт. Он выглядит точно так же, как настоящий, но создан с единственной целью: собирать данные о ничего не подозревающих клиентах этого бизнеса. По существу, мы имеем поддельное письмо, ведущее на поддельный сайт. Это простенькая цепочка вранья называется фишинговой атакой и оказывается куда более прибыльным делом, чем вы можете себе представить.

Но к чему этот разговор? Да к тому, что иногда требуется, чтобы поддельный объект возвращал (из метода или свойства) другой поддельный компонент, так что порождается цепочка заглушек, заканчивающаяся подставным объектом где-то глубоко в недрах системы, и этот подставной объект позволяет собрать какие-то данные во время теста. Мы можем создать заглушку, которая порождает подставной объект, запоминающий данные.

Дизайн многих тестируемых систем допускает создание сложных цепочек объектов. Нередко можно встретить код такого вида:

```
IServiceFactory factory = GetServiceFactory();
IService service = factory.GetService();
```

Или такого:

```
String connstring =
    GlobalUtil.Configuration.DBConfiguration.ConnectionString;
```

Предположим, что на время тестирования мы хотим заменить строку соединения. Можно было бы сделать свойство `Configuration` объекта `GlobalUtil` объектом-заглушкой. Затем точно также мы превратили бы в заглушку свойство `DBConfiguration` и в конечном итоге вернули бы поддельный объект, который можно было бы использовать как подставку или заглушку строки соединения.

Техника, безусловно, мощная, но стоит задаться вопросом, не лучше ли переработать код как-то так:

```
string connstring = GetConnectionString();
protected virtual string GetConnectionString()
{
 return GlobalUtil.Configuration.DBConfiguration.ConnectionString;
}
```

А затем переопределить виртуальный метод, как описано в главе 3 (раздел 3.4.5). Такой код будет проще читать и сопровождать и не потребуется вводить новые интерфейсы, чтобы вставить в систему еще две заглушки.

Совет. Есть еще один неплохой способ избежать цепочек вызовов – создать специальные классы, обертывающие API, чтобы упростить его использование и тестирование. Подробнее об этом методе читайте в книге Майкла Фэзерса «Эффективная работа с унаследованным кодом», где этот паттерн называется «Адаптация параметра».

У рукописных заглушек и подставок есть достоинства, но и проблемами они также не обделены. Рассмотрим последние более пристально.

4.7. Проблемы рукописных заглушек и подставок

При использовании рукописных заглушек и подставок возникают следующие проблемы:

- на их написание требуется время;
- трудно писать заглушки и подставки для классов и интерфейсов с большим количеством методов, свойств и событий;
- для сохранения состояния при вызовах методов подставки приходится писать много стереотипного кода;
- для проверки того, что параметры метода правильно заданы вызывающей стороной, необходимы многочисленные утверждения – писать их скучно и долго;
- повторно использовать заглушку или подставку в других тестах трудно. В простых случаях все работает, но если в интерфейсе более двух-трех методов, то сопровождать код становится утомительно.

Существует ли в программе такое место, где подделка выступает одновременно в роли заглушки и подставки? Очень редко – может

быть, одно-два места во всем проекте. Мне такое доводилось встречать всего пару разу за последние два года.

Подставка может оказаться заглушкой, если требуется воспользоваться подставным объектом, в котором есть метод, возвращающий значение. Чтобы компилятор не ругался (ох, уж эти статические языки – подставляют нас на каждом углу), поддельный объект тоже должен возвращать какое-то фиктивное значение, иначе код не только не выполнится, но даже не откомпилируется. Вот в таком случае подставка оборачивается одновременно заглушкой, но я утверждаю, что если подставной объект возвращает системе какое-то значение, то, скорее всего, вы изначально тестируете не тот конечный результат. Я обычно ищу такие результаты, которые являются обращениями к сторонней системе, не возвращающими ничего. Меня интересуют в первую очередь методы типа `void`. Иногда система спроектирована так, что метод, обращающийся к стороннему компоненту, все же возвращает значение (так часто делают при программировании на C++ для информирования об ошибках). Это чуть ли не единственный случай, когда я готов смириться с тем, что подставка является и заглушкой тоже. Приведу пример:

```
public interface IComNotificationService
{
  int SendNotification(string info);
}
```

В данном случае если конечным результатом работы является вызов метода `SendNotification`, то подставной объект используется для проверки того, что метод вызывался, но, чтобы удовлетворить компилятор, придется также вернуть какое-то значение, которое, однако, в данном тесте безразлично.

Я перечислил проблемы, свойственные ручному написанию заглушек и подставок. По счастью, есть и другие способы их создания, с которыми мы познакомимся в следующей главе.

4.8. Резюме

В этой главе мы рассмотрели различие между заглушками и подставными объектами. Подставной объект похож на заглушку, но относительно него можно высказывать утверждения в тесте. Заглушка никогда не является причиной отказа теста и существует только для имитации различных ситуаций. Это различие существенно, потому что подразумевается во многих каркасах для генерации подставок, с

которыми мы познакомимся в следующей главе, так что вы должны понимать, когда использовать подставку, а когда – заглушку.

Комбинирование заглушек и подставок в одном тесте – действенная техника, но нужно следить за тем, чтобы в каждом тесте было не более одной подставки. Все остальные поддельные объекты должны быть заглушками, не способными привести к отказу теста. Следование этому совету приводит к тестам, которые проще сопровождать и которые реже ломаются при изменении внутренней структуры кода.

Заглушки, порождающие другие заглушки или подставки, – эффективный способ внедрения поддельных зависимостей в код, где для получения данных используются объекты. Для этой цели очень удобны фабричные классы и методы. Никто не мешает создавать длинные цепочки заглушек, но в какой-то момент возникает вопрос, а стоит ли игра свеч. В таком случае прочитайте еще раз описание методик внедрения заглушек в проект в главе 3. (В следующей главе обсуждаются некоторые изолирующие каркасы, в которых для создания цепочки вызовов подделок достаточно одной строки кода – рекурсивные подделки вам в помощь!)

Одна из самых распространенных ошибок при написании тестов – чрезмерное использование подставных объектов (избыточное специфицирование). Следует по возможности избегать проверок результатов обращения к методам объекта, который одновременно используется в роли заглушки и подставки. В одном тесте может встречаться несколько заглушек, потому что у класса может быть несколько зависимостей. Нужно только не забывать об удобочитаемости теста. Структурируйте код так, чтобы читатель теста понимал, что в нем делается.

Возможно, вы сочтете неудобным вручную писать заглушки и подставки, имея дело с объемными интерфейсами или сложными сценариями тестирования взаимодействий. Это действительно так, и, как вы увидите в следующей главе, существуют способы получше. Но нередко рукописные заглушки и подставки способны дать фору каркасам благодаря своей простоте и понятности. Решение о том, когда какой инструмент выбрать, – это искусство.

В следующей главе мы рассмотрим изолирующие каркасы, которые позволяют автоматически во время выполнения создавать заглушки и подставки и использовать их по крайней мере столь же эффективно, как написанные вручную, а зачастую намного эффективнее.

ГЛАВА 5. Изолирующие каркасы генерации подставных объектов

В этой главе:

- Что такое изолирующий каркас.
- Применение NSubstitute для создания заглушек и подставок.
- Более сложные сценарии использования заглушек и подставок.
- Как избежать неправильного употребления изолирующих каркасов.

В предыдущей главе мы видели, как писать заглушки и подставки вручную и какие при этом возникают проблемы. В этой главе мы рассмотрим несколько элегантных решений подобных проблем за счет применения *изолирующих каркасов* – библиотек, которые умеют создавать и настраивать поддельные объекты *во время выполнения*. Эти объекты называются *динамическими* заглушками и подставками.

Мы начнем с изложения общих сведений об изолирующих каркасах (или каркасов генерации подставных объектов, но слово «подставной» и так уже используется слишком часто) и их возможностях. Я употребляю термин «изолирующий каркас», потому что назначение этого ПО состоит в том, чтобы изолировать единицу работы от зависимостей. Мы подробно обсудим один конкретный каркас: NSubstitute. Вы увидите, как его можно использовать для тестирования различных аспектов, создания заглушек, подставок и других интересных вещей.

Но не в самом NSubstitute (для краткости NSub) дело. Знакомясь с NSub, вы увидите, как его API поощряет и поддерживает определенные свойства тестов (удобочитаемость, пригодность для сопровождения, стабильность и актуальность на протяжении длительного времени и т. д.), и поймете, когда изолирующий каркас хорош, а когда, напротив, становится обузой.

Поэтому в конце главы я сравню NSub с другими каркасами, доступными разработчикам на платформе .NET, с точки зрения API и их влияния на удобочитаемость, пригодность для сопровождения и стабильность, и в заключение приведу контрольный список характеристик, на которые следует обращать внимание, принимая решение об использовании каркаса для написания тестов.

Но начнем с самого начала: что такое изолирующий каркас?

5.1. Зачем использовать изолирующие каркасы?

Я начну с простого определения, которое может показаться слишком общим, но оно вынуждено быть таковым, чтобы охватить разнообразные имеющиеся каркасы.

Определение. *Изолирующий каркас* – это набор программируемых API, благодаря которым создавать поддельные объекты становится гораздо проще, быстрее и лаконичнее, чем вручную.

Хорошо спроектированный изолирующий каркас может избавить разработчика от необходимости снова и снова писать один и тот же код для утверждений или имитации взаимодействия объектов. А если каркас спроектирован *очень* хорошо, то с его помощью можно создавать тесты, которые будут работать годами, не заставляя разработчика возвращаться к ним после малейшего изменения продуктового кода.

Изолирующие каркасы существуют для большинства языков, для которых имеются каркасы автономного тестирования. Например, для C++ есть mockpp и другие каркасы, для Java – jMock и PowerMock и прочие. Для .NET существует несколько популярных каркасов: Moq, FakeItEasy, NSubstitute, Typemock Isolator и JustMock. Есть и другие, но я их больше не использую и о них не рассказываю, потому что они либо устарели, либо слишком громоздки, либо не располагают возможностями, появившимися в более поздних каркасах. Это Rhino Mocks, NMock, EasyMock, NUnit.Mocks и Moles. Каркас Moles включен в Visual Studio 2012 под названием Microsoft Fakes, но я все равно рекомендую держаться от него подальше. Более подробно все эти инструменты описываются в приложении.

У создания заглушек и подставок с помощью изолирующих каркасов, а не вручную, как в предыдущих главах, есть ряд преимуществ, которые позволяют разрабатывать более элегантные и сложные тесты быстрее, проще и с меньшим количеством ошибок.

Чтобы по-настоящему оценить полезность изолирующего каркаса, решим с его помощью какую-нибудь задачу. Одна из проблем при создании заглушек и подставок вручную – необходимость писать повторяющийся код.

Рассмотрим интерфейс, чуть более сложный, чем нам встречались до сих пор:

```
public interface IComplicatedInterface
{
  void Method1(string a, string b, bool c, int x, object o);
  void Method2(string b, bool c, int x, object o);
  void Method3(bool c, int x, object o);
}
```

На создание рукописной заглушки или подставки для такого интерфейса может уйти много времени, потому что необходимо помнить параметры каждого метода. Судите сами.

Листинг 5.1. Реализация сложного интерфейса с помощью рукописной заглушки

```
class MytestableComplicatedInterface:IComplicatedInterface
{
  public string meth1_a;
  public string meth1_b,meth2_b;
  public bool meth1_c,meth2_c,meth3_c;
  public int meth1_x,meth2_x,meth3_x;
  public int meth1_0,meth2_0,meth3_0;

  public void Method1(string a, string b, bool c,

  int x, object o)
  {
     meth1_a = a;
     meth1_b = b;
     meth1_c = c;
     meth1_x = x;
     meth1_0 = 0;
  }

  public void Method2(string b, bool c, int x, object o)
  {
     meth2_b = b;
```

На выписывание этих предложений уходит много времени

```
        meth2_c = c;
        meth2_x = x;
        meth2_0 = 0;
    }

    public void Method3(bool c, int x, object o)
    {
        meth3_c = c;
        meth3_x = x;
        meth3_0 = 0;
    }
}
```

Во-первых, вручную кодировать этот поддельный объект долго и скучно. Во-вторых, что, если вы захотите проверить, что метод вызывается несколько раз? (Напомню, что в главе 4 я определил *подделку* как нечто, похожее на настоящий объект, но не являющееся им. В зависимости от способа использования подделка может быть заглушкой или подставкой.) Или если потребуется, чтобы он возвращал значение, зависящее от полученных параметров? Или если нужно запоминать значения всех параметров при каждом вызове метода (вести историю передачи параметров)? Код очень быстро станет безобразным.

А благодаря изолирующему каркасу код, решающий все эти задачи, становится тривиальным, удобочитаемым и куда более коротким. И мы сейчас в этом убедимся, создав свою первую динамическую подставку.

5.2. Динамическое создание поддельного объекта

Дадим определение *динамического поддельного объекта* и его отличий от обычной рукописной подделки.

Определение. *Динамический поддельный объект* – это заглушка или подставка, создаваемая во время выполнения без необходимости кодировать реализацию вручную.

Динамические подделки устраняют необходимость вручную кодировать классы, реализующие интерфейсы или наследующие другим классам, поскольку нужный класс можно сгенерировать автоматически во время выполнения, написав всего несколько простых строк кода.

Теперь посмотрим, как каркас NSubstitute помогает решить некоторые из описанных выше проблем.

5.2.1. Применение NSubstitute в тестах

В этой главе я буду использовать NSubstitute (http://nsubstitute.github.com/) – бесплатный изолирующий каркас с открытым исходным кодом, который можно установить с помощью NuGet (можно скачать с сайта http://nuget.org). Я долго думал, что взять: NSubstitute или FakeItEasy. Оба хороши, поэтому прежде чем решать, на каком остановиться, рекомендую ознакомиться с тем и другим. Я приведу результаты сравнения каркасов в следующей главе и в приложении, но уже сейчас скажу, что выбрал NSubstitute, потому что он лучше документирован и поддерживает большую часть того, что должен поддерживать достойный изолирующий каркас.

Краткости ради я далее буду называть NSubstitute просто NSub. Этот каркас легко использовать, и изучение его API не займет много времени. Я рассмотрю несколько примеров, чтобы показать, как применение каркаса облегчает труд программиста (иногда). В следующей главе мы более детально изучим некоторые «метавопросы», относящиеся к изолирующим каркасам, разберемся, как они работают, и выясним, почему одни каркасы умеют делать вещи, которые другим не по силам. Но это все потом, а сейчас – за дело.

Для начала создайте библиотеку классов, которая станет проектом с автономными тестами, и добавьте ссылку на NSub, установив его с помощью NuGet (**Tools → Package Manager → Package Manager console → Install-Package NSubstitute**).

NSub поддерживает модель *подготовка-действие-утверждение* – ту самую, которой мы придерживались при написании тестов до сих пор. Идея в том, чтобы создать и настроить подделки на стадии подготовки, затем воздействовать на тестируемый продуктовый код, после чего в утверждении проверить, что подделка вызывалась.

В NSub имеется класс `Substitute`, который используется для генерации подделок во время выполнения. У этого класса есть один метод `For(type)`, предлагаемый в обычном и обобщенном исполнении; это основной способ внедрения поддельного объекта в приложение. При вызове этого метода указывается тип, для которого нужно создать поддельный экземпляр.

Метод *динамически* создает и возвращает поддельный объект, согласованный с указанным типом или интерфейсом. Вам ничего реализовывать не надо.

Поскольку NSub – ограниченный каркас, лучше всего он работает с интерфейсами. Подделать реальный класс можно, только если он не запечатан, да и в этом случае подделываются лишь виртуальные методы.

5.2.2. Замена рукописной подделки динамической

Пусть требуется поддельный объект, который проверяет, правильно ли произведена запись в журнал. Ниже приведен тестовый класс и рукописная подделка, созданная без применения изолирующего каркаса.

Листинг 5.2. Утверждение относительно рукописного поддельного объекта

```
[TestFixture]
class LogAnalyzerTests
{
  [Test]
  public void Analyze_TooShortFileName_CallLogger()
  {
     FakeLogger logger = new FakeLogger();
     LogAnalyzer analyzer = new LogAnalyzer(logger);

     analyzer.MinNameLength= 6;
     analyzer.Analyze("a.txt");

     StringAssert.Contains("слишком короткое",logger.LastError);
  }
}

class FakeLogger: ILogger
{
  public string LastError;

  public void LogError(string message)
  {
     LastError = message;
  }
}
```

Создаем подделку

Используем подделку в роли подставного объекта, поскольку о нем высказывается утверждение

Полужирным шрифтом выделены те части, которые изменятся при использовании динамических заглушек и подставок.

Теперь создадим динамический подставной объект и заменим написанный ранее тест. В следующем листинге показано, как просто

подделать ILogger и проверить, что ему была передана правильная строка.

Листинг 5.3. Подделка объекта с помощью NSub

```
[Test]
public void Analyze_TooShortFileName_CallLogger()
{
  ILogger logger = Substitute.For<ILogger>();
  LogAnalyzer analyzer = new LogAnalyzer(logger);

  analyzer.MinNameLength = 6;
  analyzer.Analyze("a.txt");

  logger.Received().LogError("Слишком короткое имя файла: a.txt");
}
```

❶ Создаем подставной объект, относительно которого в конце теста высказывается утверждение

❷ Задаем ожидаемый результат с помощью API NSub

Всего две строки – и вы избавлены от необходимости вручную писать заглушку или подставку, потому что она генерируется автоматически ❶. Динамически сгенерирован поддельный объект, который реализует интерфейс ILogger, но все методы ILogger в нем – пустышки.

Начиная с этого момента и до конца теста, все обращения к поддельному объекту автоматически запоминаются, и эту информацию можно затем проанализировать, как в последней строке ❷.

В последней строке вместо привычного утверждения используется метод расширения, предоставляемый NSub. В интерфейсе ILogger нет метода с именем Received(). С помощью этого метода мы утверждаем, что данный поддельный объект действительно вызывался (и, значит, концептуально является подставкой).

То, как метод Received() работает, наводит на мысли о чуде. Он возвращает объект того же типа, для которого был вызван, только вот его методы используются для высказывания утверждения.

Если бы мы написали в последней строке теста просто

```
logger.LogError("Слишком короткое имя файла: a.txt");
```

то поддельный объект рассматривал бы это обращение так, будто оно произведено при прогоне продуктового кода, и ровным счетом ничего не сделал бы – если, конечно, не настроить его на выполнение какого-то специального действия в методе LogError.

Вызвав Received() непосредственно перед LogError(), мы сообщаем NSub о том, что нас интересует, вызывался указанный метод объекта-подделки или нет. Если метод не вызывался, то будет воз-

буждено исключение. Читать это предложение следует так: «Нечто *получало* (received) вызов метода, иначе этот тест не прошел бы».

Если метод `LogError` не вызывался, в журнале непрошедших тестов появится примерно такое сообщение об ошибке:

```
NSubstitute.Exceptions.ReceivedCallsException : Expected to receive a call
  matching:
  LogError("Filename too short: a.txt")
Actually received no matching calls.
```

Подготовка–действие–утверждение

Обратите внимание, как точно использование изолирующего каркаса соответствует схеме подготовка–действие–утверждение. Сначала мы готовим поддельный объект, затем воздействуем на то, что тестируем, а в конце высказываем утверждение о результате.

Но не всегда было так просто.

Давным-давно (году эдак в 2006) большинство изолирующих каркасов с открытым исходным кодом не поддерживало концепцию подготовка–действие–утверждение, а работало по принципу запись–воспроизведение (record–replay).

Это был пренеприятнейший механизм, который требовал сначала сообщить каркасу, что созданный им поддельный объект находится в режиме записи, после чего следовало вызывать методы объекта так же, как это должно происходить в продуктовом коде.

Затем надо было переключить каркас в режим воспроизведения и только *потом* можно было отправить подделку в сердцевину продуктового кода.

Пример можно найти в блоге Google, посвященном тестированию: http://googletesting.blogspot.no/2009/01/tott-use-easymock.html.

В таких тестах утверждения обычно сводились просто к вызову метода каркаса `verify()` или `verifyAll()`, а бедный читатель теста должен был вернуться назад и догадаться, что же все-таки ожидалось.

Эта трагедия стоила многим разработчикам миллионов часов усердного чтения тестов в попытках выяснить, почему тест не проходит. Современная модель подготовка–действие–утверждение куда понятнее.

Если у вас имеется первое издание этой книги, то вы сможете найти пример записи–воспроизведения в разделе, посвященном каркасу Rhino Mocks. Ах, старые добрые времена! Теперь я держусь от Rhino Mocks подальше, потому что его API не так хорош, как в новых каркасах, и потому что Орен Эйни (http://Ayende.com) поставил его дальнейшее сопровождение под вопрос. Похоже, что Орен, который во многих отношениях заслужил репутацию суперпрограммиста, остепенился, женился и теперь должен расставлять приоритеты. Rhino Mocks, по-видимому, в число приоритетов не входит.

Итак, вы только что видели, как использовать подделки в роли подставок. А теперь посмотрим, как они используются в роли заглушек, имитирующих значения, получаемые тестируемой системой.

5.3. Подделка значений

В следующем листинге показано, как вернуть значение из поддельного объекта, реализующего интерфейс, в котором есть метод, возвращающий не `void`. Для этого примера мы добавим в систему интерфейс `IFileNameRules` (см. файл NSubBasics.cs в репозитории исходного кода к этой книге).

Листинг 5.4. Возврат значения из поддельного объекта

```
[Test]
public void Returns_ByDefault_WorksForHardCodedArgument()
{
  IFileNameRules fakeRules = Substitute.For<IFileNameRules>();

  fakeRules.IsValidLogFileName("strict.txt").Returns(true);

  Assert.IsTrue(fakeRules.IsValidLogFileName("strict.txt"));
}
```

Заставляет метод вернуть поддельное значение

А что, если аргумент нам безразличен? С точки зрения удобства сопровождения, было бы правильнее *всегда* возвращать определенное поддельное значение, каким бы ни был аргумент, потому что тогда при любом изменении внутренней структуры продуктового кода тест все равно пройдет, даже если продуктовый код вызывает метод несколько раз. Выиграет и понятность, потому что сейчас читатель теста не знает, важно ли конкретное имя файла. Если вы сможете облегчить ему жизнь, не заставляя усваивать ненужную информацию, то ему будет проще сопровождать ваш код.

Для этой цели каркас предоставляет сопоставители аргументов:

```
[Test]
public void Returns_ByDefault_WorksForHardCodedArgument()
{
  IFileNameRules fakeRules = Substitute.For<IFileNameRules>();

  fakeRules.IsValidLogFileName(Arg.Any<String>()).Returns(true);

  Assert.IsTrue(fakeRules.IsValidLogFileName("anything.txt"));
}
```

Игнорировать значение аргумента

Обратите внимание, как класс `Arg` используется для того, чтобы сообщить, что это поддельное значение следует возвращать при любом входном аргументе. Такие *сопоставители аргументов* широко используются в изолирующих каркасах для задания порядка обработки аргументов.

А что, если требуется имитировать исключение? В NSub это делается так:

```
[Test]
public void Returns_ArgAny_Throws()
{
  IFileNameRules fakeRules = Substitute.For<IFileNameRules>();

  fakeRules.When(x => x.IsValidLogFileName(Arg.Any<string>()))
     .Do(context => { throw new Exception("fake exception"); });

  Assert.Throws<Exception>(() =>
     fakeRules.IsValidLogFileName("anything"));
}
```

Здесь лямбда-выражение обязательно

Обратите внимание, как с помощью `Assert.Throws` проверяется, что исключение действительно возбуждалось.

Я не в восторге от синтаксических наворотов NSub. В FakeItEasy добиться того же результата было бы проще, но NSub лучше документирован, потому я его и выбрал.

Отметим, что лямбда-выражения здесь обязательны. В методе `When` аргумент `x` обозначает поддельный объект, поведение которого мы изменяем. В методе `Do` обратите внимание на аргумент `context` типа `CallInfo`. Во время выполнения `context` содержит значения аргументов и позволяет делать удивительные вещи, но в данном примере это не нужно.

Научившись имитировать различное поведение, перейдем к более практическим вопросам и посмотрим, что же у нас имеется.

5.3.1. Встретились в тесте подставка, заглушка и священник

Воспользуемся в одном тесте двумя видами поддельных объектов: заглушкой и подставкой.

В исходном коде к книге найдите пример Analyzer2 в папке для главы 5. Он похож на пример в листинге 4.2 из главы 4, где речь шла об использовании в `LogAnalyzer` классов `MailSender` и `WebService`, но на этот раз требование такое: если объект-регистратор возбуждает

исключение, то об этом нужно уведомить веб-службу. Схема взаимодействия показана на рис. 5.1.

Мы хотим убедиться, что если регистратор возбуждает исключение, то `LogAnalyzer2` уведомит `WebService` о проблеме.

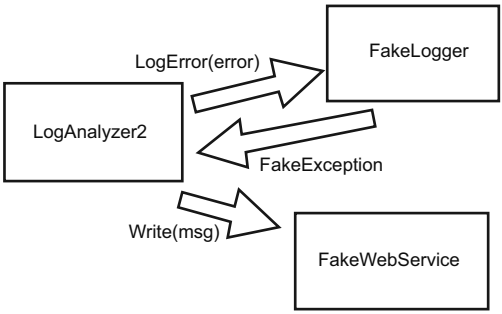


Рис. 5.1. Вместо регистратора мы используем заглушку, имитирующую исключение, а вместо веб-службы – подставку, контролирующую правильность обращения. В целом тестируется взаимодействие `LogAnalyzer2` с другими объектами

В листинге ниже показан как продуктовый тест, так и его тесты.

Листинг 5.5. Тестируемый метод и тест с рукописными подставкой и заглушкой

```
[Test]
public void Analyze_LoggerThrows_CallsWebService()   <-- Тест
{
  FakeWebService mockWebService = new FakeWebService();

  FakeLogger2 stubLogger = new FakeLogger2();
  stubLogger.WillThrow = new Exception("fake exception");

  var analyzer2 = new LogAnalyzer2(stubLogger, mockWebService);

  analyzer2.MinNameLength = 8;
  string tooShortFileName="abc.ext";
  analyzer2.Analyze(tooShortFileName);

  Assert.That(mockWebService.MessageToWebService,
         Is.StringContaining("fake exception"));
}

public class FakeWebService:IWebService   <-- Поддельная веб-служба, выступающая в роли подставки
{
  public string MessageToWebService;

  public void Write(string message)
```

```
   {
      MessageToWebService = message;
   }
}

public class FakeLogger2:ILogger
{
  public Exception WillThrow = null;
  public string LoggerGotMessage = null;

  public void LogError(string message)
  {
     LoggerGotMessage = message;
     if (WillThrow != null)
     {
        throw WillThrow;
     }
  }
}

//---------- Продуктовый код
public class LogAnalyzer2
{
  private ILogger _logger;
  private IWebService _webService;

  public LogAnalyzer2(ILogger logger,IWebService webService)
  {
     _logger = logger;
     _webService = webService;
  }

  public int MinNameLength { get; set; }

  public void Analyze(string filename)
  {
     if (filename.Length<MinNameLength)
     {
        try
        {
           _logger.LogError(
             string.Format("Слишком короткое имя файла: {0}",filename));
        }
        catch (Exception e)
        {
           _webService.Write("Ошибка регистратора: " + e);
        }
     }
  }
}

public interface IWebService
```

Поддельный регистратор, выступающий в роли заглушки

Тестируемый класс

```
{
  void Write(string message);
}
```

А ниже показано, как тот же тест выглядит при использовании NSubstitute.

Листинг 5.6. Предыдущий тест, переделанный с применением NSubstitute

```
[Test]
public void Analyze_LoggerThrows_CallsWebService()
{
  var mockWebService = Substitute.For<IWebService>();
  var stubLogger = Substitute.For<ILogger>();
  stubLogger.When(
     logger => logger.LogError(Arg.Any<string>()))
     .Do(info => { throw new Exception("fake exception");});

  var analyzer = new LogAnalyzer2(stubLogger, mockWebService);

  analyzer.MinNameLength = 10;
  analyzer.Analyze("Short.txt");

  mockWebService.Received()
     .Write(Arg.Is<string>(s => s.Contains("fake exception")));
}
```

Имитируем исключение при любых входных данных

Проверяем, что была вызвана подставная веб-служба с аргументом, содержащим строку "fake exception"

В этом тесте хорошо то, что не нужны никакие рукописные подделки, однако обратите внимание, какой урон это наносит удобочитаемости. На мой вкус, все эти лямбда-выражения выглядят не слишком дружелюбно, но это то небольшое зло, с которым приходится мириться, если хочешь работать на C#, потому что именно они позволяют не именовать методы явно, а, значит, упростить рефакторинг в случае, если впоследствии имя метода изменится.

Отметим, что ограничения в сопоставителях аргументов можно использовать как на стадии подготовки, где настраивается заглушка, так и на стадии утверждения, где проверяется, вызывалась ли подставка.

В NSubstitute существуют и другие сопоставители аргументов, все они хорошо документированы на сайте. Эта книга не является руководством по NSub (в конце концов, для чего-то же Бог создал онлайновую документацию), поэтому если вы хотите больше узнать об API, ступайте на страницу http://nsubstitute.github.com/help/argument-matchers/.

Сравнение объектов и свойств

Что происходит, когда мы ожидаем получить в качестве аргумента объект с определенными свойствами? Например, что, если в обращении к методу `webservice.Write` передан объект `ErrorInfo`, в котором заданы свойства `severity` и `message`?

```
[Test]
public void Analyze_LoggerThrows_CallsWebServiceWithNSubObject()
{
  var mockWebService = Substitute.For<IWebService>();
  var stubLogger = Substitute.For<ILogger>();
  stubLogger.When(logger => logger.LogError(Arg.Any<string>()))
      .Do(info => { throw new Exception("fake exception");});

  var analyzer = new LogAnalyzer3(stubLogger, mockWebService);

  analyzer.MinNameLength = 10;
  analyzer.Analyze("Short.txt");

  mockWebService.Received()
     .Write(Arg.Is<ErrorInfo>(info => info.Severity == 1000
           && info.Message.Contains("fake exception")));
}
```

Строго типизированный сопоставитель аргументов сравнивает значение свойства с учетом типа

Обычный оператор C# «и» служит для создания более сложного ожидания

Обратите внимание на использование стандартных конструкций C# для создания сложных сопоставителей, проверяющих один и тот же аргумент. Мы проверяем, что в переданном аргументе заданы вполне определенная серьезность (severity) *и* конкретное сообщение.

Также отметим, как это отражается на удобочитаемости. Вообще говоря, я замечаю, что чем больше я использую изолирующие каркасы, тем менее читаемым становится код теста, но иногда результат получается все же приемлемым. Тут мы имеем как раз пограничный случай. Но если бы в одном утверждении оказалось больше одного лямбда-выражения, я задумался бы, не будет ли рукописная подделка понятнее.

Впрочем, если вы хотите проверить что-то самым простым способом, то можно сравнить два объекта и посмотреть, не получится ли код понятнее. Создайте объект `expected` с ожидаемыми свойствами и сравните его с фактически полученным объектом, как показано ниже.

Листинг 5.7. Сравнение объектов целиком

```
[Test]
public void Analyze_LoggerThrows_CallsWebServiceWithNSubObjectCompare()
{
  var mockWebService = Substitute.For<IWebService>();
  var stubLogger = Substitute.For<ILogger>();
  stubLogger.When(logger => logger.LogError(Arg.Any<string>()))
       .Do(info => { throw new Exception("fake exception");});

  var analyzer = new LogAnalyzer3(stubLogger, mockWebService);

  analyzer.MinNameLength = 10;
  analyzer.Analyze("Short.txt");

  var expected = new ErrorInfo(1000, "fake exception");
  mockWebService.Received().Write(expected);
}
```

Создаем объект, который ожидаем получить

Утверждаем, что получили в точности такой же объект (в действительности вызываем Assert.Equals())

Проверка объекта целиком работает при выполнении следующих условий:

- создать объект с ожидаемыми свойствами просто;
- мы хотим проверять *все* свойства объекта;
- точно известны значения каждого свойства;
- в классе сравниваемых объектов корректно реализован метод `Equals()`. (Вообще говоря, не стоит полагаться на готовую реализацию метода `object.Equals()`. Если вы не реализовали метод `Equals()` самостоятельно, то этот тест никогда не пройдет, потому что реализация `Equals()` по умолчанию вернет `false`.)

И еще одно замечание о стабильности данного теста. Поскольку эта техника не позволяет спросить, содержит ли значение некоторого свойства некоторую подстроку, то тесты оказываются несколько более хрупкими в плане будущих изменений.

Кроме того, если впоследствии ожидаемое строковое значение свойства изменится, пусть даже всего на один лишний пробел в начале или в конце, то тест не пройдет и придется сопоставлять значение с новой строкой. Искусство состоит в том, чтобы решить, в какой мере вы готовы пожертвовать удобочитаемостью ради стабильности. Лично мне кажется, что если свойств немного, то их сравнение с помощью сопоставителей аргументов чуть более предпочтительно, чем сравнение объектов целиком, поскольку так мы

получаем большую стабильность. Я терпеть не могу изменять тесты без веских причин.

5.4. Тестирование операций, связанных с событием

События – это улица с двусторонним движением, и тестировать их можно в двух направлениях:

- проверить, что кто-то прослушивает событие;
- проверить, что кто-то генерирует событие.

5.4.1. Тестирование прослушивателя события

Сначала рассмотрим сценарий, который, на мой взгляд, многие разработчики плохо реализуют в тестах: проверка того, что один объект зарегистрирован в качестве получателя события от другого объекта.

Часто разработчики выбирают неудобный для сопровождения и избыточно специфицированный способ: проверяют внутреннее состояние объекта-получателя.

Такую реализацию я никому не рекомендую в реальных тестах. Подписка на событие – внутреннее закрытое поведение кода. Само по себе оно не может считаться конечным результатом, а приводит к изменению состояния, а, значит, и поведения системы.

Для проверки лучше убедиться, что объект-прослушиватель действительно что-то делает в ответ на сгенерированное событие. Если прослушиватель не подписался на событие, то никакого видимого изменения в поведении не будет, что иллюстрируется в следующем листинге.

Листинг 5.8. Как вызвать выполнение кода обработки события

```
class Presenter
{
  private readonly IView _view;

  public Presenter(IView view)
  {
     _view = view;
     this._view.Loaded += OnLoaded;
```

```
    }

    private void OnLoaded()
    {
        _view.Render("Hello World");
    }
}

public interface IView
{
    event Action Loaded;
    void Render(string text);
}

//------ ТЕСТЫ
[TestFixture]
public class EventRelatedTests
{
    [Test]
    public void ctor_WhenViewIsLoaded_CallsViewRender()
    {
        var mockView = Substitute.For<IView>();

        Presenter p = new Presenter(mockView);
        mockView.Loaded += Raise.Event<Action>();
        mockView.Received()
          .Render(Arg.Is<string>(s => s.Contains("Hello World")));
    }
}
```

Генерируем событие с помощью NSubstitute

Проверяем, что объект view вызывался

Отметим следующие моменты.

- Подставка является также заглушкой (мы имитируем событие).
- Чтобы сгенерировать событие в тесте, на него надо подписаться. Конструкция выглядит нелепо, но она необходима, чтобы компилятор не ругался, поскольку свойства, относящиеся к событиям, обрабатываются специальным образом и бдительно охраняются компилятором. Напрямую генерировать событие может только класс или структура, в которой событие объявлено.

Ниже рассмотрен еще один сценарий, в котором есть две зависимости: регистратор и представление. Тест в следующем листинге проверяет, что объект `Presenter` пишет в журнал после получения события ошибки от нашей заглушки.

Листинг 5.9. Имитация события и другая подставка

```
[Test]
public void ctor_WhenViewhasError_CallsLogger()
{
  var stubView = Substitute.For<IView>();
  var mockLogger = Substitute.For<ILogger>();

  Presenter p = new Presenter(stubView, mockLogger);
  stubView.ErrorOccured +=
       Raise.Event<Action<string>>("fake error");

  mockLogger.Received()
       .LogError(Arg.Is<string>(s => s.Contains("fake error")));
}
```

❶ Имитируем ошибку

❷ С помощью подставки проверяем, что была запись в журнал

Отметим, что для генерации события используется заглушка ❶, а для проверки того, что служба получила сообщение, – подставка ❷.

Теперь рассмотрим противоположную сторону сценария тестирования. Мы хотим проверить, что источник действительно генерирует событие в нужное время. В следующем разделе показано, как это делается.

5.4.2. Тестирование факта генерации события

Самый простой способ протестировать событие – вручную подписаться на него в тестовом методе, воспользовавшись анонимным делегатом. Ниже приведен пример.

Листинг 5.10. Использование анонимного делегата для подписки на событие

```
[Test]
public void EventFiringManual()
{
  bool loadFired = false;
  SomeView view = new SomeView();
  view.Load+=delegate
  {
     loadFired = true;
  };

  view.DoSomethingThatEventuallyFiresThisEvent();

  Assert.IsTrue(loadFired);
}
```

Делегат просто запоминает, что событие имело место. Я решил использовать делегат, а не лямбда-выражение, потому что такая запись кажется мне более понятной. У делегата могут быть также параметры, которые тоже можно запомнить и затем проверить в утверждении.

Далее мы дадим краткий обзор изолирующих каркасов на платформе .NET.

5.5. Современные изолирующие каркасы для .NET

NSub, конечно, не единственный имеющийся изолирующий каркас. В неформальном опросе, проведенном в августе 2012 года, я задавал читателям своего блога вопрос: «Каким изолирующим каркасом вы пользуетесь?» Результаты представлены на рис. 5.2.

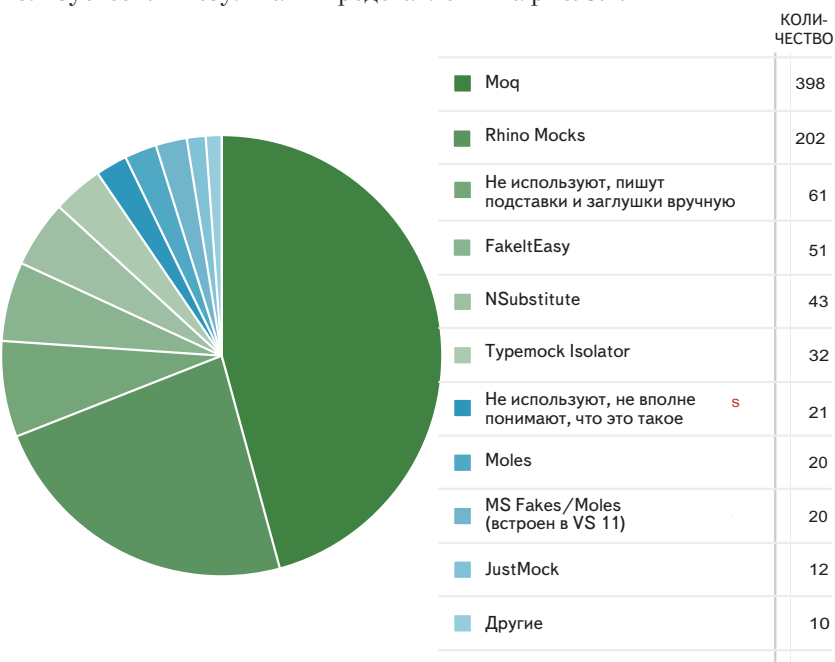


Рис. 5.2. Распространенность изолирующих каркасов среди читателей моего блога

Каркас Moq, который на момент предыдущего издания был новичком, теперь выбился в лидеры, тогда как Rhino Mocks отстал и

продолжает терять позиции (в основном, потому что активная разработка прекратилась). Кроме того, по сравнению с первым изданием увеличилось число конкурентов – вдвое. Это кое-что говорит о зрелости сообщества в плане осознания потребности в тестировании и изоляции, и мне отрадно это видеть.

Каркас FakeItEasy, о котором его создатель, наверное, даже не помышлял, когда вышло первое издание этой книги, теперь стал сильным конкурентом в тех аспектах, которые мне нравятся в NSubstitute, и я горячо рекомендую поэкспериментировать с ним. Эти аспекты (точнее сказать, полезные качества) перечислены в следующей главе, где мы углубимся во внутренние механизмы работы изолирующих каркасов.

Лично я не пользуюсь Moq из-за плохих сообщений об ошибках и слишком частого употребления слова «mock» в API. Это только вводит в заблуждение, потому что «подставки» используются также для создания заглушек.

Вообще говоря, я считаю правильным выбрать какой-то один каркас и по возможности сохранять ему верность – это повышает удобочитаемость и упрощает обучение членов команды.

В приложении к книге я рассматриваю все упомянутые каркасы более детально и объясняю, чем они мне нравятся или не нравятся.

Суммируем преимущества изолирующих каркасов по сравнению с рукописными подставками. А затем обсудим, на что обращать внимание при использовании изолирующих каркасов.

Почему не следует использовать строковые имена методов в тестах

Во многих каркасах за пределами мира .NET принято с помощью строк описывать методы, поведение которых мы собираемся изменить. Почему это плохо?

Если мы изменим имя метода в продуктовом коде, то все тесты, в которых этот метод используется, будут по-прежнему компилироваться, а об ошибке сообщат только во время выполнения, возбудив исключение из-за того, что метод не найден.

При использовании строго типизированных имен методов (благодаря лямбда-выражениям и делегатам) изменение имени не приводит к проблемам, потому что метод определен прямо в тесте. После любого несогласованного изменения тест просто перестанет компилироваться, и мы сразу узнаем, что в нем имеется ошибка.

Автоматизированные средства рефакторинга, в том числе встроенные в Visual Studio, упрощают переименование методов, но строки в исходном коде при этом как правило все равно не анализируются. (ReSharper для .NET в этом смысле исключение. Он исправляет также и строки, но это лишь частичное решение, которое к тому же в некоторых случаях само может стать причиной ошибок.)

5.6. Достоинства и подводные камни изолирующих каркасов

Из сказанного в этой главе очевидны следующие достоинства изолирующих каркасов.

- *Упрощается проверка параметров.* Использование рукописных подставок для проверки того, что методу переданы правильные параметры, – утомительный процесс, требующий времени и терпения. В большинстве изолирующих каркасов проверка переданных параметров, даже если их много, – тривиальная процедура.
- *Упрощается контроль нескольких обращений к методу.* При использовании рукописных подставок трудно проверить, что один и тот же метод вызывался несколько раз с правильными параметрами. Как мы увидим ниже, в изолирующих каркасах эта проверка также тривиальна.
- *Упрощается создание поддельных объектов.* Изолирующие каркасы позволяют легко создавать как заглушки, так и подставки.

5.6.1. Каких подводных камней избегать при использовании изолирующих каркасов

Несмотря на все достоинства изолирующих каркасов, существуют и опасности, например, безудержное использование каркасов в случаях, когда вполне хватило бы и рукописной подставки; создание тестов, которые трудно читать из-за чрезмерного изобилия подставок, или недостаточно тщательное разделение тестов.

Вот перечень моментов, на которые следует обращать особое внимание:

- неудобочитаемый тестовый код;
- проверка не того, что надо;
- наличие более одной подставки в одном тесте;
- избыточное специфицирование теста.

Рассмотрим каждый вопрос более детально.

5.6.2. Неудобочитаемый тестовый код

Наличие подставки в тесте и так уже делает его менее понятным, но все же не настолько, чтобы сторонний читатель не мог разобраться, что в нем происходит. Наличие нескольких подставок или нескольких ожиданий в одном тесте может снизить удобочитаемость до такого уровня, что будет трудно не только сопровождать его, но хотя бы понять, что именно тестируется.

Если вы замечаете, что тест становится неудобочитаемым или за его кодом трудно следить, попробуйте убрать некоторые подставки или ожидания либо разделить тест на несколько меньших и более понятных.

5.6.3. Проверка не того, что надо

Подставные объекты позволяют проверить, что вызывались методы, объявленные в интерфейсе, но это еще не означает, что вы тестируете то, что надо. Проверка того, что объект подписан на событие, ничего не говорит о функциональности объекта. Для тестирования этого объекта лучше бы проверить, что после возникновения события произошло что-то значимое.

5.6.4. Наличие более одной подставки в одном тесте

Считается правильным тестировать в каждом тесте только один аспект, в противном случае может возникнуть путаница и трудности с сопровождением теста. Наличие двух подставок в одном тесте – то же самое, что тестирование нескольких конечных результатов одной и той же единицы работы. Если вы не можете придумать тесту имя, потому что он делает слишком много сразу, то следует разделить его на несколько тестов.

5.6.5. Избыточное специфицирование теста

Старайтесь обходиться без подставных объектов. Тест, в котором нет утверждений относительно вызова какого-то объекта, проще читать и сопровождать. Да, бывают случаи, когда цели можно достичь только с помощью подставок, но их не должно быть слишком много.

Если более 5 % ваших тестов завязаны на подставки (не заглушки), то, вероятно, вы увлеклись избыточным специфицированием вместо проверки изменения состояния или возвращаемых значений. Но даже и в 5 % тестов, где используются подставные объекты, можно перегнуть палку.

Тест, в котором слишком много ожиданий (`x.received().X()` и `X.received().Y()` и т. д.), может оказаться очень хрупким и отказать при малейшем изменении продуктового кода, хотя функциональность последнего не нарушена.

Тестирование взаимодействий – палка о двух концах: если проверяется слишком много, то за деревьями становится трудно рассмотреть лес – общую функциональность, а если слишком мало, то можно упустить из виду существенные взаимодействия объектов.

Приведу несколько рекомендаций по поиску компромисса.

- По возможности используйте нестрогие подставки (что такое строгие и нестрогие подставки, объясняется в следующей главе). Тогда тест будет реже отказывать из-за неожиданных вызовов методов. Это помогает, когда закрытые методы в продуктовом коде продолжают изменяться.
- По возможности используйте заглушки вместо подставок. Если подставки встречаются более чем в 5 % тестов, то это, скорее всего, перебор. Заглушки можно вставлять повсюду, с подставками надо быть скромнее. Тестировать следует только один сценарий в каждом тесте. Чем больше подставок, тем больше проверок будет в конце теста, но важной обычно является только одна. Остальные – просто шум, затемняющий смысл текущего тестового сценария.
- Избегайте заглушек в роли подставок, если это в человеческих силах. Используйте заглушку только для подделывания значений, возвращаемых тестируемой программе, или для возбуждения исключений. Не проверяйте, вызывались ли какие-то методы заглушки. Чтобы проверить факт обращения к методу, используйте подставку, но возвращать значения в тес-

тируемую программу она не должна. По большей части можно избежать использования подставок, одновременно выступающих в роли заглушек, но не всегда (как мы видели в примере, касающемся событий, выше в этой главе).

5.7. Резюме

Изолирующие каркасы – отличная штука, и вы обязательно должны научиться с ними работать. Но по возможности предпочитайте тестирование на основе проверки возвращаемых значений и изменения состояния (в противоположность тестированию взаимодействий), поскольку в этом случае в тестах содержится минимум предположений о внутренних деталях реализации. К подставкам следует прибегать только тогда, когда нет никакого другого способа протестировать реализацию, поскольку в результате получаются тесты, которые труднее сопровождать.

Если подставки (не заглушки) встречаются более чем в 5 % тестов, то, скорее всего, вы увлеклись избыточным специфицированием.

Изучайте продвинутые средства изолирующих каркасов, в частности NSub, – и тест сможет достоверно сказать, что происходит, а чего не происходит в программе. Нужно лишь, чтобы код был тестопригодным.

Можно также навредить себе, создавая избыточно специфицированные тесты, которые трудно читаются и легко ломаются. Умение принять правильное решение о том, когда использовать динамические, а когда рукописные подставки – это искусство. Мой совет: если код с применением изолирующего каркаса начинает казаться безобразным, значит, пора что-то упрощать. Напишите подставку вручную или проверьте другой результат, который доказывает то же самое, но проще.

Если, несмотря на все усилия, код все равно трудно протестировать, то остается три варианта: воспользоваться суперкаркасом типа Typemock Isolator (рассматривается в следующей главе), изменить проект или бросать работу.

Изолирующие каркасы могут существенно упростить процесс тестирования и помочь в создании тестов, удобных для чтения и сопровождения. Но важно понимать, что иногда они не столько помогают, сколько мешают разработке. Например, при работе с унаследованным кодом, возможно, придется воспользоваться другим каркасом, с иными возможностями. Для каждой работы нужен свой инструмент,

поэтому обдумывая, как подойти к решению конкретной проблемы тестирования, обязательно оценивайте общую картину.

В следующей главе мы глубже копнем изолирующие каркасы, рассмотрев их внутреннее устройство и его влияние на спектр предоставляемых возможностей.

ГЛАВА 6. Внутреннее устройство изолирующих каркасов

В этой главе:

- Ограниченные и неограниченные каркасы.
- Как работает неограниченный каркас на основе профилировщика.
- Определение полезных качеств хорошего изолирующего каркаса.

В предыдущей главе мы использовали каркас NSubstitute для создания поддельных объектов. В этой главе мы отступим назад и представим общую картину изолирующих каркасов на платформе .NET и за ее пределами. Мир изолирующих каркасов велик и, решая, какой выбрать, нужно учитывать разные факторы.

Начнем с простого вопроса: почему у одних каркасов возможности шире, чем у других? Например, одни каркасы умеют подделывать статические методы, а другие – нет. Некоторые каркасы даже способны подделывать объекты, которые еще не были созданы, другие пребывают в блаженном неведении о таких подвигах. С чего бы это?

6.1. Ограниченные и неограниченные каркасы

Изолирующие каркасы в .NET (а также в Java, C++ и других статических языках) можно отнести к двум основным группам – по способности делать определенные вещи на языке программирования.

Я называю эти архетипы *ограниченным* и *неограниченным*.

6.1.1. Ограниченные каркасы

В .NET к ограниченным относятся каркасы Rhino Mocks, Moq, NMock, EasyMock, NSubstitute и FakeItEasy, в Java – jMock и EasyMock.

Я называю их *ограниченными*, потому что есть такие вещи, которые они неспособны подделать. Что именно каркас не может подделать, зависит от того, на какой платформе он работает и как ее использует.

В .NET ограниченные каркасы не умеют подделывать статические методы, невиртуальные методы, неоткрытые методы и т. д.

Почему так? Ограниченный изолирующий каркас работает так же, как рукописная подделка: генерирует код и компилирует его во время выполнения, поэтому он не может сделать больше того, что позволяет компилятор и промежуточный язык. В Java компилятор и результирующий байт-код эквивалентны. В C++ ограничения налагаются возможностями языка.

Ограниченный каркас обычно генерирует во время выполнения классы, наследующие или переопределяющие методы интерфейсов или базовых классов, – так же, как мы сами; только мы это делаем до выполнения программы, а каркасы – во время выполнения. Это означает, что каркас связан теми же ограничениями, налагаемыми компилятором: подделываемый код должен быть либо интерфейсом, либо открытым и допускающим наследование (незапечатанным) классом с открытым конструктором. В случае базовых классов переопределяемые методы должны быть виртуальными.

Все это означает, что при работе с ограниченными каркасами мы связаны такими же правилами компилятора, как и обычный код. Статические методы, закрытые методы, запечатанные классы, классы с закрытыми конструкторами и прочее в том же роде – все это нам не подделать.

6.1.2. Неограниченные каркасы

В .NET к *неограниченным* относятся каркасы Typemock Isolator, JustMock и Moles (известный также под названием MS Fakes), в Java – PowerMock и JMockit, в C++ – Isolator++ и Hippo Mocks. Неограниченные каркасы не генерируют и не компилируют во время выполнения код, наследующий каким-то классам. Они применяют для достижения цели совсем другие средства – какие именно, зависит от платформы.

Прежде чем объяснять, как это работает в .NET, хочу предупредить, что в этой главе мы копнем довольно глубоко. К автономному

тестированию излагаемый материал не имеет прямого отношения, но позволит вам понять, почему некоторые вещи устроены именно так, а не иначе, и на базе этих знаний принимать обоснованные решения о проектировании автономных тестов и выбирать курс действий.

В .NET все неограниченные каркасы построены на основе профилировщика. Это означает, что они пользуются рядом неуправляемых API профилирования, которые обертывают работающий экземпляр CLR (общеязыковой среды выполнения). Прочитать об этом можно на странице http://msdn.microsoft.com/en-us/library/bb384493.aspx#profiling_api. API профилирования включает события, извещающие обо всем, что происходит, когда CLR выполняет код, и даже о том, что случается, до того как промежуточный код .NET (IL) компилируется в двоичный в памяти. Некоторые из этих событий позволяют вставить новый или изменить существующий IL-код перед компиляцией, т. е. расширить функциональные возможности существующего кода. Многие инструменты, от профилировщика ANTS до профилировщиков памяти, пользуются API профилирования. Typemock Isolator стал первым каркасом (больше семи лет назад), авторы которого осознали потенциал API профилирования в части изменения поведения «поддельных» объектов.

Поскольку события профилирования генерируются для всех частей кода, в том числе статических методов, закрытых конструкторов и даже не принадлежащего вам стороннего кода, например SharePoint, неограниченный каркас может расширить или изменить поведение любого кода, какой пожелает, – в любом классе, в любой библиотеке, в том числе и той, которую вы не компилировали. Возможности безграничны. Более подробно я рассматриваю различия между каркасами на основе профилировщика в приложении.

В .NET, для того чтобы включить профилирование и прогнать тесты, написанные для каркаса на основе профилировщика, необходимо задать некоторые переменные окружения, читаемые исполняемым процессом. По умолчанию они не заданы, поэтому код не профилируется. Чтобы подключить профилировщик к процессу прогона тестов, задайте `Cor_Enable_Profiling=0x1` и `COR_PROFILER=НЕКИЙ_GUID` (да, в каждый момент времени присоединить можно только один профилировщик).

Для каркасов Moles, Typemock и JustMock имеются специальные надстройки над Visual Studio, которые задают эти переменные окружения и позволяют прогонять тесты. Обычно к таким инструментам

прилагается специальная консольная программа, которая запускает другие программы, предварительно установив эти переменные.

Если вы попытаетесь прогнать тесты, не включив профилировщик, то исполнитель тестов будет выводить странные сообщения об ошибках в окно вывода. Я вас предупредил. Изолирующий каркас может, например, сообщить, что ничего не записано или что ни один тест не выполнен.

У неограниченных изолирующих каркасов есть ряд преимуществ.

- Мы можем написать автономные тесты для кода, который раньше было невозможно протестировать; нужно лишь подделать все окружение единицы работы, изолировав ее; как-то изменять код при этом не придется. Позже, уже имея тесты, можно будет заняться рефакторингом.
- Можно подделать сторонние системы, которые мы не контролируем и которые серьезно осложняют тестирование, например, если наши классы наследуют базовому классу, который находится в сторонней библиотеке и имеет много низкоуровневых зависимостей (к таковым относятся, к примеру, SharePoint, CRM, Entity Framework и Silverlight).
- Можно выбрать собственный уровень проектирования, не связывая себя по рукам и ногам конкретными паттернами. Дизайн создают не инструменты, а люди. Если вы не понимаете, что делаете, никакой инструмент не поможет. Я еще вернусь к этому вопросу в главе 11.

Но у неограниченных изолирующих каркасов есть и недостатки.

- Если вы недостаточно аккуратны, то можете загнать себя в угол, подделывая всякие ненужные вещи, вместо того чтобы поискать истинную единицу работы на более высоком уровне.
- При небрежной работе некоторые тесты могут стать несопровождаемыми, потому что вы подделали API, которыми не владеете. Такое случается, но не так часто, как вы думаете. Судя по моему опыту, крайне маловероятно, что подделанный в каркасе низкоуровневый API изменится в будущем. Чем глубже API, тем больше уровней над ним надстроено и тем маловероятнее, что кто-то рискнет его изменить.

Далее мы рассмотрим механизмы, которые дают неограниченным каркасам возможность совершать эти невероятные подвиги.

6.1.3. Как работают неограниченные каркасы на основе профилировщика

Этот раздел относится только к платформе .NET и CLR, потому что лишь там имеется API профилирования. Излагаемый материал будет интересен только читателям, которые хотят знать о мельчайших деталях. Для создания хороших автономных тестов это необязательно, но даст дополнительные очки, если вы когда-нибудь захотите написать конкурирующий каркас. В Java и в C++ для достижения аналогичных целей применяются совсем другие способы.

В .NET в инструментах типа Typemock Isolator имеется написанный на C++ код, который подключается к COM-интерфейсу API профилирования в CLR и подписывается на обратные вызовы, относящиеся к различным специальным событиям. Typemock даже владеет патентом на эту технологию (см. http://bit.ly/typemockpatent), Правда, компания, похоже, не требует его неукоснительного соблюдения, иначе не было бы конкурентов в лице JustMock и Moles.

События `JitCompilationStarted` и `SetILFunctionBody`, являющиеся членами COM-интерфейса `ICorProfilerCallback2`, позволяют во время выполнения получать и изменять подлежащий исполнению IL-код еще *до* его компиляции в двоичный. Этот IL-код можно подменить своим собственным. Typemock и подобные ему инструменты добавляют IL-вставки до и после каждого метода, до которого могут дотянуться. По сути дела, эти вставки обращаются к управляемому коду на C# и проверяют, установил ли кто-то специальное поведение для данного метода. Можно рассматривать эту процедуру как глобальную аспектно-ориентированную сквозную проверку поведения всех методов в вашем коде. Внедренные IL-вставки обращаются также к различным точкам подключения (написанным на управляемом языке, как правило на C#, и находящимся в самом сердце изолирующего каркаса) – к каким именно, зависит от поведения, установленного пользователем API каркаса (например, «возбудить исключение» или «вернуть поддельное значение»).

В .NET любой код подвергается JIT-компиляции (если только не был предварительно откомпилирован в машинный код с помощью NGen.exe). Это относится не только к вашему, но и к чужому коду, в том числе коду самого каркаса .NET Framework, SharePoint и других библиотек.

Это означает, что Typemock может внедрить IL-код куда захочет, даже внутрь .NET Framework. Вставки можно поместить до или после

любого метода, в том числе написанного не вами, именно поэтому такие каркасы – дар господний для тестирования унаследованного кода, который вы не можете переработать.

Примечание. API профилирования не очень хорошо документирован (специально?). Но если поискать в Google `JitCompilationStarted` и `SetILFunctionBody`, то вы найдете много ссылок и историй, которые могут стать путеводной нитью при построении собственного неограниченного изолирующего каркаса для .NET. Приготовьтесь к долгому и трудному путешествию и изучайте C++. Не забудьте взять в дорогу бутылочку виски.

Разные каркасы задействуют разные возможности профилировщика

В принципе, *всем* изолирующим каркасам на основе профилировщика доступны одни и те же базовые возможности. Но на практике функциональность основных каркасов в .NET различается. В каждом из каркасов большой тройки – JustMock, Typemock и MS Fakes – реализовано некоторое подмножество функций.

Примечание. Я употребляю названия Typemock и Typemock Isolator как синонимы, потому что так принято называть продукт Isolator.

Typemock, имея самую долгую историю, поддерживает почти все конструкции, которые могли бы показаться нетестопригодными в унаследованном коде, в том числе будущие значения, статические конструкторы и прочие странные создания. Не умеет он только подделывать API из библиотеки mscorlib.dll, где находятся такие важнейшие классы, как DateTime, и пространства имен System.String и System.IO. Для этой конкретной DLL (и только для нее) Typemock поддерживает не все API, а только часть.

Технически Typemock мог бы подделать и все типы из этой библиотеки, но это бессмысленно по соображениям производительности. Представьте, что вы захотели подменить все строки в системе, возвращая вместо них поддельные значения. Примите во внимание, сколько раз каждая строка используется в базовом API .NET Framework, и помножьте это число на одну-две проверки на каждое обращение внутри Typemock, необходимые, чтобы понять, нужно подделывать данное действие или нет, – и вы поймете, что с производительностью можно распроститься.

За исключением некоторых базовых типов .NET Framework, Typemock готов подделать практически все, что вы ему подбросите.

У каркаса MS Fakes есть одно преимущество над Typemock Isolator. Он разработан внутри Microsoft, первоначально как дополнение к другому инструменту – Pex (описан в приложении). Понятно, что сотрудники Microsoft лучше осведомлены обо всех тонкостях плохо документированного API профилирования, поэтому встроили поддержку некоторых типов, которые даже Typemock Isolator не позволяет подделать. С другой стороны, API каркаса MS Fakes не содержит большинства средств для работы с унаследованным кодом, которые имеются в Isolator или JustMock и которых было бы естественно ожидать от каркаса с такими возможностями. Этот API в основном разрешает подменять открытые методы (статические и нестатические) собственными делегатами, но без дополнительных надстроек не позволяет подделывать неоткрытые методы.

С точки зрения API и того, что можно подделывать, JustMock по своим возможностям очень близок к Typemock Isolator, но пока ему недостает некоторых средств, нужных при работе с унаследованным кодом, в том числе возможности подделывать статические конструкторы и закрытые методы. Объясняется это, главным образом, недостаточной зрелостью. MS Fakes и JustMock существуют всего-то года три. Typemock опережает их на 3–4 года.

Пока важно уяснить, что выбирая изолирующий каркас, вы вместе с ним выбираете и набор возможностей или ограничений.

Примечание. За использование каркасов на основе профилировщиков приходится расплачиваться некоторой потерей производительности. Поскольку при вызове каждого метода добавляются обращения к написанному вами коду, то работа замедляется. Возможно, вы начнете это замечать, только когда число тестов перевалит за несколько сотен, но замедление есть, и его нельзя назвать пренебрежимо малым. Впрочем, я считаю это небольшой ценой за те огромные выгоды, которые такие каркасы дают при подделывании и тестировании унаследованного кода.

6.2. Полезные качества хороших изолирующих каркасов

В .NET (и отчасти в Java) в последние два года стало подрастать новое поколение изолирующих каркасов. Они избавились от лишнего веса, присущего старым заслуженным каркасам, и сделали гигантский рывок в плане удобочитаемости, удобства работы и простоты. Самое главное, что они теперь поддерживают стабильность тестов благодаря особенностям, о которых я расскажу чуть ниже.

К числу этих новых каркасов относятся Typemock Isolator (хотя он-то как раз не первой молодости), NSubstitute и FakeItEasy. Первый из них неограниченный, два других ограниченные, но, несмотря на ограничения, обладают рядом интересных возможностей.

К сожалению, в таких языках, как Ruby, Python, JavaScript и другие, изолирующие каркасы все еще отстают по части удобочитаемости и удобства пользования. Возможно, дело в недостаточной зрелости самих каркасов, а, возможно, в том, что культура автономного тестирования в этих языках еще не достигла таких высот, как на платформе .NET. А может быть и так, что мы все неправы, а идти надо тем путем, который применяется для изоляции в Ruby. Да, так о чем это я?

Хороший изолирующий каркас должен обладать двумя важными полезными качествами:

- неустареваемость;
- удобство пользования.

Вот перечень некоторых особенностей новых каркасов, благодаря которым они этими качествами обладают:

- рекурсивные подделки;
- игнорирование аргументов по умолчанию;
- массовое подделывание;
- нестрогое поведение подделок;
- нестрогие подставки.

6.3. Особенности, обеспечивающие неустареваемость и удобство пользования

Неустаревающий тест перестает проходить только при наличии основательных причин, связанных с существенным изменением продуктового кода. Удобство пользования – это качество, благодаря которому каркас легко понять и применять. Изолирующие каркасы – такая штука, которую проще простого использовать *плохо*, что приводит к очень хрупким и не защищенным от будущих изменений тестам.

Перечислю особенности, способствующие повышению стабильности тестов:

- рекурсивные подделки;
- игнорирование по умолчанию аргументов для поведений и проверок;

- нестрогие проверки и поведение;
- массовое подделывание.

6.3.1. Рекурсивные подделки

Рекурсивное подделывание – это специальное поведение поддельных объектов в случае, когда их методы возвращают другие объекты. Эти объекты автоматически становятся подделками. Любые объекты, возвращенные любым методом такого автоматически подделанного объекта, также будут поддельными. То есть налицо рекурсия. Например:

```
public interface IPerson
{
  IPerson GetManager();
}

[Test]
public void RecursiveFakes_work()
{
  IPerson p = Substitute.For<IPerson>();

  Assert.IsNotNull(p.GetManager());
  Assert.IsNotNull(p.GetManager().GetManager());
  Assert.IsNotNull(p.GetManager().GetManager().GetManager());
}
```

Обратите внимание, что для получения такого поведения мы ничего специально не делали. Почему эта возможность так важна? Чем меньше на стадии подготовки теста нужно сообщать о том, какие конкретно API подделывать, тем слабее тест связан с фактической реализацией продуктового кода и тем менее вероятно, что придется изменять тест, если в будущем изменится продуктовый код.

Не все изолирующие каркасы поддерживают рекурсивное подделывание, потому обращайте на это внимание при выборе. Насколько я знаю, пока такая возможность реализована только на платформе .NET, и очень хотелось бы увидеть ее в каркасах для других языков.

Отметим также, что ограниченные каркасы в .NET способны поддержать рекурсивные подделки только для методов, которые можно переопределить в сгенерированном коде, т. е. открытых виртуальных методов или методов, объявленных в интерфейсе.

Некоторые опасаются, что такая возможность позволяет легко нарушить закон Деметры (http://en.wikipedia.org/wiki/Law_of_Demeter). Я с этим не согласен, потому что хороший проект не навязывается инструментами, а создается людьми, которые общаются между собой,

учатся друг у друга и парами проводят критический анализ кода. Но о проектировании у нас будет время поговорить в главе 11.

6.3.2. Игнорирование аргументов по умолчанию

В настоящее время во всех каркасах, кроме Typemock Isolator, значения аргументов, переданные API изменения поведения или проверки, используются как ожидаемые значения по умолчанию. Isolator по умолчанию игнорирует переданные вами значения, если только явно не сказать (с помощью вызова API), что значения аргументов важны. Отпадает необходимость включать `Arg.IsAny<Type>` во все методы, что сокращает размер теста и позволяет обойтись без обобщенных методов, которые только затрудняют чтение. В Typemock Isolator, чтобы возбудить исключение при любых аргументах, нужно просто написать:

```
Isolate.WhenCalled(() => stubLogger.Write(""))
     .WillThrow(new Exception("Fake"));
```

6.3.3. Массовое подделывание

Под массовым подделыванием (wide faking) понимается возможность подделывать сразу несколько методов. В каком-то смысле рекурсивное подделывание – частный случай этой идеи, но существуют и другие реализации.

Например, FakeItEasy позволяет сказать, что все методы некоторого объекта должны возвращать одно и то же значение. Или что это распространяется только на методы, возвращающие значения определенного типа:

```
A.CallTo(foo).Throws(new Exception());
A.CallTo(foo).WithReturnType<string>().Returns("hello world");
```

В Typemock можно сказать, что по умолчанию все статические методы некоторого типа должны возвращать поддельное значение:

```
Isolate.Fake.StaticMethods(typeof(HttpRuntime));
```

Начиная с этого момента, любой статический метод объекта типа `HttpRuntime` будет возвращать поддельное значение, зависящее от типа, или – если он возвращает объект – рекурсивную подделку.

Я считаю, что это очень важно для обеспечения неустареваемости тестов в случае развития продуктового кода. Метод, добавленный в

продуктовый код через шесть месяцев после написания теста, будет автоматически подделываться всеми существующими тестами, которым, следовательно, появление нового метода безразлично.

6.3.4. Нестрогое поведение подделок

Мир изолирующих каркасов был и по большей части остается весьма строгим. Многие каркасы для платформ, отличных от .NET (например, для Java и Ruby), по умолчанию строгие, тогда как каркасы для .NET уже перешагнули этот этап.

Методы строгой подделки будут успешно вызваны, только если указать с помощью API, что они «ожидаемые». Эта возможность ожидать вызова метода поддельного объекта отсутствует в NSubstitute (и в FakeItEasy), но существует во многих других каркасах для .NET и других языков (Moq, Rhino Mocks и старый API каркаса Typemock Isolator).

Если метод помечен как ожидаемый, то любое обращение, отличное от ожидаемого (например, я ожидаю, что в начале теста метод `LogError` будет вызван с параметром `a`), – либо из-за различий в значениях параметров, либо из-за имени метода – обычно заканчивается исключением.

Тест обычно отказывает при первом неожиданном вызове метода строгой подставки. Я говорю *обычно*, потому что подставка может возбуждать или не возбуждать исключение в зависимости от реализации изолирующего каркаса. Некоторые каркасы позволяют отложить возбуждение всех исключений до вызова `verify()` в конце теста.

Почему многие каркасы устроены именно так, можно прочитать в книге Freeman, Pryce «Growing Object-Oriented Software, Guided by Tests» (Addison-Wesley Professional, 2009). Авторы используют утверждения относительно подставок для описания «протокола» взаимодействия объектов. Поскольку протокол обязан быть строгим, то чтение теста должно помогать в понимании того, какие способы взаимодействия с объектом допустимы.

И в чем тут проблема? Дело не в самой идее, а в том, с какой легкостью эту возможность можно употребить во вред.

Строгая подставка может отказывать в двух случаях: если вызывается неожиданный метод или если не вызываются ожидаемые (что определяется при обращении к `Received()`).

Меня волнует первый случай. В предположении, что мне безразличны внутренние протоколы общения объектов внутри моей еди-

ницы работы, я не должен высказывать никаких утверждений об их взаимодействиях, если не хочу попасть в весьма затруднительное положение. Тест может отказать, если я решу вызвать метод какого-то внутреннего для единицы работы объекта. И хотя он никак связан с ее конечным результатом, тест все равно хнычет: «Ты мне не говорил, что кто-то собирается вызывать *mom* метод».

6.3.5. Нестрогие подставки

Как правило, при использовании нестрогих подставок тесты получаются менее хрупкими. Нестрогий подставной объект позволяет вызывать любые свои методы, даже неожиданные. Если метод возвращает значение, то подставка вернет значение по умолчанию либо `null`, если требуется вернуть объект. В более развитых каркасах есть даже понятие рекурсивных подделок – если у поддельного объекта имеется метод, возвращающий объект, то по умолчанию будет возвращен поддельный объект соответствующего типа. И этот объект также – рекурсивно – будет возвращать поддельные объекты из методов, возвращающих объекты. (Такая функция имеется в Typemock Isolator, NSub, Moq и частично в Rhino Mocks.)

В листинге 5.3 из главы 5 приведен незамутненный пример нестрогой подставки. Нам безразлично, какие еще методы вызывались. В листинге 5.4 и следующем за ним фрагменте показано, как можно сделать тест более стабильным и неустаревающим, воспользовавшись сопоставителем аргументов вместо полной строки. Сопоставление аргументов позволяет создавать правила, описывающие, как следует передавать поддельному объекту параметры, чтобы тот считал их правильными. Но обратите внимание, как это уродует тестовый код.

6.4. Антипаттерны проектирования в изолирующих каркасах

Приведу перечень некоторых имеющихся в современных каркасах антипаттернов, которые можно без труда сгладить:

- смешение понятий;
- запись и воспроизведение;
- липкое поведение;
- сложный синтаксис.

Рассмотрим их поочередно.

6.4.1. Смешение понятий

Смешение понятий я называю *передозировкой подставок*. Я предпочитаю каркасы, в которых слово «подставка» (mock) вообще не употребляется.

Мы должны знать, сколько в тесте используется заглушек и сколько подставок, потому что наличие более одной подставки обычно свидетельствует о проблеме. Если каркас не различает то и другое, то он может назвать нечто подставкой, хотя на самом деле оно используется как заглушка. У нас уйдет время на то, чтобы понять, есть тут проблема или нет, а, значит, удобочитаемость пострадает.

Вот пример из Moq:

```
[Test]
public void ctor_WhenViewhasError_CallsLogger()
{
  var view = new Mock<IView>();
  var logger = new Mock<ILogger>();

  Presenter p = new Presenter(view.Object, logger.Object);
  view.Raise(v => v.ErrorOccured += null, "fake error");

  logger.Verify(log =>
      log.LogError(It.Is<string>(s=> s.Contains("fake error"))));
}
```

Вот как можно избежать смешения понятий.

- Использовать в API специальные слова для *подставки* и *заглушки*. Так сделано, например, в Rhino Mocks.
- Вообще не употреблять термины *подставка* и *заглушка* в API, а использовать общее обозначение для любого поддельного объекта. Например, в FakeItEasy нет ничего, кроме *Fake*<Something>. Подставки и заглушки отсутствуют как класс. В NSubstitute, как вы помните, имеется лишь *Substitute*<Something>. В Typemock Isolator вызывается Isolate.Fake.Instance<Something>. Подставка и заглушка нигде не упоминаются.
- Если вы пользуетесь изолирующим каркасом, который не различает заглушки и подставки, то хотя бы называйте переменные по принципу mockXXX и stubXXX, чтобы как-то сгладить проблемы с удобочитаемостью.

Если вообще отказаться от перегруженного термина или дать пользователю возможность указать, что именно он создает, то тесты ста-

нут понятнее. Или, по крайней мере, будет меньше путаницы в терминологии.

Вот тот же тест, что и выше, только переменные названы в соответствии со способом использования. Так ведь читается лучше, правда?

```
[Test]
public void ctor_WhenViewhasError_CallsLogger()
{
  var stubView = new Mock<IView>();
  var mockLogger = new Mock<ILogger>();

  Presenter p = new Presenter(stubView.Object, mockLogger.Object);
  stubView.Raise(view => view.ErrorOccured += null, "fake error");

  mockLogger.Verify(logger =>
       logger.LogError(It.Is<string>(s=>s.Contains("fake error"))));
}
```

6.4.2. Запись и воспроизведение

Изолирующие каркасы, основанные на записи и воспроизведении, порождали неудобочитаемые тесты. Верный признак неудобочитаемости – необходимость несколько раз пробегать текст глазами, чтобы понять, что он означает. Обычно это наблюдается в коде, написанном с применением каркаса, поддерживающего API записи и воспроизведения.

Возьмем пример использования Rhino Mocks (который поддерживает запись и воспроизведения) из блога Расмуса Кроманн-Ларсена по адресу http://rasmuskl.dk/post/Why-AAA-style-mockingis-better-than-Record-Playback.aspx (не пытайтесь его компилировать, это всего лишь пример).

```
[Test]
public void ShouldIgnoreRespondentsThatDoesNotExistRecordPlayback()
{
  // Подготовка
  var guid = Guid.NewGuid();
  // Часть действия
  IEventRaiser executeRaiser;

  using(_mocks.Record())
  {
     // Подготовка (или уже утверждение?)
     Expect.Call(_view.Respondents).Return(new[] {guid.ToString()});
     Expect.Call(_repository.GetById(guid)).Return(null);

     // Часть действия
```

```
        _view.ExecuteOperation += null;
        executeRaiser = LastCall.IgnoreArguments()
             .Repeat.Any()
             .GetEventRaiser();

        // Утверждение
        Expect.Call(_view.OperationErrors = null)
             .IgnoreArguments()
             .Constraints(List.IsIn("Non-existant respondent: " + guid));
    }

    using(_mocks.Playback())
    {
        // Подготовка
        new BulkRespondentPresenter(_view, _repository);

        // Действие
        executeRaiser.Raise(null, EventArgs.Empty);
    }
}
```

А вот тот же самый код в каркасе Moq (который поддерживает стиль подготовка-действие-утверждение).

```
[Test]
public void ShouldIgnoreRespondentsThatDoesNotExist()
{
    // Подготовка
    var guid = Guid.NewGuid();
    _viewMock.Setup(x => x.Respondents).Returns(new[] { guid.ToString() });
    _repositoryMock.Setup(x => x.GetById(guid)).Returns(() => null);

    // Действие
    _viewMock.Raise(x => x.ExecuteOperation += null, EventArgs.Empty);

    // Утверждение
    _viewMock.VerifySet(x => x.OperationErrors =
    It.Is<IList<string>>(l=>l.Contains("Non-existant respondent: "+guid)));
}
```

Не правда ли, разница гигантская и не в пользу записи и воспроизведения?

6.4.3. Липкое поведение

Если вы один раз сказали, что поддельный метод должен при вызове вести себя определенным образом, то что произойдет, когда он будет вызван в продуктовом коде в следующий раз? Или следующие 100

раз? Должно ли тесту быть до этого дело? Если задумано так, что поддельное поведение методов проявляется только один раз, то тест должен давать ответ на вопрос «что делать дальше?» всякий раз, как в продуктовый код вносятся изменения, приводящие к вызову поддельного метода, даже если тесту нет дела до этих дополнительных вызовов. Теперь тест более тесно связан с внутренними деталями реализации.

Для решения этой проблемы изолирующий каркас может сделать поведение «липким». Однажды сказав, что метод должен вести себя определенным образом (например, возвращать `false`), мы можем рассчитывать, что он будет себя так, до тех пор пока не получит иное указание (все последующие вызовы будут возвращать `false`, хоть 100 раз метод вызывай). Это освобождает тест от необходимости знать, как метод должен вести себя в дальнейшем, когда это уже несущественно для теста.

6.4.4. Сложный синтаксис

В некоторых каркасах трудно запомнить, как выполняются стандартные операции, даже если вы это уже несколько раз делали. Это мешает кодированию. API должен легко запоминаться. Например, в FakeItEasy все допустимые операции *обязаны* начинаться с прописной буквы `A`. Вот пример, взятый с вики-сайта FakeItEasy по адресу https://github.com/FakeItEasy/FakeItEasy/wiki:

```
var lollipop = A.Fake<ICandy>();
var shop = A.Fake<ICandyShop>();

// Настроить метод, так чтобы он возвращал
// определенное значение, тоже просто:
A.CallTo(() => shop.GetTopSellingCandy()).Returns(lollipop);

A.CallTo(() => foo.Bar(A<string>.Ignored,
    "second argument")).Throws(new Exception());

// Используем подделку так, будто это настоящий экземпляр типа.
var developer = new SweetTooth();
developer.BuyTastiestCandy(shop);

// В утверждениях применяется точно такой же синтаксис, как при
// настройке вызовов, ничего нового запоминать не надо.
A.CallTo(() => shop.BuyCandy(lollipop)).MustHaveHappened();
```

Создание подделки начинается с A

Определение поведения метода начинается с A

Использование сопоставителя аргумента начинается с A

Проверка факта вызова метода начинается с A

Та же идея применяется в Typemock Isolator, где все вызовы API начинаются словом `Isolate`.

Наличие единственной точки входа позволяет правильно начать слово, а затем с помощью встроенного в IDE механизма Intellisense решить, что писать дальше.

В NSubstitute нужно запомнить, что для создания подделок служит метод `Substitute`, что нужно использовать методы расширения реальных объектов для проверки или изменения поведения и что `Arg<T>` используется в сопоставителях аргументов.

6.5. Резюме

Изолирующие каркасы делятся на две категории: ограниченные и неограниченные. В зависимости от платформы возможностей может быть больше или меньше, поэтому, выбирая каркас, важно понимать, что он умеет, а чего не умеет делать.

В .NET неограниченные каркасы построены на основе API профилирования, тогда как большинство ограниченных генерируют и компилируют код во время выполнения – точно так же, как поступаем мы сами, когда пишем заглушки и подставки вручную.

Изолирующие каркасы, которые поддерживают неустареваемость и удобство пользования, могут облегчить вашу жизнь в стране автономных тестов, а те, что не поддерживают, – осложнят ее.

Вот и все! Мы рассмотрели основные приемы написания автономных тестов. В третьей части книги мы займемся управлением кодом тестов, организацией тестов и паттернами, которые приводят к созданию надежных, легко сопровождаемых и понятных тестов.

Часть III.

ТЕСТОВЫЙ КОД

В этой части рассматриваются способы организации автономных тестов и управления ими, дающие гарантии высокого качества тестов в реальных проектах.

В главе 7 сначала раскрывается роль автономного тестирования в процессе автоматизированной сборки, а затем описывается несколько подходов к организации различных видов тестов по категориям (быстродействию, типу). При этом цель состоит в том, чтобы достичь безопасной зеленой зоны. Объясняется также, как расширять API или инфраструктуру тестов для приложения.

В главе 4 мы рассмотрим три фундаментальных качества хороших автономных тестов – удобочитаемость, удобство сопровождения и надежность – а также способы их достижения. Если у вас есть время для чтения только одной главы этой книги, то это должна быть глава 8.

ГЛАВА 7. Иерархии и организация тестов

В этой главе:

- Прогон автономных тестов во время ночной автоматизированной сборки.
- Использование непрерывной интеграции для автоматизированной сборки.
- Организация тестов в решении.
- Паттерны наследования тестовых классов.

Автономные тесты важны для приложения не менее, чем продуктовый код. И точно так же следует тщательно продумывать, как разместить тесты – физически и логически – относительно тестируемого кода. Если поместить автономные тесты в неподходящее место, то все старания могут оказаться тщетны – тесты не запустятся.

Аналогично, если вы не придумаете, как повторно использовать части тестов, не создадите служебных методов для тестирования, не будете организовывать тесты иерархически, то код тестов будет невозможно ни понять, ни сопровождать.

В этой главе мы рассмотрим паттерны и рекомендации, которые помогут вам определить, как тесты должны выглядеть, выполняться и при этом органично интегрироваться с продуктовым кодом и другими тестами.

Куда помещать тесты, зависит от того, где они будут использоваться и как запускаться. Существует два сценария: прогон тестов в составе автоматизированной процедуры сборки и локально разработчиком на его машине. Автоматизированная сборка – дело очень важное, поэтому с нее и начнем.

7.1. Прогон автоматизированных тестов в ходе автоматизированной сборки

Эффективность автоматизированной процедуры сборки никак не следует игнорировать. Я занимаюсь автоматизацией сборки и доставки больше десяти лет, и это один из важнейших факторов, благодаря которым моя команда продуктивнее работает и быстрее получает обратную связь. Всякий, кто хочет сделать свою команду более гибкой и оснащенной для своевременного реагирования на изменения требований, должен позаботиться о следующих вещах.

- Вносить изменения в код понемногу.
- Прогонять все тесты, чтобы иметь гарантию, что написанное ранее по-прежнему работает.
- Следить за тем, чтобы при интеграции кода не возникало ошибок и не повреждались другие проекты, от которых код зависит.
- Организовать процедуру создания передаваемого пакета и автоматического развертывания одним нажатием кнопки.

По всей вероятности, чтобы решить указанные задачи, вам потребуется несколько конфигураций и скриптов сборки. Скрипты сборки должны располагаться в системе управления версиями вместе с исходным продуктовым кодом. Они вызываются сервером непрерывной интеграции в соответствии с его конфигурацией.

Некоторые скрипты сборки запускают тесты, особенно те, что активируются сразу после записи нового кода в систему управления версиями (СУВ). Прогон тестов позволяет узнать, работает ли новый или уже существующий код, написанный вами или еще кем-то. Вы интегрируете свой код с другими проектами. Тесты должны показать, не сломалось ли что в вашем коде или в том, что логически зависит от него. Делая это автоматически при каждом сохранении в СУВ, вы запускаете процесс непрерывной интеграции. Что это означает, я расскажу в разделе 7.1.2.

Если бы вы интегрировали код лично, то, наверное, делали бы следующее:

- извлекли бы последние версии кода, написанного всеми участниками проекта, из репозитория системы управления версиями;

- попытались бы откомпилировать весь код на своей машине;
- локально прогнали бы все тесты;
- исправили бы все обнаруженные ошибки;
- записали бы новые версии кода в СУВ.

Для автоматизации этой работы существуют инструменты, а именно скрипты автоматизированной сборки и серверы непрерывной интеграции.

Процедура автоматизированной сборки собирает все эти шаги под одной логической крышей, на которой можно было бы повесить вывеску «как мы здесь выпускаем код». Процедура сборки – это совокупность скриптов сборки, автоматизированных триггеров, сервера, возможно, агентов сборки (которые и выполняют фактическую работу) и принимаемое всеми членами команды соглашение работать таким способом.

Смысл соглашения в том, что все понимают последовательность шагов, обязательных для того, чтобы все это работало настолько непрерывно и автоматически, насколько возможно и необходимо (иногда не требуется автоматически развертывать систему на производственной машине без участия человека), и готовы эти шаги выполнять.

Если в процессе сборки обнаруживается ошибка, сервер может уведомить заинтересованные стороны о *прерывании сборки*.

Уточним: процесс сборки – это логическое понятие, охватывающее скрипты сборки, серверы непрерывной интеграции, триггеры сборки и понимаемое и поддерживаемое всеми членами команды соглашение о развертывании и интеграции.

7.1.1. Анатомия скрипта сборки

У меня обычно образуется несколько скриптов сборки, каждый из которых служит одной какой-то цели. При такой организации процесс сборки оказывается более удобным для сопровождения и целостным. Скрипты бывают следующих видов:

- скрипт сборки непрерывной интеграции (НИ-сборки);
- скрипт ночной сборки;
- скрипт развертываемой сборки.

Я предпочитаю их разделять, потому что рассматриваю скрипты сборки как небольшие функции, которые можно вызывать с параметрами, подавая на вход текущую версию исходного кода. А вызывающей стороной является сервер НИ.

Скрипт НИ-сборки как минимум компилирует текущий исходный код в отладочном режиме и прогоняет все автономные тесты. В принципе, он может прогонять и другие тесты, если они работают достаточно быстро. Предполагается, что скрипт НИ-сборки дает максимум информации в кратчайшие сроки. Чем он быстрее, тем раньше вы узнаете, что, скорее всего, ничего не сломали и можете работать дальше.

Скрипт ночной сборки обычно работает дольше. Я предпочитаю запускать его сразу после НИ-сборки, чтобы получить дополнительную информацию, но не жду, пока он завершится, а продолжаю себе кодировать. Он занимает больше времени, потому что должен выполнить операции, которые для НИ-сборки сочтены несущественными или не настолько важными, чтобы включать их в цикл НИ с быстрой обратной связью. К таким операциям можно отнести едва ли не все что угодно, но обычно сюда входит компиляция в выпускном режиме, прогон всех медленных тестов и, возможно, развертывание в тестовой среде в преддверье следующего дня.

Я называю эти сборки ночными, но вообще-то их можно запускать много раз в день. Один раз ночью – это необходимый минимум. Такие сборки дают больше информации, но для ее сбора требуется и больше времени.

Скрипт развертываемой сборки – это, по существу, механизм доставки. Он запускается сервером НИ и может содержать как простейшую команду `xcopy` копирования на удаленный сервер, так и сложную процедуру развертывания на сотнях серверов, повторной инициализации экземпляров Azure или Amazon Elastic Compute Cloud (EC2) и объединения баз данных.

Все сборки обычно извещают пользователя по электронной почте о факте прерывания из-за ошибки, но самым главным и обязательным получателем такого уведомления является сторона, запустившая скрипты сборки: сервер непрерывной интеграции.

Существует много инструментов, помогающих в создании автоматизированных систем сборки. У одних исходный код открыт, другие просто бесплатны, третьи поставляются на коммерческих условиях. Ниже перечислены некоторые такие инструменты.

Создание скриптов сборки:

- NAnt (nant.sourceforge.net)
- MSBuild (www.infoq.com/articles/MSBuild-1)
- FinalBuilder (www.FinalBuilder.com)
- Visual Build Pro (www.kinook.com)
- Rake (http://rake.rubyforge.org/)

Серверы непрерывной интеграции:

- CruiseControl.NET (cruisecontrol.sourceforge.net)
- Jenkins (http://jenkins-ci.org/)
- Travis CI (http://about.travis-ci.org/docs/user/getting-started/)
- TeamCity (JetBrains.com)
- Hudson (http://hudson-ci.org/)
- Visual Studio Team Foundation Service (http://tfs.visualstudio.com/)
- ThoughtWorks Go (www.thoughtworks-studios.com/go-agile-release-management)
- CircleCI (https://circleci.com/), если вы работаете только через github.com
- Bamboo (www.atlassian.com/software/bamboo/overview)

В некоторые серверы НИ встроены средства создания задач, относящихся к скриптам сборки. Я стараюсь не пользоваться такими средствами, потому что хочу, чтобы действия скриптов сборки зависели от версии (или чтобы скрипты хранились в системе управления версиями), тогда я в любой момент смогу вернуться к любой версии исходных кодов и выполнить сборку, привязанную именно к этой версии.

Из перечисленных инструментов мне больше всего нравятся FinalBuilder и TeamCity. Если бы FinalBuilder не было (а он существует только на платформе Windows), то я бы взял Rake, потому что терпеть не могу использовать XML для управления сборкой, т. к. он здорово затрудняет сопровождение скриптов. Rake – зона, свободная от XML, тогда как MSBuild и NAnt пичкают вас XML'ем до такой степени, что теги будут сниться еще несколько месяцев. Каждый из перечисленных инструментов особенно блистает в каком-то одном деле, хотя авторы TeamCity добавляют все больше и больше встроенных задач, и мне кажется, что получающиеся в результате скрипты сборки становятся менее удобными для сопровождения.

7.1.2. Запуск сборки и интеграции

Мы уже упомянули о непрерывной интеграции, а теперь рассмотрим ее более формально. Понимать этот термин следует буквально – процесс автоматической сборки и интеграции происходит непрерывно. Например, определенный скрипт сборки может запускаться всякий раз, как кто-то добавляет в систему новый исходный код, или раз в 45 минут или сразу по завершении предыдущего скрипта сборки.

Основные задачи сервера НИ таковы:

- запускать скрипт сборки по определенным событиям;
- предоставлять скрипту сборки контекст и данные, например: номер версии, исходный код, артефакты других сборок, параметры сборки и т. д.;
- предоставлять историю сборки и метрики;
- информировать о текущем состоянии активных и неактивных сборок.

Сначала рассмотрим события-триггеры, по которым автоматически запускается скрипт сборки. Это может быть изменение в системе управления версиями, наступление некоторого момента времени или успешное или неудачное завершение другого скрипта сборки. Сервер НИ позволяет настроить несколько триггеров для запуска единицы работы. Единицы работы в этом контексте часто называют конфигурациями сборки.

В конфигурации сборки задаются исполняемые команды, например: выполнить определенную программу, откомпилировать и т. д. Я рекомендую ограничиться запуском исполняемого файла, который выполняет скрипт сборки, хранящийся в СУВ, чтобы обеспечить максимальную совместимость с текущей версией исходного кода. Например, TeamCity позволяет включать шаги сборки в конфигурацию. Шаг сборки может принимать различные формы. Например, это может быть выполнение команды DOS или компиляция SLN-файла для .NET. Я не пользуюсь ничем, кроме простого шага запуска пакетного файла или скрипта сборки, который извлечен из СУВ агентом сборки вместе с исходным кодом.

У конфигурации сборки может быть контекст. Контекст может содержать много всего, но как правило в него входит текущий снимок исходного кода, извлеченного из системы управления версиями. В контекст могут также включаться переменные окружения, необходимые скрипту сборки, или параметры, непосредственно задаваемые в командной строке. В состав контекста могут входить и копии артефактов, оставшихся от предыдущего запуска той же или какой-то другой конфигурации сборки. Артефактом называется конечный результат выполнения скрипта сборки. Это могут быть двоичные файлы, конфигурационные файлы и вообще файлы любого типа.

У конфигурации сборки может быть история. Мы можем узнать, когда она запускалась, сколько времени работала и когда в последний раз завершилась успешно. Можно также узнать, сколько было выполнено тестов и какие тесты не прошли. Что именно включается в историю, зависит от сервера НИ.

У сервера НИ обычно имеется инструментальная панель, на которой показывается текущее состояние сборок. Некоторые серверы даже позволяют создавать нестандартный HTML и JavaScript-код, который можно разместить на страницах компании в интранете для настройки отображения состояния. В состав некоторых серверов НИ входят средства интеграции, которые работают на персональном компьютере, ведут непрерывный мониторинг состояния сборки и уведомляют пользователя об аварийном прерывании сборки.

Дополнительные сведения об автоматизации сборки

Есть немало полезных рекомендаций, с которыми неплохо бы познакомиться, но они выходят за рамки этой книги. Читателю, желающему больше узнать о непрерывной интеграции, я рекомендую книги Jez Humble, David Farley «Continuous Delivery» (Addison-Wesley Professional, 2010) и Paul Duvall, Steve Matyas, Andrew Glover «Continuous Integration»[1] (Addison-Wesley Professional, 2007). Возможно, вас заинтересует и написанная мной книга на эту тему «Beautiful Builds». Это моя попытка создать язык паттернов для описания типичных проблем и решений в области процесса сборки. Ее текст опубликован на сайте www.BeautifulBuilds.com.

7.2. Распределение тестов по скорости и типу

Совсем нетрудно прогнать тесты, измерить время выполнения каждого и решить, какие отнести к автономным, а какие – к интеграционным. Сделав это, поместите тесты в разные места. Необязательно заводить для них отдельные проекты, достаточно разных папок и пространств имен.

На рис. 7.1 показана простая структура папок внутри одного проекта Visual Studio. В некоторых компаниях считают, что проще заводить разные проекты для автономных и интеграционных тестов; это зависит от используемой системы сборки и каркаса автономного тестирования. При такой организации проще работать с командными программами, которые принимают на входе сборку, содержащую только тесты одного вида, и прогоняют их. На рис. 7.2 показано, как можно организовать два проекта с тестами в одном решении.

---

[1] Поль М. Дюваль, Стивен Матиас, Эндрю Гловер «Непрерывная интеграция. Улучшение качества программного обеспечения и снижение риска». Вильямс, 2008. – *Прим. перев.*

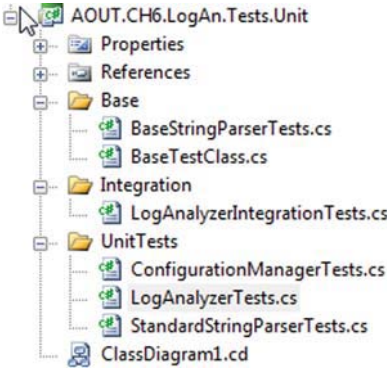


Рис. 7.1. Интеграционные и автономные тесты могут находиться в разных папках и пространствах имен, оставаясь при этом в одном проекте. Для базовых классов заведена отдельная папка

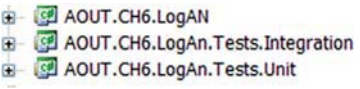


Рис. 7.2. Для автономных и интеграционных тестов, относящихся к проекту LogAn, заведены отдельные проекты с разными пространствами имен

Даже если вы еще не настроили автоматизированную систему сборки, отделить автономные тесты от интеграционных все равно полезно. Смешение тестов обоих видов может повлечь за собой серьезные последствия, например, другие члены команды не станут прогонять ваши тесты. Об этой проблеме мы и поговорим в следующем разделе.

7.2.1. Разделение автономных и интеграционных тестов и человеческий фактор

Я рекомендую разделять автономные и интеграционные тесты. Если этого не сделать, велик риск, что разработчики не будут прогонять тесты достаточно часто. Казалось бы – если тесты существуют, то почему бы не прогонять их так часто, как необходимо? Одна из причин – лень или работа в условиях катастрофической нехватки времени.

Допустим, что разработчик, выгрузивший последнюю версию исходного кода, обнаружил, что какой-то автономный тест не проходит. Возможно несколько причин:

- в тестируемом коде ошибка;
- тест написан неправильно;

- тест уже не актуален;
- для прогона теста необходимо что-то настроить.

Во всех случаях, кроме последнего, у разработчика имеются достаточные основания, чтобы прерваться и изучить код. Последний же случай к разработке отношения не имеет; это проблема конфигурации, которая часто считается менее важной, потому что мешает прогону тестов. Если такой тест не проходит, разработчик просто игнорирует ошибку и переходит к другим делам (у всех ведь есть «более важные» занятия).

Наличие в составе автономных тестов таких скрытых интеграционных тестов с неизвестными или неожиданными требованиями к настройке (например, подключение к базе данных) плохо с разных точек зрения. Такие тесты сложнее прогонять, они отвлекают время и деньги на поиск несуществующих проблем и вообще уменьшают доверие разработчика к комплекту тестов. Они, как гнилое яблоко в корзине, портят все остальные. В следующий раз, когда случится что-то подобное, разработчик, возможно, даже не захочет искать причину ошибки, а просто скажет: «А, этот тест иногда падает, это нормально».

Чтобы такого не было, можно создать безопасную зеленую зону.

7.2.2. Безопасная зеленая зона

Размещайте свои интеграционные и автономные тесты в разных местах. Тем самым вы предоставляете другим членам команды безопасную зеленую зону, содержащую только автономные тесты. И все знают, что могут взять последнюю версию кода, прогнать все тесты из соответствующей папки или пространства имен, и все тесты при этом должны быть зелеными. Если какой-то тест из безопасной зеленой зоны не проходит, то имеет место реальная проблема, а не ошибка в настройках.

Это не означает, что интеграционные тесты вообще не нужно прогонять. Но поскольку они по сути своей работают дольше, то, скорее всего, разработчики будут прогонять автономные тесты несколько раз в день, а интеграционные реже, но уж во время ночной сборки – обязательно. Разработчик может сосредоточиться на своей работе, и, если автономные тесты проходят, то у него будет ощущение уверенности – пусть даже неполной. Во время ночной сборки должны быть выполнены все автоматизированные задачи, которые настраивают среду, так чтобы интеграционные тесты проходили.

Кроме того, создание отдельной интеграционной зоны (в отличие от безопасной зеленой) – это не только место, где медленные интеграционные тесты проходят карантин, сюда же можно поместить документацию по настройке, которую нужно выполнить, чтобы эти тесты работали.

Автоматизированная система сборки настроит все сама. Но если вы хотите прогонять тесты локально, то должны завести в своем решении или проекте интеграционную зону, в которой хранится вся информация о том, что нужно сделать для успешного прогона. Однако если требуется выполнить только быстрые тесты (в безопасной зеленой зоне), то в интеграционную можно не заглядывать.

Впрочем, все это неактуально, если вы не храните свои тесты в системе управления версиями. Об этом мы и поговорим ниже.

7.3. Тесты должны храниться в системе управления версиями

Тесты должны храниться в системе управления версиями. Написанный вами тестовый код должен находиться в ее репозитории наряду с продуктовым. Вообще к тестам следует относиться столько же внимательно, как к продуктовому коду. Они должны присутствовать в каждой ветви дерева версий и извлекаться автоматически вместе с последней версией.

Поскольку автономные тесты так тесно связаны с кодом и API, они всегда должны сохранять привязку к версии кода, который тестируют. Получить версию 1.0.1 продукта значит получить и версию 1.0.1 тестов для него. В версии 1.0.2 код продукта и тестов будет другим.

Кроме того, только благодаря хранению тестов в СУВ процедуры автоматизированной сборки вообще могут надежно применять текущую версию тестов к вашему продукту.

Но если мы установили, что тесты – часть СУВ, то где именно они должны находиться?

7.4. Соответствие между тестовыми классами и тестируемым кодом

Структура и размещение тестовых классов должны быть устроены так, чтобы можно было легко решить следующие задачи:

- найти все тесты, относящиеся к конкретному проекту;
- найти все тесты, относящиеся к конкретному классу;
- найти все тесты, относящиеся к конкретному методу.

Существует несколько паттернов, помогающих обустроить тесты таким образом.

7.4.1. Соответствие между тестами и проектами

Я предпочитаю создавать отдельный проект для тестов и называть его так же, как тестируемый проект, с добавлением суффикса .UnitTests. Например, для проекта Osherove.MyLibrary я бы завел еще проекты Osherove.MyLibrary.UnitTests, Osherove.MyLibrary.IntegrationTests и, быть может, еще что-то в том же духе (пример см. на рис. 7.2). Пусть грубовато, зато интуитивно понятно и позволяет легко найти все тесты, относящиеся к конкретному проекту.

Возможно, вы захотите воспользоваться имеющейся в Visual Studio возможностью создавать папки внутри решения и поместить всю эту троицу в отдельную папку, но это уже дело вкуса.

7.4.2. Соответствие между тестами и классами

Существует несколько подходов к установлению соответствия между тестами и тестируемыми классами. Рассмотрим два наиболее распространенных: один тестовый класс на каждый тестируемый и несколько тестовых классов для сложных тестируемых методов.

Совет. Этими двумя паттернами я пользуюсь чаще всего, но существуют и другие. Подробнее о них можно прочитать в книге Джерарда Мезароша «Шаблоны тестирования xUnit. Рефакторинг кода тестов».

Один тестовый класс на каждый тестируемый класс или единицу работу

Чтобы быстро найти все тесты для конкретного класса, можно применить решение, очень похожее на описанное выше для проектов: создать в тестовом проекте класс с таким же именем, как у тестируемого класса, и суффиксом `UnitTests`. Для класса `LogAnalyzer` тестовый класс в тестовом проекте назывался бы `LogAnalyzer.UnitTests`.

Обратите внимание на множественное число: в этом классе будет несколько тестов для тестируемого класса. Точность важна. Когда речь идет о тестировании, удобочитаемость и ясность языка много значат, а если вы начнете срезать углы в одном месте, то захотите поступить так же и в других и в конце концов проблем не оберетесь.

Паттерн «один тестовый класс на один тестируемый» (он также упоминается в книге Мезароша) – самый простой и употребительный способ организации тестов. Все тесты для всех методов тестируемого класса находятся в одном большом тестовом классе. Но при этом для некоторых методов тестов может оказаться так много, что тестовый класс станет трудно читать и найти в нем что-нибудь будет проблематично. Иногда для одного метода тестов больше, чем для всех остальных вместе взятых. Кстати, это само по себе можно расценивать как признак того, что метод делает слишком много.

Совет. Удобочитаемостью тестов не следует пренебрегать. Вы пишете тесты не только для компьютера, который будет их выполнять, но и для человека, который станет их читать. Вопросы удобочитаемости будут рассмотрены в следующей главе.

Если читатель тестов должен будет тратить на поиск нужного кода больше времени, чем на его понимание, то по мере увеличения размера кода проблема сопровождения встанет во весь рост. Поэтому, возможно, стоить поискать другой способ организации.

Один тестовый класс на одну функцию

Альтернатива – создать отдельный тестовый класс для функции приложения (которая может и совпадать с методом). Этот паттерн также упомянут в книге Мезароша. Если в тестовом классе так много методов, что его трудно читать, найдите метод или группу методов, тесты которых оттесняют все остальные на второй план, и создайте для них отдельный тестовый класс с именем, соответствующим функции.

Пусть в классе `LoginManager` имеется метод `ChangePassword`, который требуется протестировать, но возможных случаев так много, что вы хотите завести для него отдельный класс. В результате получится два тестовых класса: `LoginManagerTests`, который содержит все прочие тесты, и `LoginManagerTestsChangePassword`, который содержит только тесты метода `ChangePassword`.

7.4.3. Соответствие между тестами и точками входа в единицу работы

Одна из ваших главных целей – не только сделать имена тестов удобочитаемыми и понятными, но легко находить все тестовые методы, относящиеся к конкретной единице работы. Поэтому имена тестов должны быть осмысленными. Можно сделать частью имени теста имя исходного открытого метода.

Можно было бы назвать тест `ChangePassword_scenario_expectedbehavior`. Это соглашение обсуждалось в главе (раздел 2.3.2). Бывает так, что в продуктовом коде нежелательно использовать приемы внедрения, рассмотренные в предыдущих главах, например выделение интерфейсов или переопределение виртуальных методов. Это случается, когда речь идет о сквозной функциональности (cross-cutting concerns).

7.5. Внедрение сквозной функциональности

Если использовать вышеупомянутые приемы в отношении сквозной функциональности, например управления временем, обработки исключений или протоколирования, то может получиться код, неудобный для чтения и сопровождения.

Проблема в том, что если сквозные функции, например из класса `DateTime`, вообще используются в приложении, то они встречаются в таком количестве, что, оформив их все в виде, допускающем внедрение, как детали конструктора Lego, мы получим код, который будет замечательно просто тестировать, но совершенно невозможно прочитать и понять.

Предположим, что приложению нужно текущее время для планирования или протоколирования, и мы хотим проверить, что приложение действительно помещает в журналы текущее время. В системе мог бы присутствовать такой код:

```
public static class TimeLogger
{
 public static string CreateMessage(string info)
 {
   return DateTime.Now.ToShortDateString() + " " + info;
 }
}
```

Если бы мы захотели повысить тестопригодность, введя интерфейс `ITimeProvider`, то его пришлось бы использовать всюду, где встречается `DateTime`. Это займет очень много времени, а ведь существуют и более простые подходы.

В примере со временем я предпочел бы создать специальный класс `SystemTime` и использовать его всюду в продуктовом коде вместо встроенного класса `DateTime`.

Этот класс и переработанный под него продуктовый код показаны в следующем листинге.

Листинг 7.1. Использование класса `SystemTime`

```
public static class TimeLogger
{
  public static string CreateMessage(string info)
  {
     return SystemTime.Now.ToShortDateString() + " " + info;
  }
}

public class SystemTime
{
  private static DateTime _date;

  public static void Set(DateTime custom)
  { _date = custom; }

  public static void Reset()
  { _date=DateTime.MinValue; }

  public static DateTime Now
  {
     get
     {
        if (_date != DateTime.MinValue)
        {
           return _date;
        }
        return DateTime.Now;
     }
  }
}
```

В продуктовом коде используется SystemTime

SystemTime позволяет изменять текущее время...

... и сбрасывать текущее время

SystemTime возвращает реальное текущее время или подставное, если оно было установлено

Нехитрый трюк состоит в том, что в классе `SystemTime` имеются специальные функции, позволяющие изменять текущее время во всей системе. Поэтому всюду, где используется этот класс, будут видны те дата и время, которые мы установили.

В результате мы получаем прекрасный способ проверить, что в продуктовом коде действительно используется текущее время. Он показан в листинге ниже.

Листинг 7.2. Тест с использованием `SystemTime`

```
[TestFixture]
public class TimeLoggerTests
{
  [Test]
  public void SettingSystemTime_Always_ChangesTime()
  {
     SystemTime.Set(new DateTime(2000,1,1));
     string output = TimeLogger.CreateMessage("a");
     StringAssert.Contains("01.01.2000", output);
  }

  [TearDown]
  public void afterEachTest()
  {
     SystemTime.Reset();
  }
}
```

Устанавливаем поддельную дату

Сбрасываем дату в конце каждого теста

Дополнительное преимущество в том, что не требуется внедрять в приложение миллион интерфейсов. А цена за это невысока – простенький метод с атрибутом `[TearDown]` в тестовом классе, который гарантирует, что ни один тест не изменит время, видимое другим тестам.

Однако следует учитывать, что выходная строка зависит от текущей культуры, действующей в системе (например, en-US или ru-RU). В таком случае можно также снабдить тест атрибутом `CultureInfoAttribute` (в случае NUnit), чтобы он работал в контексте конкретной культуры.

Такой способ внешнего абстрагирования сквозной функциональности позволяет создавать в продуктовом коде одну крупную точку подделывания вместо множества мелких. Но это разумно только для таких вещей, которые пронизывают систему насквозь. Если распространить эту практику и на все остальное тоже, то получится код, который очень трудно читать, – а именно этого мы и хотели избежать.

Видя этот пример, многие разработчики задают мне вопрос: «Как заставить всех пользоваться этим классом?». Я отвечаю, что во время анализа кода я слежу за тем, чтобы никто не использовал класс `DateTime` напрямую. Я стараюсь не слишком полагаться на инстру-

менты, потому что считаю, что по-настоящему чему-то научиться можно, лишь когда два (или больше) человека садятся рядом, чтобы видеть и слышать друг друга и по очереди работать за одной клавиатурой, обсуждая код. Если же речь идет об уже существующем проекте, который мы хотим перевести на использование `SystemTime`, то я просто воспользуюсь функцией «поиск в файлах», найду все вхождения `DateTime` и, если возможно, заменю их. Имя `SystemTime` выбрано так, чтобы упростить поиск и замену.

Далее мы обсудим разработку API тестов приложения.

7.6. Разработка API тестов приложения

Начав писать тесты, вы рано или поздно придете к необходимости подвергать их рефакторингу, создавать служебные методы и классы и многие другие конструкции (как в тестовых проектах, так и в тестируемом коде) исключительно ради тестопригодности или удобства чтения и сопровождения тестов.

Вот перечень того, что можно сделать:

- реализовать наследование тестовых классов для различных целей, в том числе повторного использования;
- написать служебные классы и методы;
- оповестить о предлагаемом API других разработчиков.

Рассмотрим все по порядку.

7.6.1. Наследование тестовых классов

Один из самых сильных аргументов в пользу объектно-ориентированного программирования – возможность повторно использовать уже имеющуюся функциональность, а не изобретать ее снова и снова в других классах. В книге «The Pragmatic Programmer»[2] (Addison-Wesley Professional, 1999) Энди Хант и Дэйв Томас назвали это принципом DRY («don't repeat yourself» – не повторяйся). Поскольку автономные тесты, которые мы пишем в .NET и в большинстве объектно-ориентированных языков тоже опираются на объектно-ориентированную парадигму, то не преступление использовать наследование и в самих тестах. На самом деле, я даже всячески рекомендую так

[2] Э. Хант, Д. Томас «Программист-прагматик. Путь от подмастерья к мастеру». Лори, 2009. – *Прим. перев.*

поступать, если нет основательных контрдоводов. Реализация базового класса поможет преодолеть типичные проблемы при написании тестов за счет:

- повторного использования служебных и фабричных методов;
- прогона одного и того же набора тестов для разных классов (мы рассмотрим этот момент подробнее);
- использования общего кода подготовки и очистки (полезно также для интеграционного тестирования);
- задания направления тестирования для программистов, которые унаследуют вашему базовому классу.

Я познакомлю вас с тремя паттернами на основе наследования тестовых классов, каждый из которых опирается на предыдущий. Я также объясню, когда использовать тот или иной паттерн, и расскажу об их плюсах и минусах.

Вот эти три основных паттерна:

- абстрактный тестовый инфраструктурный класс;
- шаблонный тестовый класс;
- абстрактный тестовый управляющий класс.

Мы также рассмотрим следующие приемы рефакторинга, полезные в случае применения вышеупомянутых паттернов:

- создание иерархии классов;
- использование универсальных типов.

Паттерн абстрактный тестовый инфраструктурный класс

Этот паттерн подразумевает создание абстрактного тестового класса, который содержит необходимую инфраструктуру, общую для производных от него классов. Использовать такой класс можно в самых разных ситуациях: от общего кода подготовки и очистки до специальных утверждений, которые высказываются в нескольких тестовых классах.

Мы рассмотрим пример повторного использования метода подготовки в двух тестовых классах. Постановка задачи такова: во всех тестах необходимо переопределить подразумеваемую по умолчанию реализацию класса ведения журнала, так чтобы журнал записывался не в файл, а в память (то есть во всех тестах необходимо разорвать зависимость от настоящего журнала).

Интересующие нас классы перечислены ниже.

- *Класс и метод* `LogAnalyzer` – это тот класс и метод, который требуется протестировать.
- *Класс* `LoggingFacility` – класс, где находится реализация регистратора, которую требуется переопределить в тестах.
- *Класс* `ConfigurationManager` – еще один пользователь класса `LoggingFacility`, который мы будем тестировать позже.
- *Класс и метод* `LogAnalyzerTests` – тестовый класс и метод, которые нам предстоит написать сначала.
- *Класс* `ConfigurationManagerTests` – класс с тестами `ConfigurationManager`.

Листинг 7.3. Пример игнорирования принципа DRY в тестовых классах

```
// Внутри этого класса используется LoggingFacility
public class LogAnalyzer
{
  public void Analyze(string fileName)
  {
     if (fileName.Length < 8)
     {
        LoggingFacility.Log("Слишком короткое имя файла: " + fileName);
     }
     // остальная часть метода
  }
}

// Другой класс, внутри которого используется LoggingFacility
public class ConfigurationManager
{
  public bool IsConfigured(string configName)
  {
     LoggingFacility.Log("проверяется " + configName);
     return result;
  }
}

public static class LoggingFacility
{
  public static void Log(string text)
  {
     logger.Log(text);
  }
  private static ILogger logger;

  public static ILogger Logger
  {
     get { return logger; }
```

```
      set { logger = value; }
   }
}

[TestFixture]
public class LogAnalyzerTests
{
   [Test]
   public void Analyze_EmptyFile_ThrowsException()
   {
      LogAnalyzer la = new LogAnalyzer();
      la.Analyze("myemptyfile.txt");
      // остальная часть теста
   }

   [TearDown]
   public void teardown()
   {
      // необходимо сбрасывать статический ресурс между тестами
      LoggingFacility.Logger = null;
   }
}

[TestFixture]
public class ConfigurationManagerTests
{
   [Test]
   public void Analyze_EmptyFile_ThrowsException()
   {
      ConfigurationManager cm = new ConfigurationManager();
      bool configured = cm.IsConfigured("something");
      // остальная часть теста
   }

   [TearDown]
   public void teardown()
   {
      // необходимо сбрасывать статический ресурс между тестами
      LoggingFacility.Logger = null;
   }
}
```

Класс `LoggingFacility`, скорее всего, будет использоваться во многих классах. Он спроектирован так, чтобы использующий его код можно было протестировать, подменив реализацию регистратора путем установки статического свойства.

Имеются два класса, в которых используется `LoggingFacility`: `LogAnalyzer` и `ConfigurationManager`, и мы бы хотели протестировать оба.

Один из способов улучшить этот код – выделить служебный метод, чтобы устранить повторы в двух тестовых классах. Оба они подделывают подразумеваемую по умолчанию реализацию регистратора. Мы можем включить служебный метод в базовый тестовый класс и вызывать его из производных.

Мы не хотим включать метод `[SetUp]` в базовый класс, потому что тогда производные стали бы менее понятными. Вместо этого мы напишем служебный метод `FakeTheLogger()`. Полный код тестовых классов приведен ниже.

Листинг 7.4. Код после рефакторинга

```
[TestFixture]
public class BaseTestsClass
{
  public ILogger FakeTheLogger()
  {
     LoggingFacility.Logger = Substitute.For<ILogger>();
     return LoggingFacility.Logger;
  }

  [TearDown]
  public void teardown()
  {
     // необходимо сбрасывать статический ресурс между тестами
     LoggingFacility.Logger = null;
  }
}

[TestFixture]
public class ConfigurationManagerTests:BaseTestsClass
{
  [Test]
  public void Analyze_EmptyFile_ThrowsException()
  {
     FakeTheLogger();

     ConfigurationManager cm = new ConfigurationManager();
     bool configured = cm.IsConfigured("something");
     // остальная часть теста
  }
}

[TestFixture]
public class LogAnalyzerTests : BaseTestsClass
{
  [Test]
  public void Analyze_EmptyFile_ThrowsException()
```

Выделяем общий служебный метод, который будет использоваться в производных классах

Автоматическая очистка производных классов

Вызываем вспомогательный метод базового класса

```
    {
        FakeTheLogger();                    <-- Вызываем вспомогательный метод базового класса

        LogAnalyzer la = new LogAnalyzer();
        la.Analyze("myemptyfile.txt");
        // остальная часть теста
    }
}
```

Если бы мы включили в базовый класс метод с атрибутом `Setup`, то он автоматически вызывался бы перед каждым тестом в любом производном классе. Но тогда читателю производного класса было бы трудно понять, что происходит на стадии подготовки. Ему пришлось бы искать метод подготовки в базовом классе, чтобы понять, что производные классы получают по умолчанию. Поскольку при таком подходе ухудшается удобочитаемость тестов, то мы решили остановиться на более явном служебном методе.

Удобочитаемость ухудшается и потому, что коллеги, пользующиеся базовым классом, почти не имеют документации о его API. Поэтому я рекомендую применять эту технику настолько редко, насколько возможно, но не реже. Точнее, у меня никогда не было достаточных оснований для использования нескольких базовых классов. Мне всегда удавалось добиться повышения удобочитаемости с помощью одного базового класса, путь даже сопровождение становилось чуть менее удобным. Еще хочу сказать – *никогда* не используйте в тестах более одного уровня наследования. Эта мешанина станет неудобочитаемой быстрее, чем вы успеете произнести «почему сборка не работает?».

Обратимся к более интересному применению наследования для решения типичной проблемы.

Шаблонный тестовый класс

Допустим, требуется, чтобы люди, которые разрабатывают определенного вида классы, никогда не забывали создавать для них специальные наборы автономных тестов. Например, речь может идти о коде для работы с сетью, обеспечения безопасности, работы с базой данных или просто старого доброго синтаксического разбора. Идея в том, что для такого класса в тестируемом коде заведомо должны существовать некоторые тесты, потому что API этого класса предоставляет заранее известный набор услуг.

Шаблонный тестовый класс – это абстрактный класс, содержащий абстрактные тестовые методы, которые должны быть реализованы в производных классах. Движущая сила этого паттерна состоит в том,

чтобы заставить производные классы реализовать определенные тесты.

Подходящими кандидатами для этого паттерна являются классы с интерфейсами. Я обычно пользуюсь им, когда имеется расширяющаяся иерархия классов, и в каждом новом производном классе реализуются примерно одни и те же идеи.

Можно считать, что интерфейс – это контракт о поведении, то есть от всех производных классов ожидается одно и то же поведение, хотя его конкретные формы могут быть разными. Примером контракта о поведении может служить набор синтаксических анализаторов, каждый из которых реализует методы разбора, которые работают одинаково, но применяются к разным входным типам. Разработчики часто забывают или просто не хотят писать все тесты, необходимые для каждого частного случая. Наличие класса, базового для всех классов с одним и тем же интерфейсом, поможет создать базовый тестовый контракт, который все разработчики будут обязаны реализовать в производных тестовых классах.

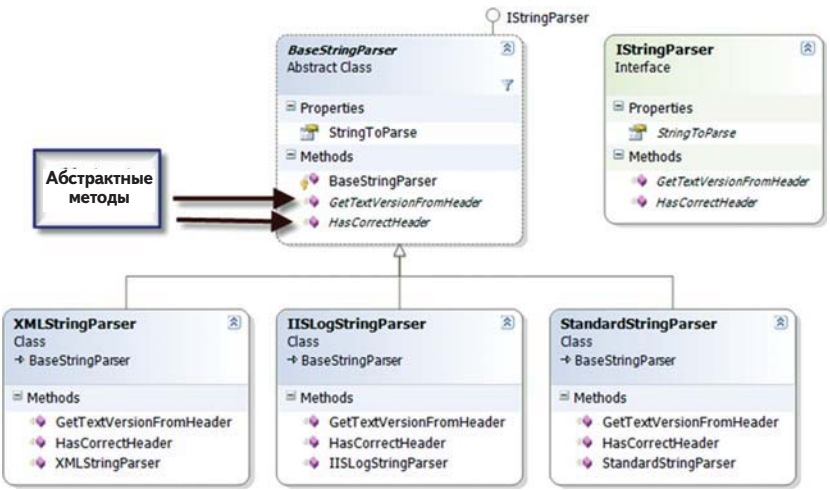


Рис. 7.3. Типичная иерархия наследования, подлежащая тестированию, состоит из абстрактного базового класса и наследующих ему классов

Рассмотрим реальную ситуацию. Допустим, что требуется протестировать объектную модель, показанную на рис. 7.3. `BaseStringParser` – абстрактный класс, которому наследуют другие классы, реализующие некоторую функциональность для строк с раз-

ным типом содержимого. Из любой строки (XML-код, строки журнала IIS, стандартные строки) можно получить какую-то информацию о версии (метаданные). Эти сведения можно извлечь из заголовка (первые несколько строчек строки) и проверить, отвечает ли заголовок целям приложения. Классы `XMLStringParser`, `IISLogStringParser` и `StandardStringParser` наследуют базовому классу и реализуют его методы, сообразуясь со структурой конкретных строк.

Чтобы протестировать такую иерархию, нужно первым делом написать набор тестов для одного из производных классов (в предположении, что в абстрактном классе нет никакой логики). Затем аналогичные тесты придется написать для других классов с такой же функциональностью.

В следующем листинге приведены тесты для класса `StandardStringParser`. С них мы начнем, прежде чем переработать тестовые классы в соответствии с паттерном шаблонного базового класса.

Листинг 7.5. Набросок тестового класса для `StandardStringParser`

```
[TestFixture]
public class StandardStringParserTests
{
  private StandardStringParser GetParser(string input)
  {
     return new StandardStringParser(input);
  }

  [Test]
  public void GetStringVersionFromHeader_SingleDigit_Found()
  {
     string input = "header;version=1;\n";
     StandardStringParser parser = GetParser(input

     string versionFromHeader = parser.GetStringVersionFromHeader();
     Assert.AreEqual("1",versionFromHeader);
  }

  [Test]
  public void GetStringVersionFromHeader_WithMinorVersion_Found()
  {
     string input = "header;version=1.1;\n";
     StandardStringParser parser = GetParser(input);
    // остальная часть теста
  }

  [Test]
```

❶ Определяем фабричный метод создания анализатора

❷ Используем фабричный метод

```
    public void GetStringVersionFromHeader_WithRevision_Found()
    {
       string input = "header;version=1.1.1;\n";
       StandardStringParser parser = GetParser(input);
      // остальная часть теста
    }
}
```

Обратите внимание, как вспомогательный метод `GetParser()` ❶ позволил абстрагировать ❷ операцию создания анализатора, присутствующую во всех тестах. Мы пользуемся этим вспомогательным методом, а не методом подготовки, потому что конструктору следует передать разбираемую строку, поскольку каждый тест создает анализатор для тестирования определенных входных данных.

В тестах для других классов из этой иерархии мы захотим повторить те же проверки, что и для данного конкретного класса. У всех остальных анализаторов точно такое же внешнее поведение: получить из заголовка номер версии и проверить, что заголовок корректен. Делают они это по-разному, но семантика не меняется. Это означает, что для любого класса, производного от `BaseStringParser`, необходимо будет написать одни и те же тесты, изменится только тип тестируемого класса.

Но начнем с начала: посмотрим, как легко продиктовать производным тестовым классам, какие тесты необходимы. В следующем листинге показан простой пример такого подхода (интерфейс `IStringParser` имеется в исходном коде к этой книге на GitHub).

Листинг 7.6. Шаблонный тестовый класс для тестирования анализаторов строк

```
[TestFixture]
public abstract class TemplateStringParserTests <-- Шаблонный тестовый класс
{
  public abstract
    void TestGetStringVersionFromHeader_SingleDigit_Found();

  public abstract
    void TestGetStringVersionFromHeader_WithMinorVersion_Found();

  public abstract
    void TestGetStringVersionFromHeader_WithRevision_Found();
}

[TestFixture]                                   Производный класс
public class XmlStringParserTests : TemplateStringParserTests <--
{
```

```
    protected IStringParser GetParser(string input)
    {
       return new XMLStringParser(input);
    }

    [Test]
    public override
      void TestGetStringVersionFromHeader_SingleDigit_Found()
    {
       IStringParser parser = GetParser("<Header>1</Header>");

       string versionFromHeader = parser.GetStringVersionFromHeader();
       Assert.AreEqual("1",versionFromHeader);
    }

    [Test]
    public override
      void TestGetStringVersionFromHeader_WithMinorVersion_Found()
    {
       IStringParser parser = GetParser("<Header>1.1</Header>");

       string versionFromHeader = parser.GetStringVersionFromHeader();
       Assert.AreEqual("1.1",versionFromHeader);
    }

    [Test]
    public override
      void TestGetStringVersionFromHeader_WithRevision_Found()
    {
       IStringParser parser = GetParser("<Header>1.1.1</Header>");

       string versionFromHeader = parser.GetStringVersionFromHeader();
       Assert.AreEqual("1.1.1",versionFromHeader);
    }
}
```

Этот код наглядно представлен на рис. 7.4, где мы видим три производных класса. Отметим, что `GetParser()` – не унаследованный метод, в производных классах его можно назвать как угодно.

Меня эта техника выручала во многих ситуациях – и в роли разработчика, и в роли архитектора. Как архитектор, я предлагал разработчикам список наиболее существенных базовых классов и указывал, какие тесты они должны написать. В такой ситуации важно присваивать тестам понятные имена. Я начинаю имена абстрактных методов базового класса словом `Test`, чтобы тем, кто переопределяет их в производных классах, было проще искать, что именно нужно переопределить.

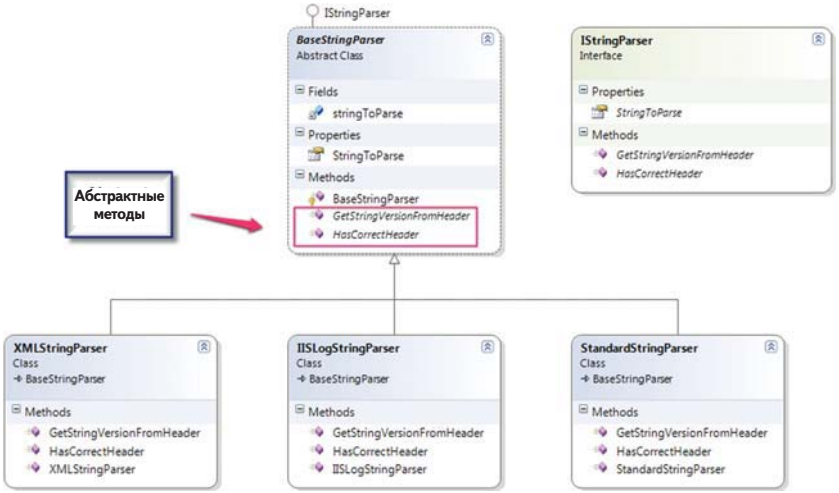


Рис. 7.4. Паттерн шаблонного базового класса гарантирует, что разработчики не забудут о важных тестах. Базовый класс содержит абстрактные тестовые методы, которые должны быть реализованы в производных классах

Но нельзя ли поручить базовому классу побольше работы?

Паттерн абстрактного управляющего тестового класса, в котором нужно «заполнить недостающее»

Паттерн абстрактного управляющего тестового класса (я называю его «заполни недостающее») является развитием изложенной выше идеи. Только теперь логика тестов реализуется в самом базовом классе, но содержит абстрактные методы-точки подключения, которые должны быть реализованы в производных классах.

Важно, чтобы тесты относились не к одному конкретному типу тестируемого класса, а были написаны в соответствии с интерфейсом или базовым классом в продуктовом коде.

Ниже приведен пример такого базового класса.

Листинг 7.7. Базовый тестовый класс «заполни недостающее»

```
public abstract class FillInTheBlanksStringParserTests
{
  protected abstract IStringParser GetParser(string input);
```

Абстрактный фабричный метод, который должен вернуть интерфейс

Абстрактные методы ввода, которые в производных классах возвращают данные в определенном формате

```
protected abstract string HeaderVersion_SingleDigit { get; }
protected abstract string HeaderVersion_WithMinorVersion {get;}
protected abstract string HeaderVersion_WithRevision { get; }
public const string EXPECTED_SINGLE_DIGIT = "1";
public const string EXPECTED_WITH_REVISION = "1.1.1";
public const string EXPECTED_WITH_MINORVERSION = "1.1";
```

Предопределенные ожидаемые результаты для производных классов

```
[Test]
public void GetStringVersionFromHeader_SingleDigit_Found()
{
    string input = HeaderVersion_SingleDigit;
    IStringParser parser = GetParser(input);

    string versionFromHeader = parser.GetStringVersionFromHeader();
    Assert.AreEqual(EXPECTED_SINGLE_DIGIT,versionFromHeader);
}
```

Общая логика тестов фиксирована, но ввод реализован в производных классах

```
[Test]
public void GetStringVersionFromHeader_WithMinorVersion_Found()
{
    string input = HeaderVersion_WithMinorVersion;
    IStringParser parser = GetParser(input);

    string versionFromHeader = parser.GetStringVersionFromHeader();
    Assert.AreEqual(EXPECTED_WITH_MINORVERSION,versionFromHeader);
}

[Test]
public void GetStringVersionFromHeader_WithRevision_Found()
{
    string input = HeaderVersion_WithRevision;
    IStringParser parser = GetParser(input);

    string versionFromHeader = parser.GetStringVersionFromHeader();
    Assert.AreEqual(EXPECTED_WITH_REVISION,versionFromHeader);
}
}
```

Производный класс заполняет недостающее

```
[TestFixture]
public class StandardStringParserTests : FillInTheBlanksStringParserTests
{
  protected override string HeaderVersion_SingleDigit
  {
     get {
        return string.Format("header\tversion={0}\t\n",
           EXPECTED_SINGLE_DIGIT);
     }
  }
```

Возврат данных в требуемом формате

```
    protected override string HeaderVersion_WithMinorVersion
    {
        get {
            return string.Format("header\tversion={0}\t\n",
                EXPECTED_WITH_MINORVERSION);
        }
    }

    protected override string HeaderVersion_WithRevision
    {
        get {
            return string.Format("header\tversion={0}\t\n",
                EXPECTED_WITH_REVISION);
        }
    }

    protected override IStringParser GetParser(string input)
    {
        return new StandardStringParser(input);
    }
}
```

Возврат экземпляра тестируемого класса нужного типа

В этом листинге в производных классах нет тестов. Они унаследованы. Но при необходимости можно было бы добавить в производные классы дополнительные тесты. На рис. 7.5 показана получившаяся цепочка наследования.

Как следует модифицировать существующий код, чтобы можно было воспользоваться этим паттерном? Об этом в следующем разделе.

Преобразование тестового класса в иерархию тестовых классов

Большинство разработчиков вначале не учитывают эти иерархии наследования, а пишут тесты обычным способом, как показано в листинге 7.7. Преобразовать уже написанные тесты в базовый класс сравнительно просто, особенно если в IDE имеются средства рефакторинга, как, например, в Eclipse, IntelliJ IDEA или в Visual Studio (ReSharper от компании JetBrains, JustCode от Telerik или Refactor! от DevExpress).

Ниже приведен список шагов рефакторинга тестового класса.

1. Рефакторинг: выделить суперкласс.
   - создать базовый класс (`BaseXXXTests`);
   - переместить фабричные методы (например, `GetParser`) в базовый класс;

- переместить все тесты в базовый класс;
- сделать ожидаемые результаты открытыми полями базового класса;
- перенести тестовые входные данные в методы или свойства, реализованные в производных классах.

2. Рефакторинг: сделать фабричные методы абстрактными, возвращающими интерфейсы.
3. Рефакторинг: найти в тестовых методах все места, в которых используются явные типы классов, и заменить их интерфейсами.
4. В производных классах реализовать абстрактные фабричные методы, возвращая из них экземпляры явных типов.

Для создания паттернов наследования можно также воспользоваться универсальными типами .NET.

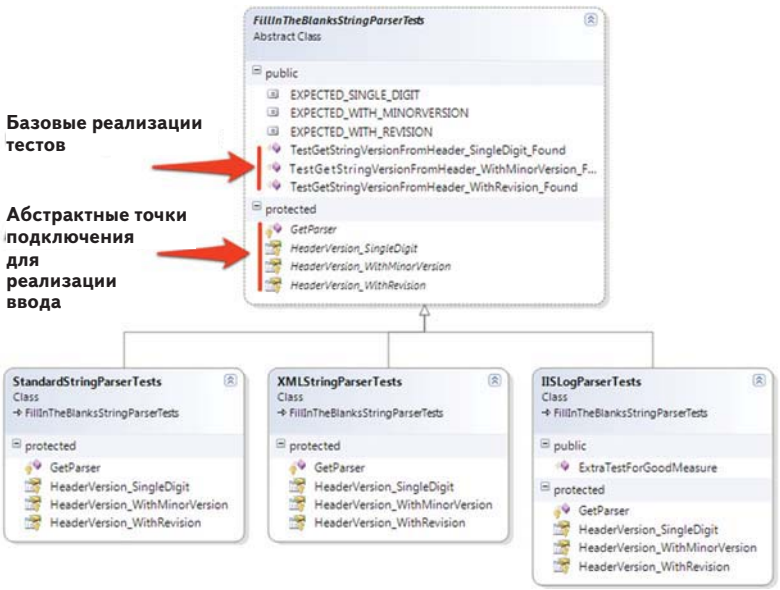


Рис. 7.5. Реализация иерархии тестовых классов. Большинство тестов находится в базовом классе, но в производных могут быть и дополнительные тесты

Использование универсальных типов .NET для реализации иерархии тестов

В базовом тестовом классе можно использовать универсальные типы. Тогда в производных классах не придется переопределять фаб-

ричный метод порождения экземпляра, достаточно будет объявить тестируемый тип. В следующем листинге показан вариант универсального базового тестового класса и производного от него.

Листинг 7.8. Реализация наследования тестов с применением универсальных типов .NET

```
// Реализация той же идеи с помощью универсальных типов
public abstract class GenericParserTests<T>
  where T:IStringParser                          ❶ Определяем ограничение на параметр универсального типа
{
  protected abstract string GetInputHeaderSingleDigit();

  protected T GetParser(string input)             ❷ Возвращает объект типа универсального класса, а не интерфейса
  {
     return (T) Activator.CreateInstance(typeof (T), input);
  }                                               ❸ Возвращает экземпляр универсального класса

  [Test]
  public void GetStringVersionFromHeader_SingleDigit_Found()
  {
     string input = GetInputHeaderSingleDigit();
     T parser = GetParser(input);

     bool result = parser.HasCorrectHeader();
     Assert.IsFalse(result);
  }

  // другие тесты
  //...
}

// Пример теста, наследующего универсальному базовому классу
[TestFixture]
public class StandardParserGenericTests           ❹ Наследует универсальному базовому классу
     :GenericParserTests<StandardStringParser>
{
  protected override string GetInputHeaderSingleDigit()
  {
     return "Header;1";                           ❺ Возвращает входные данные для конкретного тестируемого типа
  }
}
```

В универсальной реализации иерархии изменилось несколько вещей.

- Фабричный метод `GetParser` ❷ больше не переопределяется. Мы просто создаем объект методом `Activator.CreateInstance`, который позволяет создавать объекты, зара-

нее не зная их типа, и передаем входную строку конструктору наряду с типом `T` ❸.

- В самих тестах используется не интерфейс `IStringParser`, а конкретизация универсального типа ❹.
- В объявлении универсального класса имеется часть `where`, где объявляется, что тип `T` должен реализовывать интерфейс `IStringParser` ❶.
- В производном классе реализован метод, возвращающий конкретные входные данные ❺.

Вообще говоря, я не вижу особых преимуществ в использовании универсальных базовых классов. Возможный выигрыш в производительности для тестов несущественен. Решайте сами, какой вариант вам больше нравится. Это вопрос вкуса, и ничего больше.

Теперь перейдем к совершенно другой теме: инфраструктуре тестовых проектов.

7.6.2. Создание служебных классов и методов для тестов

При написании тестов создается много простеньких служебных методов, которые, возможно, будут вынесены в отдельные классы (а, возможно, и не будут). Эти служебные классы могут стать существенной частью API тестов и даже превратиться в простую объектную модель, используемую при разработке тестов. Служебные методы можно отнести к следующим категориям:

- фабричные методы для объектов, которые трудно создать или которые раз за разом создаются во многих тестах;
- методы инициализации системы (например, для установки состояния системы перед тестированием или настройки средств протоколирования на использование регистраторов-заглушек);
- методы конфигурирования объектов (например, для установки внутреннего состояния объекта, скажем, чтобы сделать объект-заказчик непригодным для какой-то операции);
- методы подготовки или чтения внешних ресурсов, например баз данных, конфигурационных файлов или тестовых входных файлов (скажем, метод, загружающий текстовый файл со всеми комбинациями входных данных, передаваемых некоторому методу, и ожидаемых от него результатов). Чаще такие

методы встречаются при интеграционном или комплексном тестировании;
- специальные методы утверждения, в которых высказывается утверждение о состоянии системы, достаточно сложное или встречающееся многократно (если что-то записывается в системный журнал, то метод может утверждать, что X, Y и Z истинны, а G – нет.)

Служебные методы можно объединить в служебные классы, например:

- служебные классы утверждений, содержащие все специальные методы утверждения;
- классы, содержащие фабричные методы;
- конфигурационные классы (в том числе для конфигурирования баз данных), относящиеся к интеграционному тестированию.

Для платформы .NET существует несколько служебных каркасов с открытым исходным кодом, предлагающих элегантные способы решения такого рода задач. Один из них – *каркас Fluent Assertions* – находится по адресу https://github.com/dennisdoomen/FluentAssertions.

Само наличие служебных методов еще не означает, что кто-то будет ими пользоваться. Я видел немало проектов, в которых разработчики раз за разом изобретали велосипед, создавая служебные методы, которые уже были написаны раньше, только никто о них не знал.

Далее мы поговорим о том, как распространить сведения о своем API.

7.6.3. Извещение разработчиков об имеющемся API

Важно, чтобы авторы тестов знали о различных API, созданных по мере разработки приложения и тестов для него. Есть несколько способов распространить сведения о написанном вами API.

- Посадить за написание тестов двух человек или группы по два человека (хотя бы ненадолго), из которых один знаком с существующими API и может рассказать другому об их достоинствах.
- Составить короткий документ (не более чем на пару страниц) или шпаргалку, в которой описывается, какие API существуют и где их найти. Можно готовить короткие документы по каж-

дой части системы тестирования (например, об API, относящихся к уровню работы с данными) или глобальный документ, охватывающий все приложение. Если документ длинный, никто не станет его сопровождать. Один из способов обеспечить актуальность – автоматизировать процедуру генерации документа:

  - договориться о наборе префиксов или суффиксов имен вспомогательных API;
  - написать специальный инструмент, который ищет имена API и генерирует документ, содержащий их перечень и сведения о местонахождении; альтернативно можно включать специальные комментарии с директивами, которые инструмент может извлечь;
  - включить генерацию документа в состав автоматизированной процедуры сборки.

- Обсуждать изменения API на собраниях команды – одна-две фразы касательно сути основных изменений и того, где их искать. Тогда команда будет знать, что это важно, а этим никогда пренебрегать не следует.
- Прорабатывать этот документ со всеми вновь принятыми на работу на стадии ознакомления.
- Проводить анализ тестов (в дополнение к анализу кода), обращая внимание на стандарты удобочитаемости, удобства сопровождения, правильности и на использование подходящих API в нужных местах. Подробнее об этой практике см. статью http://5whys.com/blog/step-4-start-doing-code-reviews-seriously.html в моем блоге для руководителей программистских коллективов.

Следование хотя бы некоторым из этих советов поможет поддерживать продуктивность команды и создать общий язык, на котором все пишут тесты.

7.7. Резюме

Подведем итоги тому, что следует вынести из только что прочитанной главы.

- Каким бы видом тестирования вы ни занимались, автоматизируйте эту деятельность; запускайте автоматизированную процедуру сборки как можно чаще – днем или ночью – и как можно чаще проводите непрерывную интеграцию продукта.

- Отделяйте интеграционные тесты от автономных (медленные тесты от быстрых), чтобы у команды была безопасная зеленая зона, в которой все тесты должны проходить.
- Классифицируйте тесты по проекту и по типу (автономные и интеграционные, медленные и быстрые) и разносите их по разным каталогам, папкам или пространствам имен (или по всему вместе). Я обычно применяю все три типа разделения.
- Используйте иерархию тестовых классов для применения одного и того же набора тестов к нескольким взаимосвязанным тестируемым типам, организованным иерархически, или к типам, имеющим общий интерфейс или базовый класс.
- Применяйте вспомогательные и служебные классы вместо иерархий, если организация иерархии делает тесты менее удобочитаемыми, а особенно когда в базовом классе оказывается общий метод подготовки. Существуют разные точки зрения на то, когда использовать каждый из этих способов, но обычно главным аргументом против иерархий служит неудобочитаемость.

Извещайте о написанном вами API других членов команды. В противном случае вы будете просто транжирить время и деньги, потому что коллеги по незнанию будут изобретать одни и те же API снова и снова.

ГЛАВА 8. Три столпа хороших автономных тестов

В этой главе:

- Написание заслуживающих доверия тестов.
- Написание удобных для сопровождения тестов.
- Написание удобочитаемых тестов.
- Соглашения об именовании автономных тестов.

Как бы вы ни организовывали свои тесты и сколько бы их ни было, грош им цена, если им нельзя доверять, неудобно сопровождать или невозможно читать. Чтобы тест считался хорошим, он должен обладать тремя свойствами.

- *Высокая степень доверия.* Разработчики хотят, чтобы результатам тестов можно было доверять. В заслуживающих доверия тестах нет ошибок, и они тестируют именно то, что надо.
- *Удобство сопровождения.* Непригодные для сопровождения тесты – сущий кошмар, потому что из-за них срываются сроки, а если график работ над проектом ужесточается, то их просто откладывают в сторонку. Разработчики перестают сопровождать и исправлять тесты, которые слишком долго изменять или приходится модифицировать при малейшем изменении продуктового кода.
- *Удобочитаемость.* Это означает, что тест легко не только прочитать, но и понять, в чем проблема, если он выдает странные результаты. Без удобочитаемости два остальных столпа очень быстро подламываются. Сопровождать тесты становится труднее, а доверять тому, чего не понимаешь, невозможно.

В этой главе даются рекомендации по каждому из этих столпов. Вы можете использовать их в ходе анализа тестов. В совокупности эти три столпа гарантируют, что вы не зря тратите время. Уберите любой из них – и появится риск, что все, что вы делаете, пойдет насмарку.

8.1. Написание заслуживающих доверия тестов

О том, заслуживает ли тест доверия, можно судить по нескольким признакам. Если тест проходит, вы не говорите себе: «А пройду-ка я этот код в отладчике, чтобы уж наверняка». Вы верите, что раз тест прошел, то тестируемый код в конкретном случае работает правильно. Если тест не проходит, вы не говорите себе: «А, да он и не должен был пройти» или «Это еще не значит, что код не работает». Вы верите, что ошибка именно в коде, а не в тесте. Короче говоря, тест достоин доверия, если вы уверены, что знаете, что происходит и как можно вмешаться.

В этой главе я дам некоторые рекомендации и опишу приемы, которые помогут вам:

- принять решение о том, когда удалять или изменять тесты;
- избежать наличия логики в тестах;
- тестировать только один результат;
- разделять автономные и интеграционные тесты;
- уделять критическому анализу кода столько же внимания, сколько покрытию кода тестами.

По моему опыту, тесты, следующие этим рекомендациям, в большей мере заслуживают доверия; я чувствую уверенность, что они и дальше будут находить ошибки в моем коде.

8.1.1. Когда удалять или изменять тесты

Написав тесты и добившись того, что они проходят, мы, вообще говоря, не должны ни удалять, ни изменять их. Это страховочная сетка, благодаря которой мы узнаем, что очередная модификация кода привела к поломке. Тем не менее, бывают случаи, когда мы вынуждены удалять или изменять имеющиеся тесты. Чтобы понять, когда это имеет смысл, рассмотрим причины того и другого. Почему удаляется тест? Главным образом, потому что он не проходит. Перечислим возможные причины внезапного отказа теста.

- *Ошибки в продуктовом коде*. В тестируемом продуктовом коде имеется ошибка.
- *Ошибки в тесте*. В тесте имеется ошибка.
- *Изменение семантики или API*. Изменилась семантика, но не функциональность тестируемого кода.
- *Конфликтующие или недействительные тесты*. Продуктовый код был изменен с целью удовлетворения требования, противоречащего предыдущим.

Для удаления или изменения тестов есть также причины, не связанные с ошибками в коде или тесте:

- чтобы переименовать или переработать тест;
- чтобы исключить дублирующиеся тесты.

Рассмотрим каждый из этих случаев.

Ошибки в продуктовом коде

Ошибка в продуктовом коде имеет место, когда после изменения продуктового кода какой-то тест перестает проходить. Если это действительно ошибка в тестируемом коде, то с тестом все в порядке и трогать его не нужно. Это наилучший и желательный результат, ради которого тесты и пишутся.

Поскольку обнаружение ошибок в продуктовом коде и есть цель автономного тестирования, то остается только исправить эту ошибку. Тест не трогайте.

Ошибки в тесте

Если ошибка вкралась в тест, то тест необходимо изменить. Ошибки в тестах особенно трудно искать, потому что предполагается, что тесты правильны. (Именно поэтому я так люблю TDD. Это дополнительный способ протестировать сам тест и убедиться, что он падает и проходит именно тогда, когда должен.) Я наблюдал несколько стадий, через которые проходят разработчики, обнаружив ошибку в тесте.

1. *Отрицание*. Разработчик продолжает искать ошибку в самом коде, изменяет его, в результате чего перестают работать все остальные тесты. Разработчик вносит в продуктовый код новые ошибки, охотясь за той, что на самом деле находится в тесте.
2. *Развлечение*. Если есть возможность, разработчик зовет коллегу, и они вместе продолжают искать несуществующую ошибку.

3. *Отладка*. Разработчик терпеливо отлаживает тест и обнаруживает, что ошибка затесалась в тест. На это может уйти от часа до нескольких дней.
4. *Признание и битье головой об стенку.* Разработчик наконец осознает, где ошибка, и начинает лупить себя по лбу.

Обнаружив и начав исправлять ошибку, убедитесь, что ошибка действительно исправлена и что тест волшебным образом проходит не потому, что вы тестируете что-то совсем другое. Необходимо сделать следующее:

1. Исправить ошибку в тесте.
2. Проверить, что тест падает, когда должен падать.
3. Проверить, что тест проходит, когда должен проходить.

Первый шаг – исправление ошибки в тесте – затруднений не вызывает. А дальше нужно убедиться, что тестируется то, что надо, и что тесту по-прежнему можно доверять.

Исправив тест, переходите к продуктовому коду и измените его так, чтобы проявилась ошибка, которую тест должен отловить. Например, для этого может понадобиться закомментировать какую-нибудь строку или изменить булево значение. Затем выполните тест. Если тест падает, значит, он наполовину работает. Вторая половина проверяется на шаге 3. Если тест не падает, значит, вы, скорее всего, тестируете не то, что надо. (Я видел, как разработчики случайно удаляли из теста утверждения в ходе исправления ошибки. Вы не поверите, как часто такое случается и насколько эффективным в таких случаях оказывается шаг 2.)

Убедившись, что тест падает, уберите ошибку из продуктового кода. Теперь тест должен проходить. Если это не так, то либо ошибка в тесте осталась, либо вы тестируете что-то не то. Обязательно нужно добиться, чтобы тест сначала не прошел, а потом – после устранения ошибки – снова прошел.

Изменение семантики или API

Тест может отказать, если продуктовый код изменяется таким образом, что тестируемый объект должен использоваться по-другому, пусть даже его конечная функциональность сохраняется.

В листинге ниже приведен простой тест.

Листинг 8.1. Простой тест класса `LogAnalyzer`

```
[Test]
public void SemanticsChange()
```

```
{
  LogAnalyzer logan = new LogAnalyzer();

  Assert.IsFalse(logan.IsValid("abc"));
}
```

Предположим, что мы изменили семантику класса `LogAnalyzer`, добавив метод `Initialize`. Теперь до вызова любого метода класса `LogAnalyzer` необходимо инициализировать его путем обращения к методу `Initialize`.

Если внести такое изменение в продуктовый код, то утверждение в листинге 8.1 возбудит исключение, потому что метод `Initialize` не вызывался. Тест не пройдет, хотя он по-прежнему корректный. Проверяемая им функциональность работает, но семантика тестируемого объекта изменилась.

В этом случае тест необходимо привести в соответствие с новой семантикой, как показано ниже.

Листинг 8.2. Тест класса `LogAnalyzer` после приведения в соответствие с новой семантикой

```
[Test]
public void SemanticsChange()
{
  LogAnalyzer logan = new LogAnalyzer();
  logan.Initialize();

  Assert.IsFalse(logan.IsValid("abc"));
}
```

Именно изменение семантики как правило является причиной недовольства разработчиков при написании и сопровождении автономных тестов, поскольку объем изменений в тестах при изменении API продуктового кода становится все больше и больше. В следующем листинге показана более удобная для сопровождения версия этого теста.

Листинг 8.3. Тест после рефакторинга – используется фабричный метод

```
[Test]
public void SemanticsChange()
{
  LogAnalyzer logan = MakeDefaultAnalyzer();

  Assert.IsFalse(logan.IsValid("abc"));
}

public static LogAnalyzer MakeDefaultAnalyzer()
```

❶ Используется фабричный метод

```
{
  LogAnalyzer analyzer = new LogAnalyzer();
  analyzer.Initialize();
  return analyzer;
}
```

В данном случае в переработанном тесте используется служебный фабричный метод ❶. Этот метод можно использовать и в других тестах. Если впоследствии семантика создания и инициализации объекта снова изменится, то не понадобится изменять все тесты, где создается объект; достаточно будет модифицировать лишь небольшой служебный метод. Если вам наскучит создавать такие фабричные методы, рекомендую обратить внимание на вспомогательный каркас AutoFixture.

Я еще уделю этому каркасу внимание в приложение, но, если в двух словах, то AutoFixture в числе прочего может использоваться как умная фабрика объектов, которая позволяет создать тестируемый объект, не особенно заботясь о структуре конструктора. Дополнительные сведения об этом каркасе можно почерпнуть из статьи Google «String Calculator Kata with AutoFixture» или на странице https://github.com/AutoFixture/AutoFixture. Я еще не уверен, стану ли заядлым сторонником этого каркаса (поскольку создать фабричный метод — не такое большое дело), но познакомиться с ним и решить, нравится он вам или нет, в любом случае стоит. При условии, что это не вредит удобству чтения и сопровождения тестов, я вас удерживать не стану.

Ниже в этой главе мы познакомимся с другими приемами написания удобных для сопровождения тестов.

Конфликтующие и недействительные тесты

Конфликт возникает, когда в продуктовом коде появляется новая функция, прямо противоречащая тесту. Это означает, что тест теперь находит не ошибку, а конфликтующие требования.

Рассмотрим простой пример. Пусть заказчик требует, чтобы `LogAnalyzer` не принимал файлы, имя которых содержит меньше четырех букв. В этом случае анализатор должен возбуждать исключение. Функция реализована, и для нее написаны тесты.

Много позже заказчик приходит к выводу, что трехбуквенные имена файлов тоже имеют смысл и требует, чтобы они обрабатывались особым образом. Эта функция добавлена и продуктовый код изменен. Мы написали новые тесты, учитывающие, что продуктовый код больше не возбуждает исключение. Но изменив код, так чтобы они

проходили, мы внезапно обнаруживаем, что старый тест (с трехбуквенным именем) проходить перестал. Он ожидает исключения, а его нет. Если исправить продуктовый код, так чтобы этот тест проходил, то перестанет проходить новый тест, проверяющий, что трехбуквенные имена обрабатываются специальным образом.

Такая ситуация «или-или», в которой может проходить только один из двух тестов, служит признаком наличия конфликтующих тестов. В таком случае сначала нужно убедиться, что конфликт действительно существует, а затем решить, какое требование оставить. После этого следует удалить (а не просто закомментировать) ставшее недействительным требование и тесты для него. (Серьезно, если я поймаю кого-то на закомментаривании вместо удаления, то напишу целую книгу под названием «Зачем Бог создал управление версиями».)

Конфликтующие тесты иногда позволяют выявить проблемы в требованиях заказчика, и тогда заказчик должен решить, какое требование правильно.

Переименование и рефакторинг тестов

Неудобочитаемый тест является скорее проблемой, чем решением. Он может затемнить существо кода, и вы не поймете, какую ошибку тест призван обнаруживать.

Встретив тест, имя которого расплывчато, сбивает с толку или может быть сделано более удобным для сопровождения, изменяйте код теста (но не его базовую функциональность). В листинге 8.3 приведен пример такого рефакторинга теста с целью повышения удобства сопровождения, в результате чего тест заодно стал и куда более понятным.

Устранение дублирующихся тестов

В случае командной разработки не редкость столкнуться с ситуацией, когда разные разработчики пишут тесты для проверки одной и той же функциональности. Я не требую непременного устранения дублирующихся тестов по двум причинам:

- чем больше (хороших) тестов, тем больше вероятность выловить ошибки;
- читая тесты, можно увидеть различные способы или семантику тестирования одной и той же функции.

Приведу несколько аргументов против сохранения дублирующихся тестов.

- Сопровождать несколько разных тестов одной и той же функциональности труднее.
- Одни тесты могут быть более качественными, чем другие, а анализировать на предмет правильности нужно все.
- В случае ошибки в одном месте упадет сразу несколько тестов (но это может и не считаться недостатком).
- Похожие тесты могут иметь разные имена или находиться в разных классах.
- Чем больше тестов, тем сложнее сопровождение.

А вот аргументы за сохранение.

- В тестах могут быть мелкие различия, поэтому можно считать, что они тестируют одну и ту же функциональность немного по-разному. Возможно, при этом сложится более верное представление о тестируемом объекте.
- Одни тесты могут быть выразительнее других, поэтому наличие нескольких тестов может повысить удобочитаемость.

Выше я сказал, что не требую непременного устранения дублирующихся тестов, но обычно все же удаляю их; доводы против как правило перевешивают.

8.1.2. Устранение логики из тестов

Шансы внести в тесты ошибки растут экспоненциально по мере добавления в них логики. Я видел, как многие тесты, которые по идее должны быть простыми, становились динамическими монстрами с изменяющейся логикой, генерацией случайных чисел, созданием потоков и записью в файлы, монстрами, которые уже сами начинали напоминать небольшие каркасы тестирования. Как это ни печально, из-за наличия атрибута `[Test]` автор не считал, что в этих методах могут быть ошибки, и не утруждал себя заботой об удобстве сопровождения. Такие тесты-монстры требовали на отладку и проверку больше времени, чем экономили.

Но даже самые чудовищные монстры поначалу были маленькими. Часто бывает, что какой-нибудь гуру в компании, глядя на тест, думает про себя: «А что если зациклить этот метод и подавать на вход случайные числа? Ведь так мы точно выловим куда больше ошибок!» Выловите, конечно, особенно в своих тестах. Ошибки в тестах сильнее всего раздражают программистов, потому что мы почти никогда не подозреваем, что причина отказа теста – в самом тесте. Я не хочу сказать, что такие тесты вовсе бесполезны. Я даже, наверное, сам их

пишу. Но я бы не стал называть их *автономными*. Я бы сказал, что это *интеграционные тесты*, потому что они слабо контролируют тестируемую сущность и доверять им на сто процентов я бы не стал (мы разовьем эту тему в разделе об отделении интеграционных тестов от автономных ниже).

Обнаружив в автономном тесте любую из следующих конструкций:

- предложения `switch`, `if`, `else`;
- циклы `foreach`, `for`, `while`,

знайте, что тест содержит логику, которой в нем быть не должно.

Тест, где имеется логика, обычно проверяет более одной вещи, что не рекомендуется, потому что такой тест менее понятен и более хрупок. Кроме того, из-за наличия логики увеличивается сложность, которая может скрывать ошибку.

В общем случае автономный тест должен представлять собой последовательность вызовов методов и утверждений, без каких-либо управляющих конструкций, в том число блоков `try-catch`. Все, что более сложно, может стать причиной следующих проблем:

- тест труднее читать и понимать;
- тест трудно воспроизвести (представьте, что внезапно упал тест с несколькими потоками или со случайными числами);
- выше вероятность того, что тест содержит ошибку или проверяет не то, что надо;
- тесту трудно придумать имя, потому что он делает несколько вещей.

Как правило, тесты-монстры приходят на смену более простым тестам, и это затрудняет поиск ошибок в продуктовом коде. Если уж приходится создавать тест-монстр, то хотя бы добавляйте его в комплект, не удаляя существующие тесты, и помещайте в проект, созданный специально для интеграционных, а не автономных тестов.

Еще один вид логики, которой следует избегать в автономных тестах, иллюстрируется на следующем примере:

```
[Test]
public void ProductionLogicProblem()
{
  string user =”USER”;
  string greeting=”GREETING”;
  string actual = MessageBuilder.Build(user,greeting);

  Assert.AreEqual(user + greeting,actual);
}
```

Проблема здесь в том, что ожидаемый результат определяется в утверждении динамически. Это, пусть простая, но все же логика. Очень может статься, что в тесте повторена логика продуктового кода – вместе со всеми содержащимися в ней ошибками (потому что код и тест писал один и тот же человек или потому что авторы кода и теста оба имели превратное представление о поведении кода).

Это означает, что если ошибка в продуктовом коде существует и повторена в тесте, то тест пройдет. В примере выше в ожидаемом значении в утверждении и в продуктовом коде пропущен пробел, поэтому тест проходит.

Было бы лучше «зашить» значения в код:

```
[Test]
public void ProductionLogicProblem()
{
  string actual = MessageBuilder.Build(“user”,”greeting”);
  Assert.AreEqual”user greeting”,actual);
}
```

Поскольку мы точно знаем, как должен выглядеть конечный результат, ничто не мешает зашить его в код. Теперь не имеет значения, как этот результат получен, но если при его вычислении допущена ошибка, то тест не пройдет. И из теста исчезла логика, которая могла бы содержать ошибку.

Логику можно обнаружить не только в тестах, но и во вспомогательных методах, рукописных подделках и служебных классах для тестирования. Помните – любая логика в этих местах серьезно осложняет чтение кода и повышает шансы на появление ошибки во вспомогательном методе, используемом в тестах.

Если вы считаете, что по какой-то причине комплект тестов должен содержать нетривиальную логику (хотя лично я сделал бы такие тесты интеграционными, а не автономными), то, по крайней мере, включите в тестовый проект парочку тестов, проверяющих логику служебных методов. Тогда вам впоследствии не придется кусать локти.

8.1.3. Тестирование только одного результата

Под результатом здесь понимается конечный результат единицы работы: возвращаемое значение, изменение состояния системы или обращение к стороннему объекту. Например, если в тесте высказываются утверждения о нескольких объектах, то, вероятно, тес-

тируется более одного результата. То же самое имеет место, когда проверяется, что объект возвращает правильное значение, и одновременно что состояние системы изменилось, так что объект стал вести себя иначе.

Тестирование нескольких результатов кажется безобидным, пока вы не задумаетесь о том, как назвать тест, или о том, что должно произойти, если первое утверждение не выполнено.

Выбор имени для теста – задача вроде бы простая, но если тестируется несколько результатов, то придумать хорошее имя, которое ясно указывало бы, что именно тестируется, почти невозможно. Получается слишком общее имя, которое заставляет читателя изучать код теста (подробнее об этом будет сказано в разделе об удобочитаемости). Если тестируется только один результат, то назвать тест легко.

Хуже то, что в большинстве каркасов автономного тестирования (включая NUnit) ложное утверждение возбуждает специальное исключение, которое перехватывает исполнитель тестов. Если каркас тестирования обнаружил такое исключение, считается, что тест не прошел. К несчастью, исключения, по самой своей природе, препятствуют дальнейшему выполнению кода. В той строке, где возникло исключение, происходит выход из метода. Пример приведен в листинге 8.4. Если первое утверждение ложно, то будет возбуждено исключение, а, значит, второе утверждение вообще никогда не проверяется, и мы не узнаем, изменилось ли поведение объекта, зависящее от его состояния. Оба утверждения могут и должны рассматриваться как разные требования, а потому могут и должны проверяться в разных тестах, следующих друг за другом.

Листинг 8.4. Тест с несколькими утверждениями

```
[Test]
public void IsValid_WhenValid_ReturnsTrueAndRemembersItLater()
{
  LogAnalyzer logan = MakeDefaultAnalyzer();

  Assert.IsTrue(logan.IsValid("abc"));
  Assert.IsTrue(logan.WasLastCallValid);
}
```

Считайте невыполнение утверждения симптомом болезни. Чем больше симптомов удается найти, тем легче диагностировать болезнь. Утверждения, следующие за первым отказом, не проверяются, поэтому вы не увидите дополнительных симптомов, которые могли бы дать ценные сведения для локализации и диагностики проблемы.

Тест, показанный в листинге 8.4, должен быть разбит на два теста с подходящими именами.

Можно взглянуть на это и по-другому: если первое утверждение ложно, то важен ли результат проверки следующего? Если да, то, вероятно, следует разбить тест на два.

Проверка нескольких результатов в одном автономном тесте лишь увеличивает сложность, не давая почти ничего взамен. Дополнительные результаты следует проверять в отдельных независимых тестах, чтобы было ясно, что именно не соответствует ожиданиям.

8.1.4. Разделение автономных и интеграционных тестов

В главе 7 мы говорили о безопасной зеленой зоне для тестов. Я возвращаюсь к этому вопросу, потому что считаю его очень важным. Если коллеги не верят, что ваши тесты будут работать быстро, давая всякий раз один и тот же результат, то они вообще не станут их прогонять. Переработав тесты в этом направлении, вы сможете сделать их заслуживающими больше доверия. Если ваши тесты находятся в безопасной зеленой зоне, то другие разработчики склонны доверять им в большей степени. Создать такую зеленую зону несложно – нужно лишь завести отдельный проект для автономных тестов и следить за тем, чтобы туда попадали только тесты, работающие исключительно в памяти, повторяемые и дающие всякий раз один и тот же результат.

8.1.5. Проводите анализ кода, уделяя внимание покрытию кода

Что означает стопроцентное покрытие кода? Да ничего, если не сопровождается анализом кода. Возможно, генеральный директор потребовал, чтобы каждый сотрудник «обеспечил как минимум 95-процентное покрытие кода», и все будут делать то, что приказано. А в тестах-то, быть может, и утверждений нет. Людям свойственно делать то, что требуется для достижения метрики, на которой основана оценка их труда.

Что означает стопроцентное покрытие кода наряду с тестами и анализом кода? А то, что весь мир в ваших руках. Если вы проводили анализ кода и анализ тестов и убедились, что тесты хороши и покрывают весь код, значит, вы растянули страховочную сетку, ко-

торая предохранит от глупых ошибок, и одновременно дали возможность всем членам команды поделиться знаниями и не переставать учиться.

Говоря «анализ кода», я не имею в виду равнодушное замечание по поводу чужого кода, написанное кем-то на другом конце света, которое автор увидит через три часа, когда критик уже ушел с работы.

Нет, говоря «анализ кода», я вижу двух людей, которые сидят и беседуют, глядя на один и тот же кусок кода и внося в него изменения. (Хотелось бы, чтобы они сидели бок о бок, но такие приложения для удаленного диалога, как Skype и TeamViewer тоже подойдут.) В следующей главе я еще поделюсь своими впечатлениями о том, как захватывающе выглядят такие встречи, а сейчас просто запомните, что, пренебрегая постоянным анализом кода и тестов вдвоем, вы теряете замечательную возможность учиться и повышать продуктивность. В таком случае сделайте все возможное, чтобы не лишать себя этого обязательного и непременного умения. Анализ кода – это техника создания удобочитаемого высококачественного годами работающего кода, которая позволит вам с уважением приветствовать свое отражение в зеркале по утрам.

Да не смотрите на меня так! Этот скептицизм только мешает вам превратить свою нынешнюю работу в предмет мечтаний.

Однако же вернемся к разговору о покрытии кода.

Для обеспечения хорошего покрытия нового кода воспользуйтесь каким-нибудь автоматизированным инструментом (например, dotCover компании JetBrains, OpenCover, NCover или Visual Studio Pro). На данный момент мне больше нравится программа NCrunch, которая в реальном времени дает представление о коде в терминах красный-зеленый, изменяющееся по мере кодирования. Она стоит денег, но зато и экономит деньги. Задача в том, чтобы найти достойный инструмент, освоить его в полной мере и выжать из него все до капли, дабы покрытие кода больше никогда не было низким.

Если код покрыт меньше чем на 20 %, значит, очень многих тестов не хватает, а ведь никогда не знаешь, что захочет сделать с твоим кодом разработчик, пришедший тебе на смену. Возможно, он попытается что-то оптимизировать или удалит какую-то важную строку. Если нет теста, который в этом случае откажет, то ошибка может остаться незамеченной.

Во время анализа кода и тестов можно также проводить ручные проверки, это отличный способ спонтанного тестирования тестов.

Попробуйте закомментировать какую-нибудь строку или изменить значение булевой переменной в продуктовом коде. Если все тесты по-прежнему проходят, значит, каких-то тестов не хватает или тесты проверяют не то, что нужно.

Добавив тест, который раньше отсутствовал, проверьте, то ли вы добавили, выполнив следующие действия:

1. Закомментируйте продуктовый код, который, как вам кажется, не покрыт.
2. Прогоните все тесты.
3. Если все тесты проходят, значит, какого-то теста не хватает или тестируется не то, что нужно. В противном случае должен существовать тест, который ожидает, что эта строка будет выполнена или что окажется истинно какое-то утверждение, вытекающее из выполнения этой строки. Тогда этот тест не должен пройти.
4. Обнаружив недостающий тест, добавьте его. Закомментированный код таким и оставьте, при этом новый тест не должен пройти – это докажет, что закомментированного кода действительно не хватает.
5. Раскомментируйте прежде закомментированный код.
6. Написанный тест теперь должен пройти. Вы нашли и добавили недостающий тест!
7. Если тест по-прежнему не проходит, значит, либо в тесте ошибка, либо вы тестируете не то, что нужно. Измените тест, так чтобы он прошел. Ваша задача добиться того, чтобы тест не только проходил, когда должен проходить, но и падал, когда должен падать. Чтобы убедиться в последнем, снова внесите ошибку в продуктовый код (закомментировав в нем строку) и проверьте, падает ли тест.

Чтобы повысить степень доверия к тесту, попробуйте заменить различные параметры и внутренние переменные тестируемого метода константами (например, присвойте булевой переменной значение `true` и посмотрите, что получится).

Но хитрость в том, чтобы вся эта возня с тестами не заняла слишком много времени, иначе она не окупится. Именно этому вопросу – удобству сопровождения – посвящен следующий раздел.

8.2. Написание удобных для сопровождения тестов

Удобство сопровождения – одна из основных проблем, стоящих перед большинством программистов, которые пишут автономные тесты. В конечном итоге сопровождать и понимать тесты становится все труднее, и после малейшего изменения в системе то один, то другой тест перестает проходить, даже если ошибок-то и нет. Со временем появляется разрыв между вашим представлением о том, что делает код, и тем, что он делает в действительности.

В этом разделе я поделюсь знаниями, которые достались мне нелегко – в процессе написания автономных тестов в составе различных команд. Сюда входит тестирование выполнения открытых контрактов, устранение дублирования и обеспечение изоляции тестов.

8.2.1. Тестирование закрытых и защищенных методов

Закрытыми или защищенными методы обычно делаются не просто так. Иногда причина в том, чтобы скрыть детали реализации и оставить возможность для последующего изменения реализации без изменения заявленной функциональности. А иногда дело в безопасности и защите интеллектуальной собственности (к примеру, путем запутывания).

Тестируя закрытый метод, мы тестируем выполнение внутреннего системного контракта, который вполне может измениться. Внутренние контракты динамичны и подвержены изменению в процессе рефакторинга системы. Если такой контракт изменится, то тест может отказать, потому что какая-то внутренняя операция стала выполняться по-другому, пусть даже заявленная функциональность системы осталась той же самой.

С точки зрения тестирования, нас должен интересовать только открытый контракт (заявленная функциональность). Тестирование закрытых методов может привести к отказу тестов, хотя заявленная функциональность сохранилась.

Взгляните на это под таким углом зрения: закрытые методы существуют не без причины. Где-то в системе существует открытый метод, который вызывает этот закрытый метод напрямую или опосредованно. Это означает, что любой закрытый метод является частью более

крупной единицы работы, которая начинается в каком-то открытом API и заканчивается одним из трех результатов: возвратом значения, изменением состояния или обращением к стороннему объекту (или всеми тремя сразу).

Но тогда, увидев закрытый метод, ищите открытую единицу работы, которая приведет к вызову этого метода. Если вы протестируете один лишь закрытый метод и убедитесь, что он работает, это еще не означает, что остальные части системы вызывают этот метод корректно или что они корректно обрабатывают его результаты. Возможно, система правильно работает на внутреннем уровне, но открытые API применяют всю эту скрытую красоту катастрофически неверно.

Если закрытый метод достоин тестирования, то, быть может, его вообще стоит сделать открытым, статическим или хотя бы внутренним и определить открытый контракт, которого должны придерживаться все пользователи этого метода. А в некоторых случаях дизайн станет чище, если поместить метод совсем в другой класс. Чуть ниже мы рассмотрим оба подхода.

Означает ли это, что в кодовой базе вообще не должно остаться закрытых методов? Нет. Применяя TDD, мы обычно пишем тесты для открытых методов, а позже эти открытые методы подвергаются рефакторингу с выделением более мелких закрытых методов. На всем протяжении этого процесса тесты открытых методов проходят.

Преобразование методов в открытые

Сделать метод открытым необязательно плохо. Возможно, это идет вразрез с принципами объектно-ориентированного проектирования, на которых вы воспитаны, но желание протестировать метод может означать, что у этого метода имеется известное поведение или контракт с вызывающей стороной. Делая метод открытым, мы официально признаем этот факт. Оставляя метод закрытым, мы сообщаем всем разработчикам, которые придут нам на смену, что реализацию этого метода можно изменять, не заботясь о неизвестном коде, который может его использовать, потому что он имеет смысл только в составе более крупного образования, которое и заключает контракт с вызывающей стороной.

Перенос методов в новые классы

Если метод содержит много самостоятельной логики или пользуется состоянием класса, которое только к этому методу и относится, то имеет смысл перенести метод в новый класс, который будет играть

в системе отдельную роль. Затем этот класс можно тестировать независимо. Майкл Фэзерс в книге «Эффективная работа с унаследованным кодом» приводит интересные примеры такой техники, а Роберт Мартин в «Чистом коде» рассказывает, как решить, действительно ли эта идея хороша.

Преобразование метода в статический

Если метод вообще не пользуется внутренним состоянием класса, то разумно сделать его статическим. Такие методы тестировать гораздо проще, однако это решение также означает, что у метода имеется открытый контракт, определяемый его именем.

Преобразование метода во внутренний

Если вы ни в коем случае не согласны раскрыть метод официально, то, быть может, стоит сделать его внутренним и задать атрибут `[InternalsVisibleTo("TestAssembly")]` для сборки с продуктовым кодом – тогда тесты все же смогут вызывать этот метод. Мне это не нравится, но иногда другого выбора не остается (то ли из соображений безопасности, то ли из-за отсутствия контроля над проектом программы, то ли еще по какой-то причине).

Сделав метод внутренним, вы отнюдь не сделаете тесты более удобными для сопровождения, потому что у кодировщика может сложиться впечатление, что такой метод разрешено беспрепятственно изменять. Делая же метод частью явного открытого контракта, вы можете быть уверены, что программист, которому придется его изменять, точно знает – метод реально используется и нарушать контракт нельзя.

В версиях Visual Studio до 2012 была возможность создать закрытый акцессор – генерируемый Visual Studio класс-обертку, который с помощью отражения вызывал закрытый метод. Не пользуйтесь этой возможностью. Получающийся в результате код трудно сопровождать и нелегко понять по прошествии некоторого времени. Вообще, лучше избегать любых инструментов, которые обещают сгенерировать за вас автономные тесты или еще что-нибудь, относящееся к тестированию, – если только нет никакого другого выбора.

Удаление метода – не лучшее решение, потому что он используется и в продуктовом коде. В противном случае вообще не было бы причины писать тесты.

Еще один способ сделать код более удобным для сопровождения – устранить дублирование тестов.

8.2.2. Устранение дублирования

Дублирование в автономных тестах может навредить разработчикам ничуть не меньше, чем дублирование в продуктовом коде (а то и больше). Принцип DRY относится к коду тестов в той же мере, что и к продуктовому. Дублирование означает, что придется вносить больше изменений, когда изменяется что-то в тестируемом коде. Простое изменение конструктора или семантики использования класса может оказать огромное влияние на тесты, в которых много дублирующегося кода.

Чтобы понять, почему это так, рассмотрим простой пример.

Листинг 8.5. Тестируемый класс и тест, в котором он используется

```
public class LogAnalyzer
{
  public bool IsValid(string fileName)
  {
     if (fileName.Length < 8)
     {
        return true;
     }
     return false;
  }
}

[TestFixture]
public class LogAnalyzerTestsMaintainable
{
  [Test]
  public void IsValid_LengthBiggerThan8_IsFalse()
  {
     LogAnalyzer logan = new LogAnalyzer();

     bool valid = logan.IsValid("123456789");

     Assert.IsFalse(valid);
  }
}
```

Тест в нижней части листинга 8.5 кажется совершенно нормальным, но это лишь до поры – пока не появится еще один тест того же класса. Теперь у нас есть два теста, показанных в следующем листинге.

Листинг 8.6. Два теста, содержащих дублирование

```
[Test]
public void IsValid_LengthBiggerThan8_IsFalse()
```

```
{
  LogAnalyzer logan = new LogAnalyzer();

  bool valid = logan.IsValid("123456789");

  Assert.IsFalse(valid);
}

[Test]
public void IsValid_LengthSmallerThan8_IsTrue()
{
  LogAnalyzer logan = new LogAnalyzer();

  bool valid = logan.IsValid("1234567");

  Assert.IsTrue(valid);
}
```

Что не так с этими тестами? Основная проблема в том, что если способ использования `LogAnalyzer` (его семантика) изменится, то тесты придется сопровождать по отдельности, т. е. работы будет больше. В следующем листинге приведен пример такого изменения.

Листинг 8.7. Семантика `LogAnalyzer` изменилась – теперь требуется инициализация

```
public class LogAnalyzer
{
  private bool initialized=false;

  public bool IsValid(string fileName)
  {
     if(!initialized)
     {
        throw new NotInitializedException(
          "До любой операции необходимо вызвать " +
          "метод analyzer.Initialize()!");
     }
     if (fileName.Length < 8)
     {
        return true;
     }
     return false;
  }

  public void Initialize()
  {
     // здесь должна быть логика инициализации
     ...
     initialized=true;
```

```
    }
}
```

Теперь оба теста в листинге 8.6 не пройдут, потому что в них не вызван метод `Initialize()`. Так как налицо дублирование кода (в обоих тестах создается экземпляр класса), то придется внести изменения в каждый тест.

Можно провести рефакторинг тестов и устранить дублирование, создавая экземпляр `LogAnalyzer` в методе `CreateDefaultAnalyzer()`, который будут вызывать оба теста. Можно также перенести создание и инициализацию в новый метод подготовки, включив его в тестовый класс.

Устранение дублирование с помощью вспомогательного метода

В листинге 8.8 показано, как сделать тесты более удобными для сопровождения, написав общий фабричный метод, который создает экземпляр `LogAnalyzer` по умолчанию. В предположении, что все тесты пользуются этим методом, можно поместить в него вызов `Initialize()`, тогда не придется изменять все тесты, добавляя в них обращение к `Initialize()`.

Листинг 8.8. Включение вызова `Initialize()` в фабричный метод

```
[Test]
public void IsValid_LengthBiggerThan8_IsFalse()
{
  LogAnalyzer logan = GetNewAnalyzer();

  bool valid = logan.IsValid("123456789");

  Assert.IsFalse(valid);
}

[Test]
public void IsValid_LengthSmallerThan8_IsTrue()
{
  LogAnalyzer logan = GetNewAnalyzer();

  bool valid = logan.IsValid("1234567");

  Assert.IsTrue(valid);
}

private LogAnalyzer GetNewAnalyzer()
{
```

```
  LogAnalyzer analyzer = new LogAnalyzer();
  analyzer.Initialize();
  return analyzer;
}
```

Фабричные методы – не единственный способ устранить дублирование в тестах, как станет ясно из следующего раздела.

Устранение дублирования с помощью атрибута [SetUp]

Инициализировать `LogAnalyzer` можно и в методе `Setup()`, как показано ниже.

Листинг 8.9. Использование метода подготовки для устранения дублирования

```
[SetUp]
public void Setup()
{
  logan=new LogAnalyzer();
  logan.Initialize();
}

private LogAnalyzer logan= null;

[Test]
public void IsValid_LengthBiggerThan8_IsFalse()
{
  bool valid = logan.IsValid("123456789");
  Assert.IsFalse(valid);
}

[Test]
public void IsValid_LengthSmallerThan8_IsTrue()
{
  bool valid = logan.IsValid("1234567");
  Assert.IsTrue(valid);
}
```

При таком подходе в тестах даже не нужна строка, где создается объект анализатора; до начала каждого теста создается новый экземпляр класса `LogAnalyzer`, после чего для него вызывается метод `Initialize()`. Однако остерегайтесь: использовать метод подготовки для устранения дублирования не всегда разумно, как я объясню в следующем разделе.

8.2.3. Применение методов подготовки без усложнения сопровождения

Метод `Setup()` использовать просто. Пожалуй, даже слишком просто – настолько, что разработчики применяют его и в тех случаях, на которые он не рассчитан. И расплачиваются за это тестами, которые неудобно читать и сопровождать.

Кроме того, у методов подготовки есть ограничения, которые можно обойти с помощью простых вспомогательных методов.

- Методы подготовки полезны только для инициализации.
- Методы подготовки – не всегда оптимальное средство для устранения дублирования. Дублирование не обязательно сводится к созданию и инициализации новых объектов. Иногда требуется устранить дублирование в утверждениях.
- Методы подготовки не принимают параметры и не возвращают значение.
- Методы подготовки нельзя использовать в качестве фабричных методов, возвращающих значение. Они вызываются до запуска теста, поэтому должны быть более общими. Иногда тесту требуется запросить что-то конкретное или вызвать общий код с параметром, уникальным для данного теста (к примеру, получить объект и присвоить его свойству определенное значение).
- Методы подготовки должны содержать лишь код, относящийся ко всем тестам в данном тестовом классе, иначе будет трудно понять, что метод делает.

Перечислив основные ограничения методов подготовки, посмотрим, как разработчики пытаются обойти их, стремясь во что бы то ни стало использовать именно методы подготовки, а не вспомогательные методы. Можно привести следующие примеры неправильного использования:

- инициализация в методах подготовки объектов, которые используются только в некоторых тестах из одного класса;
- написание длинного и малопонятного кода подготовки;
- настройка в методе подготовки поддельных объектов.

Рассмотрим эти примеры по очереди.

Инициализация объектов, которые используются только в некоторых тестах

Это смертный грех. Предавшись ему, вы серьезно усложните не только сопровождение, но даже чтение тестов, потому что метод под-

готовки очень быстро наполняется объектами, относящимися лишь к некоторым тестам. В листинге ниже показано, во что может превратиться тестовый класс, в котором объект `FileInfo` инициализируется в методе подготовки, но используется только в одном тесте.

Листинг 8.10. Плохо реализованный метод `Setup()`

```
[SetUp]
public void Setup()
{
  logan=new LogAnalyzer();
  logan.Initialize();

  fileInfo=new FileInfo("c:\\someFile.txt");
}

private FileInfo fileInfo = null;
private LogAnalyzer logan = null;

[Test]
public void IsValid_LengthBiggerThan8_IsFalse()
{
  bool valid = logan.IsValid("123456789");
  Assert.IsFalse(valid);
}

[Test]
public void IsValid_BadFileInfoInput_returnsFalse()
{
  bool valid = logan.IsValid(fileInfo);
  Assert.IsFalse(valid);
}

[Test]
public void IsValid_LengthSmallerThan8_IsTrue()
{
  bool valid = logan.IsValid("1234567");
  Assert.IsTrue(valid);
}

private LogAnalyzer GetNewAnalyzer()
{
  ...
}
```

Используется только в одном тесте

Почему этот метод подготовки плохо пригоден для сопровождения? Дело вот в чем – впервые видя тесты и пытаясь понять, как они работают, читатель ведет себя следующим образом:

1. Изучает метод подготовки и смотрит, что в нем инициализируется.
2. Делает предположение, что инициализированные переменные используются во всех тестах.
3. Позднее обнаруживает, что это предположение ложно, и снова, уже более внимательно, просматривает тесты, чтобы понять, где используются объекты, могущие стать источником проблем.
4. Углубляется в код теста без достаточных оснований, затрачивая больше времени и сил, чтобы понять, что делает код.

Всегда, когда пишете тесты, ставьте себя на место читателя. Представьте, что вы впервые видите код. Заботьтесь о читателе, не раздражайте его.

Длинный и малопонятный код подготовки

Поскольку метод подготовки – единственное место в тестовом классе, где можно провести инициализацию, разработчики стараются инициализировать там как можно больше объектов, из-за чего понятность кода неизбежно страдает. Возможное решение – завести небольшие вспомогательные методы, вызываемые из метода подготовки для инициализации отдельных объектов. Рефакторинг метода подготовки – вообще удачная мысль, потому что чем понятнее этот метод, тем понятнее и весь тестовый класс.

Однако существует тонкая грань, за которой чрезмерное увлечение рефакторингом вредит удобочитаемости. Где она проходит, вопрос субъективный. Но вы должны следить, чтобы код не становился непонятным. Я рекомендую во время рефакторинга интересоваться мнением партнера. Все мы подпадаем под чары собственного кода, поэтому вторая пара глаз, наблюдающая за рефакторингом, может привнести объективный взгляд. Анализ кода (теста) коллегами постфактум – тоже вещь полезная, но не настолько, как критика в процессе работы.

Настройка поддельных объектов в методе подготовки

Прошу вас – никогда не настраивайте подделки в методе подготовки. Это сильно затрудняет чтение и сопровождение тестов.

Я предпочитаю в каждом тесте создавать нужные заглушки и подставки, вызывая вспомогательные методы, тогда читатель будет точно

знать, что происходит, не перескакивая глазами от теста к методу подготовки и обратно, чтобы составить полную картину.

Отказ от методов подготовки

Я перестал использовать методы подготовки в своих тестах. Это пережиток времен, когда считалось нормальным писать никуда не годные, нечитаемые тесты. Те времена прошли. Тестовый код должен быть не менее красивым и чистым, чем продуктовый. Но если продуктовый код выглядит ужасно, не считайте это оправданием для написания нечитаемых тестов. Пользуйтесь фабричными и вспомогательными методами – и всем станет лучше.

Если все тесты устроены одинаково, то методы подготовки можно с успехом заменить параметризованными тестами ([TestCase] в NUnit, [Theory] в XUnit.net, [УвыЗаПятьЛетМыЭтоТакИНеСделали] в MSTest). Ладно, шутка не очень удачная, но в MSTest действительно так и не появилось простой поддержки для этой функции.

8.2.4. Принудительная изоляция тестов

Отсутствие изоляции – самая главная причина подозрительного отношения к тестам, с которой я сталкивался в процессе консультирования и работы над автономными тестами. Основная идея состоит в том, что тест должен работать в собственном мирке, полностью изолированном от знания о существовании других тестов, которые делают нечто похожее или совершенно другое.

Тест, который кричал караул

В одном проекте, где я работал, были автономные тесты, которые вели себя странно, и со временем всё «страньше и страньше». В один день тест падал, а в следующие два дня проходил нормально. Потом снова падал, по видимости совершенно случайно, а иногда проходил, даже если тестируемое поведение кода изменялось или удалялось. Дошло до того, что разработчики говорили между собой: «Да нормально все. Проходит иногда – и ладно».

Как выяснилось, из этого теста вызывался другой тест и, когда тот падал, падал и этот.

У нас ушло три дня, чтобы понять, в чем дело, – после того как мы месяц мирились с ситуацией. Когда наконец тест заработал правильно, оказалось, что в коде уйма ошибок, на которые мы не обращали внимания, списывая их на ложноположительные результаты сбоящего теста. История о мальчике, который кричал «волки», применима и к разработке программного обеспечения.

Если тесты плохо изолированы, они могут наступать друг другу на пятки, отравляя вам жизнь, заставляя жалеть о том, что вы связались с этими чертовыми автономными тестами, и клясться никогда больше этого не делать. Я такое наблюдал. Разработчики не ожидают встретить ошибку в тестах, поэтому, когда проблема кроется именно там, на ее поиск уходит очень много времени.

Существует несколько «запашков», которые могут навести на мысль о нарушенной изоляции тестов.

- *Ограничения на порядок тестов*. Ожидается, что тесты должны выполняться в определенном порядке или получать информацию из результатов прогона других тестов.
- *Скрытый вызов тест*. Один тест вызывает другой.
- *Повреждение разделяемого состояния*. Несколько тестов пользуются одной и той же переменной в памяти, не возвращая ее в исходное состояние.
- *Повреждение внешнего разделяемого состояния*. Интеграционные тесты пользуются общими ресурсами, но не возвращают их в исходное состояние.

Рассмотрим эти антипаттерны по очереди.

Антипаттерн: ограничения на порядок тестов

Эта проблема возникает, когда при написании тестов предполагается наличие определенного состояния в памяти, во внешнем ресурсе или в текущем тестовом классе – состояния, созданное в результате выполнения других тестов из того же класса, запущенных раньше текущего. Беда в том, что на большинстве платформ тестирования (в том числе NUnit, JUnit и MbUnit) порядок выполнения тестов не определен, поэтому то, что проходит сегодня, может отказать завтра.

В следующем листинге показан тест класса `LogAnalyzer`, ожидающий, что предыдущий тест уже вызывал метод `Initialize()`.

Листинг 8.11. Ограничения на порядок тестов: второй тест не пройдет, если будет выполнен первым

```
[TestFixture]
public class IsolationsAntiPatterns
{
  private LogAnalyzer logan;

  [Test]
  public void CreateAnalyzer_BadFileName_ReturnsFalse()
  {
```

```
        logan = new LogAnalyzer();
        logan.Initialize();

        bool valid = logan.IsValid("abc");

        Assert.That(valid, Is.False);
    }

    [Test]
    public void CreateAnalyzer_GoodFileName_ReturnsTrue()
    {
        bool valid = logan.IsValid("abcdefg");

        Assert.That(valid, Is.True);
    }
}
```

Отсутствие изоляции тестов может приводить к самым разным проблемам.

- Тест может внезапно перестать работать после перехода на новую версию каркаса тестирования, в которой тесты выполняются в другом порядке.
- Прогон подмножества тестов может давать иные результаты, чем прогон всех тестов или другого подмножества.
- Сопровождение тестов усложняется, потому что нужно помнить о том, как тесты взаимосвязаны и как каждый влияет на общее состояние.
- Тесты могут проходить или падать по причинам, не связанным с тестируемым кодом; например, из-за того, что перед данным тестом прошел или не прошел какой-то другой, оставив ресурсы в неопределенном состоянии.
- Удаление или изменение одних тестов может повлиять на результаты других.
- Трудно придумать для тестов подходящие имена, потому что они тестируют более одной функции.

Существуют две общих ошибки, ведущие к плохой изоляции тестов.

- *Тестирование последовательности*. Разработчик пишет тесты, которые должны выполняться в определенном порядке, чтобы проверить последовательность действий, то есть большой сценарий, состоящий из многих операций. Это может быть также полный интеграционный тест, в котором каждый отдельный тест представляет один шаг.

- *Пренебрежение очисткой.* Ленивый разработчик не возвращает состояние, которое могло измениться в ходе теста, к исходному виду, а другие разработчики – осознанно или неосознанно – пишут тесты, зависящие от этого дефекта.

Решать эти проблемы можно по-разному.

- *Тестирование последовательности.* Вместо того чтобы писать автономные тесты для проверки последовательности действий, подумайте об использовании какого-нибудь каркаса интеграционного тестирования, например FIT или FitNesse или продукта, предназначенного для контроля качества, например AutomatedQA или WinRunner.
- *Пренебрежение очисткой.* Если вы ленитесь очищать после тестирования базу данных, файловую систему или объекты в памяти, подумайте о смене профессии. Эта работа не для вас.

Антипаттерн: скрытый вызов теста

В этом случае тест содержит один или несколько прямых обращений к другим тестам в том же или ином тестовом классе, что приводит к взаимозависимости тестов. В листинге ниже приведен тест `CreateAnalyzer_GoodNameAndBadNameUsage`, который в конце вызывает другой тест, что создает зависимость между тестами и нарушает изоляцию обоих.

Листинг 8.12. Вызов одного теста из другого нарушает изоляцию и вводит зависимость

```
[TestFixture]
public class HiddenTestCall
{
  private LogAnalyzer logan;

  [Test]
  public void CreateAnalyzer_GoodNameAndBadNameUsage()
  {
     logan = new LogAnalyzer();
     logan.Initialize();

     bool valid = logan.IsValid("abc");

     Assert.That(valid, Is.False);

     CreateAnalyzer_GoodFileName_ReturnsTrue();
  }

  [Test]
```

```
    public void CreateAnalyzer_GoodFileName_ReturnsTrue() <-- ❶ Скрытый вызов теста
    {
        bool valid = logan.IsValid("abcdefg");

        Assert.That(valid, Is.True);
    }
}
```

Такого рода зависимость ❶ может приводить к нескольким проблемам.

- Прогон подмножества тестов может давать иные результаты, чем прогон всех тестов или другого подмножества.
- Сопровождение тестов усложняется, потому что нужно помнить о том, как тесты взаимосвязаны и как и когда они вызывают друг друга.
- Тесты могут проходить или падать по причинам, не связанным с тестируемым кодом. Например, если один тест упал, то другой может также упасть или вообще не будет вызван. Может также случиться, что другой тест оставил какие-то общие переменные в неопределенном состоянии.
- Изменение одних тестов может повлиять на результаты других.
- Трудно придумать хорошее имя для теста, который вызывает другие тесты.

Как возникает эта проблема?

- *Тестирование последовательности.* Разработчик пишет тесты, которые должны выполняться в определенном порядке, чтобы проверить последовательность действий, то есть большой сценарий, состоящий из многих операций. Это может быть также полный интеграционный тест, в котором каждый отдельный тест представляет один шаг.
- *Попытка устранить дублирование.* Разработчик пытается устранить дублирование в одних тестах путем вызова других (в которых уже имеется код, который не хочется повторять).
- *Пренебрежение разделением тестов.* Ленивый разработчик не нашел времени, чтобы создать отдельный тест и выполнить рефакторинг как положено, а вместо этого решил срезать угол и просто вызвать другой тест.

Решения:

- *Тестирование последовательности.* Вместо того чтобы писать автономные тесты для проверки последовательности дейст-

вий, подумайте об использовании какого-нибудь каркаса интеграционного тестирования, например FIT или FitNesse или продукта, предназначенного для контроля качества, например AutomatedQA или WinRunner.

- *Попытка устранить дублирование.* Никогда не пытайтесь устранить дублирование путем вызова одного теста из другого. Вы не даете вызываемому тесту шанса воспользоваться методами подготовки и очистки и по существу выполняете два теста в одном (поскольку вызывающий тест проверяет утверждение, содержащееся в вызываемом). Вместо этого вынесите код, который не хотите писать дважды, в отдельный метод и вызывайте его из обоих тестов.
- *Пренебрежение очисткой.* Если вы ленитесь разделять тесты, подумайте, сколько дополнительной работы вы на себя взваливаете. Попробуйте представить мир, в котором тест, который вы сейчас пишете, единственный в системе, так что зависеть ему не от кого.

Антипаттерн: повреждение разделяемого состояния

Этот антипаттерн проявляется двумя независимыми способами.

- Тест изменяет разделяемые ресурсы (в памяти или внешние, например базу данных, файловую систему и т. д.) и не откатывает сделанные изменения.
- Тест не устанавливает необходимое ему начальное состояние до начала работы, считая, что оно уже кем-то установлено.

Симптомы в обоих случаях одинаковы, мы обсудим их ниже. Проблема в том, что повторяемость поведения тестов зависит от некоторого состояния. Если тест не контролирует состояние, которое ожидает, или другие тесты по какой-то причине повреждают это состояние, то тест не сможет правильно работать, стабильно получая один и тот же результат.

Допустим, имеется класс `Person` с простой функциональностью: он хранит внутри себя список телефонов и умеет искать номер по начальным цифрам. В листинге ниже показаны два теста, которые не очищают и не инициализируют объект `Person`.

Листинг 8.13. Повреждение разделяемого состояния тестом

```
[TestFixture]
public class SharedStateCorruption
```

```
{
  Person person = new Person();   <-- Определяем разделяемое состояние Person

  [Test]
  public void CreateAnalyzer_GoodFileName_ReturnsTrue()
  {
     person.AddNumber("055-4556684(34)");
     string found = person.FindPhoneStartingWith("055");   <-- ❶ Изменяем разделяемое состояние

     Assert.AreEqual("055-4556684(34)", found);
  }

  [Test]
  public void FindPhoneStartingWith_NoNumbers_ReturnsNull()
  {
     string found = person.FindPhoneStartingWith("0");   <-- Читаем разделяемое состояние

     Assert.IsNull(found);
  }
}
```

Здесь второй тест (ожидающий, что будет возвращено значение `null`) не пройдет, потому что предыдущий тест добавил телефон ❶ в экземпляр `Person`.

Такого рода проблема проявляется рядом симптомов.

- Прогон подмножества тестов может давать иные результаты, чем прогон всех тестов или другого подмножества.
- Сопровождение тестов усложняется, потому что один тест может изменить состояние и тем самым нарушить работу других тестов, не сознавая этого.
- Тесты могут проходить или падать по причинам, не связанным с тестируемым кодом; например, из-за того, что перед данным тестом какой-то другой не прошел и оставил неопределенное разделяемое состояние или прошел, но не подчистил за собой.
- Изменение одних тестов может оказать влияние – на первый взгляд, случайное – на исход других

Как возникает эта проблема?

- *Пренебрежение установкой состояния в начале каждого теста*. Разработчик не устанавливает состояние, необходимое для работы теста, или предполагает, что оно уже установлено правильно.
- *Использование разделяемого состояния*. Разработчик использует общее состояние в памяти или во внешнем ресурсе в нескольких тестах, не принимая мер предосторожности.

- *Использование статических объектов в тестах*. Разработчик устанавливает статическое состояние, которое используется в других тестах.

Решения:

- *Пренебрежение установкой состояния в начале каждого теста*. Это обязательно нужно делать при написании автономных тестов. Либо пользуйтесь методом подготовки, либо вызывайте в начале каждого теста вспомогательный метод, который гарантированно устанавливает ожидаемое состояние.
- *Использование разделяемого состояния*. Часто можно обойтись вообще без разделяемого состояния. Самое безопасное решение – создавать новый объект в каждом тесте.
- *Использование статических объектов в тестах*. Если тесты управляют статическим состоянием, то нужна особая осторожность. Не забывайте сбрасывать такое состояние в методе подготовке или очистки. Иногда более эффективно явно вызывать из теста какой-нибудь вспомогательный метод, который сбрасывает статическое состояние. Если тестируются объекты-одиночки (singleton), то имеет смысл включить в них открытые или внутренние установщики свойств, чтобы тесты могли сбросить объект в исходное состояние.

Антипаттерн: повреждение внешнего разделяемого состояния

Этот антипаттерн похож на повреждение разделяемого состояния в памяти, но встречается при интеграционном тестировании.

- Тест изменяет внешние разделяемые ресурсы (например, базу данных или файловую систему) и не откатывает сделанные изменения.
- Тест не устанавливает необходимое ему начальное состояние до начала работы, считая, что оно уже кем-то установлено.

Поговорив об изоляции тестов, обратимся к вопросу о том, как надо управлять утверждениями, чтобы получать полную информацию в случае отказа теста.

8.2.5. Предотвращение нескольких утверждений о разных функциях

Чтобы понять, в чем состоит проблема нескольких функций, рассмотрим пример.

Листинг 8.14. Тест с несколькими утверждениями

```
[Test]
public void CheckVariousSumResultsOgnoringHigherThan1001()
{
  Assert.AreEqual(3, Sum(1001,1,2));
  Assert.AreEqual(3, Sum(1,1001,2));
  Assert.AreEqual(3, Sum(1,2,1001);
}
```

В этом методе находится несколько тестов. Можно сказать, что тестируются три разные функции.

Автор теста пытался сэкономить время и включил три теста в виде трех простых утверждений. В чем тут проблема? В случае ложности утверждения возбуждается исключение (в NUnit возбуждается специальное исключение `AssertException`, которое исполнитель тестов перехватывает и интерпретирует как знак того, что текущий тестовый метод не прошел). Раз утверждение возбудило исключение, то все последующие строки метода не выполняются. Таким образом, если ложным оказалось первое утверждение в листинге 8.14, то два остальных вообще не выполнялись. Ну и что? Может, нас и не интересуют остальные утверждения, если хотя бы одно ложно? Может быть. А может быть и так, что в каждом утверждении тестируется независимая функция приложения, и нам важно знать результаты всех тестов, даже если какой-то один не прошел.

Существует несколько способов достичь поставленной цели:

- создать свой тест для каждого утверждения;
- воспользоваться параметризованными тестами;
- обернуть вызов каждого утверждения блоком `try-catch`.

Почему так существенно, что некоторые утверждения не проверялись?

Если ложным оказалось только одно утверждение, то мы никогда не узнаем, истинны другие утверждения в том же тестовом методе или нет. Возможно, вы *думаете*, что знаете, но до проверки утверждения это не более чем предположение. Видя только часть картины, человек склонен выносить о состоянии системы суждения, которые могут оказаться неверными. Чем больше у нас информации обо всех утверждениях – истинных или ложных, – тем лучше мы вооружены для ответа на вопрос, в каком месте системы может находиться ошибка, а в каком не может.

Это относится только к утверждениям о нескольких функциях. Если же вы проверяете, что некий человек зовется *X*, имеет от роду *Y* лет и т. д., то сказанное выше несущественно, потому что коль скоро одно утверж-

дение оказалось ложно, остальные нас уже не интересуют. Однако заинтересовали бы, если бы ожидаемое действие имело несколько конечных результатов. Например, возвращало бы число `3` *и* изменяло бы состояние системы. То и другое – функции, которые должны работать независимо от других функций.

Мне доводилось охотиться за призраками ошибок, которые не желали проявляться, потому что ложным оказывалось только одно из нескольких утверждений. Если бы я дал себе труд проверить истинность остальных утверждений, то понял бы, что ошибка совсем в другом месте. Иногда программист находит «якобы ошибку», но, исправив ее, обнаруживает, что утверждение, которое раньше было ложным, стало истинным, зато перестали выполняться (или продолжают не выполняться) другие утверждения в том же тесте. Бывает, что мы не видим проблему во всей полноте, а исправление частичных проявлений только вносит в систему новые ошибки, которые станут видны, лишь если известны результаты проверки всех утверждений.

Поэтому так важно, чтобы при тестировании нескольких функций проверялись все утверждения, даже если какие-то из них оказываются ложными. В большинстве случаев для этого лучше включать в каждый тест только одно утверждение.

Использование параметризованных тестов

И xUnit.net, и NUnit поддерживают так называемые параметризованные тесты с помощью атрибута `[TestCase]`. В листинге ниже показано, как можно использовать несколько атрибутов `[TestCase]`, чтобы прогнать один и тот же тест с разными параметрами, ограничившись всего одним методом. Отметим, что в NUnit атрибут `[TestCase]` заменяет атрибут `[Test]`.

Листинг 8.15. Тот же тестовый класс с параметризованными тестами

```
[TestCase(1001,1,2,3)]
[TestCase (1,1001,2,3)]
[TestCase (1,2,1001,3)]
public void Sum_HigherThan1000_Ignored(int x, int y, int z, int expected)
{
  Assert.AreEqual(expected, Sum(x, y, z));
}
```

Параметризованные тестовые методы в NUnit и xUnit.net отличаются от обычных тем, что могут принимать параметры. В NUnit следует также снабдить тестовый метод по меньшей мере одним атрибутом `[TestCase]` вместо `[Test]`. Этот атрибут принимает произвольное число параметров, которые во время выполнения сопоставляются с параметрами, указанными в сигнатуре тестового метода.

В листинге 8.15 тест ожидает четыре аргумента. В утверждении первые три аргумента выступают в роли параметров, а последний – в роли ожидаемого значения. Таким образом, появляется возможность декларативно описать тест с различными исходными данными.

Но самое главное – если один из тестов, заданных с помощью атрибутов `[TestCase]`, не проходит, то остальные все равно выполняются, поэтому мы видим полную картину «прошел – не прошел».

Обертывание блоком try-catch

Некоторые считают допустимым погрузить каждое утверждение в свой блок `try-catch`, перехватить исключение, напечатать информацию о нем на консоли, а затем перейти к следующему предложению. Таким образом обходится проблема исключений в тестах. Я полагаю, что параметризованные тесты – более правильный способ достижения той же цели.

Итак, вы теперь знаете, как избежать в одном тесте нескольких утверждений о разных функциях. Давайте поговорим о тестировании разных сторон одного объекта.

8.2.6. Сравнение объектов

Вот еще один пример теста с несколькими утверждениями, но на этот раз мы не пытаемся втиснуть несколько логических тестов в один, а хотим проверить различные аспекты одного и того же состояния. Если какая-то проверка закончится неудачно, мы хотим узнать об этом.

Листинг 8.16. Проверка нескольких сторон одного объекта в одном тесте

```
[Test]
public void Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFields()
{
  LogAnalyzer log = new LogAnalyzer();
  AnalyzedOutput output = log.Analyze("10:05\tOpen\tRoy");

  Assert.AreEqual(1,output.LineCount);
  Assert.AreEqual("10:05",output.GetLine(1)[0]);
  Assert.AreEqual("Open",output.GetLine(1)[1]);
  Assert.AreEqual("Roy",output.GetLine(1)[2]);
}
```

Здесь проверяется, что `LogAnalyzer` правильно разбирает входную строку, для чего каждое поле результата сравнивается отдельно.

Все сравнения должны завершиться успешно, иначе будет считаться, что тест не прошел.

Повышение удобства сопровождения тестов

Ниже показано, как можно переработать тест из листинга 8.16, чтобы его было проще читать и сопровождать.

Листинг 8.17. Сравнение объектов вместо высказывания нескольких утверждений

```
[Test]
public void Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFields2()
{
  LogAnalyzer log = new LogAnalyzer();
  AnalyzedOutput expected = new AnalyzedOutput();
  expected.AddLine("10:05", "Open", "Roy");

  AnalyzedOutput output = log.Analyze("10:05\tOpen\tRoy");

  Assert.AreEqual(expected,output);
}
```

Инициализируем ожидаемый объект

Сравниваем фактический объект с ожидаемым

Вместо нескольких утверждений можно создать объект, задать ожидаемые значения его свойств, а затем сравнить результат с ожидаемым объектом в одном утверждении. Такой код гораздо проще понять и в том числе установить, что имеется всего один логический блок, а не несколько разрозненных тестов.

Внимание! Отметим, что такое сравнение возможно, лишь если в классе сравниваемых объектов переопределен метод `Equals()`. Некоторые считают это условие неприемлемым. Время от времени я такую технику применяю, но могу пойти и первым путем. Решайте сами. Поскольку я использую ReSharper, то мне достаточно нажать клавиши **Alt+Insert**, выбрать из меню команду **Generate Equality Members** и – вуаля! – к моим услугам весь код, необходимый для сравнения на равенство. Удобно.

Переопределение метода ToString()

Другой подход состоит в том, чтобы переопределить метод `ToString()` в классе сравниваемых объектов, так чтобы в случае отказа теста получать более понятные сообщения об ошибках. Вот, например, что мы увидим, если тест в листинге 8.17 не пройдет:

```
TestCase 'AOUT.CH8.LogAn.Tests.MultipleAsserts
.Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFields2'
failed:
```

```
Expected: <AOUT.CH789.LogAn.AnalyzedOutput>
But was: <AOUT.CH789.LogAn.AnalyzedOutput>
C:\GlobalShare\InSync\Book\Code\ARtOfUniTesting
\LogAn.Tests\MultipleAsserts.cs(41,0):
at AOUT.CH8.LogAn.Tests.MultipleAsserts
.Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFields2()
```

Не очень полезно, правда?

Реализовав метод `ToString()` в классах `AnalyzedOutput` и `LineInfo` (который является частью объектной модели), мы можем получить от тестов более понятную выходную информацию. В листинге ниже показаны реализации `ToString()` в обоих тестируемых классах и результаты, напечатанные тестом.

Листинг 8.18. Реализация метода `ToString()` в классах сравниваемых объектов с целью получения более внятной выходной информации

```
// Переопределение ToString в классе AnalyzedOutput
public override string ToString()
{
  StringBuilder sb = new StringBuilder();
  foreach (LineInfo line in lines)
  {
     sb.Append(line.ToString());
  }
  return sb.ToString();
}

// Переопределение ToString в классе LineInfo
public override string ToString()
{
  StringBuilder sb = new StringBuilder();
  for (int i = 0; i < this.fields.Length; i++)
  {
     sb.Append(this[i]);
     sb.Append(",");
  }
  return sb.ToString();
}

/// ВЫВОД ТЕСТА
------ Test started: Assembly: er.dll ------
TestCase 'AOUT.CH8.LogAn.Tests.MultipleAsserts
.Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFields2'
failed:
Expected: <10:05,Open,Roy,>
But was: <>
C:\GlobalShare\InSync\Book\Code\ARtOfUniTesting
\LogAn.Tests\MultipleAsserts.cs(41,0):
```

```
at AOUT.CH8.LogAn.Tests.MultipleAsserts
.Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFields2()
```

Теперь вывод стал гораздо информативнее, и мы видим, что объекты не имеют между собой ничего общего. Благодаря внятной выходной информации становится легче разобраться, почему тест не прошел, что, в свою очередь, упрощает сопровождение.

Затруднения при сопровождении могут возникнуть и потому, что тесты слишком хрупкие из-за избыточного специфицирования.

8.2.7. Предотвращение избыточного специфицирования

Избыточно специфицированным называется тест, который содержит предположения о том, как должно быть реализовано внутреннее поведение тестируемой единицы работы (продуктовый код), а не просто проверяет правильность конечного результата.

Вот как чаще всего проявляется избыточное специфицирование:

- в тесте высказывается утверждение о чисто внутреннем состоянии тестируемого объекта;
- в тесте используется несколько подставок;
- в тесте заглушки используются как подставки;
- в тесте предполагается определенный порядок следования или производится точное сравнение строк, хотя это и не нужно.

Совет. Этот вопрос обсуждается также в книге Джерарда Мезароша «Шаблоны тестирования xUnit. Рефакторинг кода тестов».

Рассмотрим примеры избыточно специфицированных тестов.

Проверка чисто внутреннего поведения

В следующем листинге приведен тест метода `Initialize()` из класса `LogAnalyzer`, в котором проверяется только внутреннее состояние без какой-либо внешней функциональности.

Листинг 8.19. Избыточно специфицированный тест, в котором проверяется чисто внутреннее поведение

```
[Test]
public void Initialize_WhenCalled_SetsDefaultDelimiterIsTabDelimiter()
{
  LogAnalyzer log = new LogAnalyzer();

  Assert.AreEqual(null,log.GetInternalDefaultDelimiter());
```

```
    log.Initialize();
    Assert.AreEqual('\t', log.GetInternalDefaultDelimiter());
}
```

Этот тест избыточно специфицирован, потому что в нем проверяется лишь внутреннее состояние объекта `LogAnalyzer`. Раз это состояние внутреннее, то в будущем оно может измениться.

Автономные тесты должны проверять выполнение открытого контракта и открытую функциональность объекта. В данном случае тестируемый код не является частью открытого контракта или интерфейса.

Использование заглушек как подставок

Использование подставок вместо заглушек – типичный случай избыточного специфицирования. Рассмотрим пример. Пусть имеется репозиторий данных, который должен возвращать поддельные данные тестам. Это заглушка. А что, если мы выскажем относительно нее утверждение, проверяющее, что она действительно вызывалась? Соответствующий код показан в листинге ниже.

Листинг 8.20. Избыточно специфицированный тест, в котором заглушка используется как подставка

```
[Test]
public void IsLoginOK_UserDoesNotExist_ReturnsFalse()
{
  IDataRepository fakeData = A.Fake<IDataRepository>();
  A.CallTo(()=> fakeData.GetUserByName(A<string>.Ignored))
     .Returns(null);
  LoginManager login = new LoginManager(fakeData);

  bool result = login.IsLoginOK("UserNameThatDoesNotExist","anypassword");

  Assert.IsFalse(result);
  A.CallTo(()=>fakeData.GetUserByName("UserNameThatDoesNotExist"))
     .MustHaveHappened(); <┐ Не следует проверять, что заглушка
}                          │ вызывалась. Это избыточное
                           │ специфицирование
```

Этот тест избыточно специфицирован, потому что в нем проверяется взаимодействие между заглушкой репозитория и `LoginManager` (с помощью FakeItEasy). Тест должен дать тестируемому методу возможность работать по его внутреннему алгоритму, а проверить только конечный результат. Тогда тест будет менее хрупким. А в таком виде, как мы сейчас видим, тест перестанет работать, если мы захотим добавить какой-нибудь внутренний вызов или оптимизировать путем из-

менения параметров вызова. Коль скоро метод делает то, что от него требуется, тесту должно быть все равно, вызывал он что-то внутри себя или нет.

Будь тест написан правильно, последняя строка отсутствовала бы.

Еще один вид избыточного специфицирования тестов – чрезмерное количество допущений.

Предположение об определенном порядке следования или точное сравнение, когда это не нужно

Часто разработчики сравнивают возвращенное строковое значение или свойство с фиксированной строкой, хотя требуется только совпадение части строки. Спросите себя: «Можно ли здесь использовать `string.Contains()` вместо `string.Equals()`?».

То же самое относится к коллекциям и спискам. Гораздо лучше проверять, что коллекция содержит ожидаемый элемент, чем утверждать, что элемент находится в определенной позиции (если, конечно, именно это и ожидается).

Внеся эти мелкие поправки, вы можете быть уверены, что коль скоро строка или коллекция содержат ожидаемую часть, тест пройдет. Даже если точный вид строки или порядок элементов в коллекции изменяется, вам не придется вносить буквалистские изменения в тест.

Теперь перейдем к третьему и последнему столпу хороших автономных тестов: удобочитаемости.

8.3. Написание удобочитаемых тестов

Неудобочитаемые тесты почти бесполезны. Удобочитаемость – связующая нить между автором и беднягой, которому придется читать тест через несколько месяцев. Тесты – это истории, которые вы рассказываете следующему поколению программистов, работающих над проектом. Они говорят разработчику, из чего состоит приложение и с чего оно начиналось.

В этом разделе мы поговорим о том, как сделать так, чтобы программисты, которые придут вслед за вами, смогли поддерживать написанный вами продуктовый код и тесты, понимая, что они делают и куда смотреть.

У удобочитаемости есть несколько граней:

- именование автономных тестов;
- именование переменных;
- хорошие сообщения в утверждениях;
- отделение утверждений от действий.

Рассмотрим все по порядку.

8.3.1. Именование автономных тестов

Стандарты именования важны, поскольку предлагают правила и шаблоны, подсказывающие, что вы должны рассказать о тесте. Имя теста состоит из трех частей.

- *Имя тестируемого метода*. Это необходимо, чтобы можно было легко понять, где искать тестируемую логику. Делая эту часть первой, мы упрощаем навигацию и применение технологии Intellisense (если IDE ее поддерживает) в тестовом классе.
- *Сценарий, в рамках которого тестируется метод*. Это часть «с» имени: «Когда метод X вызывается *со значением* `null`, он должен делать Y».
- *Поведение, ожидаемое в данном сценарии*. В этой части объясняется, что должен возвращать метод или как он должен себя вести в контексте данного сценария: «Когда метод X вызывается со значением `null`, *он должен делать Y*».

Если убрать хотя бы одну часть имени, то читателю будет непонятно, что происходит, и он начнет изучать код теста. Ваша главная цель – освободить своего преемника от необходимости читать код теста, чтобы понять, что тестируется.

Обычно три части имени разделяют знаками подчеркивания, например: `MethodUnderTest_Scenario_Behavior()`. Ниже приведен тест, в имени которого описанное соглашение соблюдается.

Листинг 8.21. Имя теста, состоящее из трех частей

```
[Test]
public void
AnalyzeFile_FileWith3LinesAndFileProvider_ReadsFileUsingProvider()
{
  //...
}
```

Этот метод тестирует метод `AnalyzeFile`, подавая ему на вход файл из трех строчек и поставщик чтения файла. Ожидается, что метод воспользуется поставщиком для чтения файла.

Если все разработчики будут придерживаться этого соглашения об именовании, то новым членам команды будет проще войти в курс дела и разобраться в тестах.

8.3.2. Именование переменных

Принципы именования переменных в автономных тестах важны не менее, а то и более, чем в продуктовом коде. Помимо основной функции – тестирования, тесты являются еще и одним из видов документации API. Давая переменным хорошие имена, вы помогаете читателям тестов максимально быстро понять, что вы пытаетесь *доказать* (в отличие от того, что вы пытаетесь *осуществить* в продуктовом коде).

В следующем листинге приведен пример плохо названного и плохо написанного теста. Такой тест я называю «нечитаемым», потому что не могу понять, какова его цель.

Листинг 8.22. Нечитаемое имя теста

```
[Test]
public void BadlyNamedTest()
{
  LogAnalyzer log = new LogAnalyzer();

  int result= log.GetLineCount("abc.txt");

  Assert.AreEqual(-100,result);
}
```

В данном случае в утверждении встречается какое-то магическое число (-100), представляющее значение, о котором разработчик должен знать. Не имея осмысленного имени для этого числа, мы можем только предполагать, что оно означает. Имя теста должно быть тут помочь, но его – как бы помягче сказать? – следует немного подправить.

Так что такое -100? Признак какой-то ошибки? Нормальное возвращаемое значение? Тут у нас есть выбор:

- изменить дизайн API и возбуждать исключение вместо возврата `-100` (в предположении, что -100 свидетельствует о наличии ошибки);
- сравнивать результат с именованной константой или удачно названной переменной, как показано ниже.

Листинг 8.23. Более удобочитаемый вариант теста

```
[Test]
public void BadlyNamedTest()
{
  LogAnalyzer log = new LogAnalyzer();

  int result= log.GetLineCount("abc.txt");

  const int COULD_NOT_READ_FILE = -100;

  Assert.AreEqual(COULD_NOT_READ_FILE,result);
}
```

Код в листинге 8.23 гораздо лучше, так как легко понять смысл возвращаемого значения.

Последняя часть теста – обычно утверждение, и мы должны выжать максимум пользы из содержащегося в нем сообщения. Если утверждение ложно, то первое, что увидит пользователь, – это напечатанное сообщение.

8.3.3. Утверждения со смыслом

Не пишите свои сообщения в утверждениях. Прошу вас. Этот раздел предназначен тем, кто считает абсолютно необходимым нестандартное сообщение, потому что тест без него ну никак не обойдется, а сделать тест более понятным по-другому не получается. Придумать хорошее сообщение в утверждении ничуть не проще, чем в исключении. Очень легко пойти по неверному пути, даже не сознавая этого, а для людей, которые будут сообщение читать, разница огромна (в том числе в плане временных затрат).

Запомните несколько советов о том, каким должно быть сообщение в утверждении.

- Не повторяйте то, что каркас тестирования и так выводит на консоль.
- Не повторяйте то, что ясно из имени теста.
- Если вам нечего сказать, не говорите ничего.
- Напишите, что должно было произойти или что не произошло, и по возможности упомяните, когда это должно было произойти.

В листинге ниже приведен пример плохого сообщения в утверждении и показано, что печатается в результате.

Листинг 8.24. Плохое сообщение в утверждении, повторяющее то, что каркас тестирования и так печатает

```
[Test]
public void BadAssertMessage()
{
  LogAnalyzer log = new LogAnalyzer();

  int result= log.GetLineCount("abc.txt");

  const int COULD_NOT_READ_FILE = -100;

  Assert.AreEqual(COULD_NOT_READ_FILE,result,
       "result was {0} instead of {1}",
       result,COULD_NOT_READ_FILE);
}

// При выполнении этого теста печатается:
TestCase 'AOUT.CH8.LogAn.Tests.Readable.BadAssertMessage'
failed:
 result was -1 instead of -100
 Expected: -100
 But was: -1
C:\GlobalShare\InSync\Book\Code
\ARtOfUniTesting\LogAn.Tests\Readable.cs(23,0)
: at AOUT.CH8.LogAn.Tests.Readable.BadAssertMessage()
```

Как видите, сообщение повторяется. Включенное в утверждение сообщение не добавило ничего, кроме лишних слов, которые приходится читать. Было бы куда лучше ничего не выводить, а подумать о хорошем имени для теста. Более внятное сообщение могло бы выглядеть так:

```
Вызов GetLineCount() для несуществующего файла должен был вернуть
COULD_NOT_READ_FILE.
```

Итак, все ваши сообщения теперь понятны. Самое время убедиться, что утверждение и вызов метода находятся в разных строках.

8.3.4. Отделение утверждений от действий

Этот раздел будет коротким, но от того не менее важным. Для повышения удобочитаемости не помещайте утверждение и вызов метода в одно предложение.

В листинге 8.25 приведен хороший, а в листинге 8.26 – плохой пример.

Листинг 8.25. Отделение утверждения от его предмета, тест легко читается

```
[Test]
public void BadAssertMessage()
{
  // какой-то код
  int result= log.GetLineCount("abc.txt");
  Assert.AreEqual(COULD_NOT_READ_FILE,result);
}
```

Листинг 8.26. Утверждение и его предмет не разделены, тест читается с трудом

```
[Test]
public void BadAssertMessage()
{
  // какой-то код
  Assert.AreEqual(COULD_NOT_READ_FILE,log.GetLineCount("abc.txt"));
}
```

Видите разницу? Тест в листинге 8.26 понять гораздо труднее, потому что вызов метода `GetLineCount()` находится внутри утверждения.

8.3.5. Подготовка и очистка

Методы подготовки и очистки в автономных тестах можно использовать до такой степени неправильно, что понять их будет невозможно. Обычно метод подготовки подвержен этому злу больше, нежели метод очистки.

Рассмотрим один пример злоупотребления. Если заглушки и подставки настраиваются в методе подготовки, значит, в самом тесте они не настраиваются. Это, в свою очередь, означает, что читатель теста может и не понять, что подставки вообще используются и чего тест от них ожидает.

Тест станет гораздо понятнее, если инициализировать подставные объекты в нем самом – там, где описываются ожидания. Если удобочитаемость для вас не пустой звук, то можете вынести создание подставок в отдельный вспомогательный метод, вызываемый из теста. Тогда читатель теста будет точно знать, что настраивается, и ему не придется заглядывать в несколько мест.

Совет. Я несколько раз писал полные тестовые классы вообще без метода подготовки – только вспомогательные методы, вызываемые из каждого теста. Все ради удобства сопровождения. Эти классы до сих пор удобочитаемы и сопровождаются.

8.4. Резюме

Немногие разработчики с первой попытки начинают писать автономные тесты, заслуживающие доверия. Чтобы сделать все правильно, нужны дисциплина и изобретательность. Свойство «заслуживает доверия» поначалу кажется трудноуловимым, но, поймав его, вы сразу ощутите разницу.

Есть несколько способов создавать достойные доверия тесты, в том числе оставлять в живых и актуализировать хорошие тесты, удаляя либо подвергая рефакторингу плохие. В этой главе мы обсудили ряд таких способов. А также проблемы, возникающие внутри тестов: наличие логики, тестирование сразу нескольких функций, простота запуска и т. д. Чтобы собрать все воедино, требуется немалое искусство.

Если вы забудете все сказанное в этой главе, кроме одной-единственной мысли, то пусть это будет мысль о том, что тесты развиваются и изменяются вместе с тестируемой системой.

Тема разработки удобных для сопровождения тестов в последние годы привлекала немало внимания, но, как я уже сказал, она недостаточно раскрыта в литературе по автономному тестированию и TDD – и не без причины. Я полагаю, что это будет следующий шаг в изучении эволюции приемов автономного тестирования. Первый шаг – приобретение начальных знаний (что такое автономный тест и как его писать), он описывается во многих местах. Второй шаг – шлифовка техники, совершенствование всех граней написанного вами кода и исследование других факторов, в том числе удобства чтения и сопровождения. Этот шаг очень важен, и именно ему посвящена эта глава (да и большая часть книги).

По существу, все просто: удобочитаемость идет рука об руку с удобством сопровождения и доверием к тестам. Человек, который может прочесть тесты, сможет также понять и сопровождать их, и будет доверять их результатам. Дойдя до этой стадии, вы будете готовы взглянуть в лицо изменениям, и ничто не помешает вам модифицировать код, если он в этом нуждается, потому что вы сразу узнаете о поломке.

В следующих главах мы взглянем на автономные тесты шире – как на часть более крупной системы: поговорим об их месте в организации, а также о взаимосвязи с существующими системами и унаследованным кодом. Вы узнаете, что делает код тестопригодным, как проектировать с учетом тестопригодности и как с помощью рефакторинга привести имеющийся код в тестопригодное состояние.

Часть IV.

ПРОЕКТИРОВАНИЕ И ПРОЦЕСС

В последних главах мы рассмотрим, с какими проблемами приходится сталкиваться при внедрении автономного тестирования в существующей организации или попытке протестировать уже имеющийся код.

В главе 9 речь пойдет об очень непростом деле внедрения технологий автономного тестирования в организации. Мы рассмотрим, как можно упростить решение этой задачи, и дадим ответы на некоторые трудные вопросы, часто возникающие при первой попытке претворить в жизнь идеи автономного тестирования.

В главе 10 мы рассмотрим типичные проблемы, относящиеся к унаследованному коду, и некоторые инструменты для их решения.

Глава 11 посвящена общим вопросам автономного тестирования. Следует ли проектировать с учетом тестопригодности? И что вообще такое тестопригодный проект?

ГЛАВА 9. Внедрение автономного тестирования в организации

В этой главе:

- Как стать инициатором перемен.
- Внедрение изменений сверху вниз и снизу вверх.
- Готовимся отвечать на трудные вопросы об автономном тестировании.

Будучи консультантом, я помогал нескольким компаниям, мелким и крупным, внедрять разработку через тестирование и автономное тестирование в корпоративную культуру. Иногда это заканчивалось неудачей, но у компаний, которым все удалось, было несколько общих черт. В этой главе я буду рассматривать следующие вопросы и черпать по ходу дела из обоих сосудов.

- *Как стать инициатором перемен*. Первые шаги, которые следует предпринять перед внедрением любых перемен.
- *Пути к успеху*. Что может стать вкладом в успех всего предприятия.
- *Пути к провалу*. Что может привести к краху всех ваших усилий.
- *Трудные вопросы и ответы на них*. Часто задаваемые вопросы о внедрении автономного тестирования в команде.

Какую бы организацию ни взять, изменение людских привычек – проблема скорее психологическая, нежели техническая. Люди не любят перемен, любая перемена обычно сопровождается страхом, неуверенностью и сомнениями. Как вы увидите из этой главы, для большинства людей это не прогулка в парке.

9.1. Как стать инициатором перемен

Если вы собираетесь стать инициатором перемен в своей организации, то для начала должны принять на себя эту роль. Хотите вы этого или нет, люди станут смотреть на вас, как на лицо, несущее ответственность за все происходящее, поэтому таиться нет смысла. Более того, попытка скрыть свою роль может привести к печальным последствиям.

Когда вы начнете внедрять перемены, люди станут задавать непростые вопросы о том, что их волнует. Сколько времени это займет? Что я, как инженер по контролю качества, буду с этого иметь? Откуда нам знать, заработает ли это? Будьте готовы отвечать. Ответы на самые типичные вопросы обсуждаются в разделе 9.4. Вы поймете, что если удастся заразить своим энтузиазмом еще кого-то внутри организации до начала перемен, то принимать трудные решения и отвечать на вопросы станет гораздо легче.

Наконец, кто-то должен все время стоять у руля, чтобы начавшиеся изменения не заглохли сами по себе. Это вы. В следующих разделах будет рассказано, как поддерживать огонь.

9.1.1. Будьте готовы к трудным вопросам

Займитесь исследованием. Прочитайте ответы в конце этой главы и изучите соответствующие ресурсы. Читайте форумы, списки рассылки и блоги, консультируйтесь с коллегами. Если вы сможете ответить на свои собственные трудные вопросы, то велики шансы, что и вопросы других не застанут вас врасплох.

9.1.2. Убедите сотрудников: сподвижники и противники

Один в поле не воин, а идти против течения в организации – едва ли не худший вид одиночества. Если вы единственный, кто считает свою задумку здравой идеей, то вряд ли кто-то станет прилагать усилия к внедрению того, за что вы ратуете. Подумайте, кто может помочь, а кто помешать. Выявите сподвижников и противников.

Сподвижники

Начав агитировать за перемены, ищите тех, кто, скорее всего, сможет помочь в вашем начинании. Это будут ваши *сподвижники*. Обыч-

но они первыми принимают новую технологию, это люди непредвзятые, готовые попробовать то, что вы предлагаете. Возможно, они уже наполовину согласны и нуждаются только в небольшом толчке, чтобы приступить к изменениям. Быть может, они уже даже пробовали сами, но без поддержки потерпели неудачу.

Поговорите, прежде всего, с ними, поинтересуйтесь их мнением о том, что вы собираетесь сделать. Возможно, они скажут что-то такое, о чем вы не задумывались: назовут группы, с которых перспективнее всего начать, или места, в которых люди более восприимчивы к подобным переменам. Они могут даже, исходя из собственного опыта, подсказать, чего следует опасаться.

Доверившись им, вы с большей вероятностью сможете привлечь их на свою сторону. Люди, ощущающие себя частью процесса, обычно стараются, чтобы он заработал. Сделайте их своими сподвижниками: спросите, согласны ли они помочь и можно ли направлять к ним людей с вопросами. Подготовьте их к такому развитию событий.

Противники

Затем выявите *противников*. Это те люди в организации, которые, скорее всего, будут противиться вводимым вами переменам. Например, какой-нибудь начальник может возражать против автономных тестов, указывая, что из-за них чрезмерно возрастет время разработки и объем нуждающегося в сопровождении кода. Сделайте их участниками, а не противниками процесса, поручив активную роль в нем (по крайней мере, тем, кто хочет и может).

Люди противятся переменам по разным причинам, ответы на некоторые возражения рассматриваются в разделе 9.4. Кто-то боится потерять работу, кто-то доволен текущим положением дел и будет против любых изменений.

Подробно объяснять таким людям, что именно они могли бы делать лучше, часто неконструктивно, я это узнал на собственном горьком опыте. Людям не нравится слышать, что они что-то делают плохо. Лучше попросите их помочь во внедрении процесса – например, заняться подготовкой стандартов автономных тестов или через день принимать участие в анализе кода и тестов, написанных коллегами. Или поручите им принять участие в подборе учебных материалов или внешних консультантов. Возложенная ответственность даст им почувствовать, что на них надеются, что они важны для организации. Таких людей надо сделать частью перемен, иначе они почти наверняка станут исподтишка вредить.

9.1.3. Выявите возможные пути внедрения

Найдите в организации места, откуда лучше всего начать внедрение перемен. Успеха обычно добиваются те, кто продвигается постепенно и неуклонно. Начните с пилотного проекта в небольшой команде и посмотрите, что получится. Если все нормально, подключайте другие команды и проекты.

Вот несколько полезных советов:

- выбирайте небольшие команды;
- создавайте подкоманды;
- принимайте во внимание осуществимость проекта;
- применяйте анализ кода и тестов как средство обучения.

Следуя этим советам, вы сможете далеко продвинуться даже в самом враждебном окружении.

Выбирайте небольшие команды

Отобрать подходящие для начала внедрения команды обычно легко. Обращайте внимание на небольшие группы, работающие над низкоприоритетным проектом с малым риском. Если риск минимален, то будет нетрудно убедить людей попробовать предлагаемые изменения в деле.

Есть, правда, одна оговорка – в команде должны быть люди, готовые изменить привычный стиль работы и учиться новому. Как ни странно, больше других открыты к изменениям наименее опытные члены команды, а те, кто поопытнее, не склонны менять укоренившиеся привычки. Если вам удастся найти команду, в которой имеется опытный руководитель с открытым складом ума и не столь опытные разработчики, то с ее стороны вы, скорее всего, почти не встретите сопротивления. Поговорите с этой командой, спросите, как она относится к тому, чтобы взять пилотный проект. Они скажут, стоит ли начинать именно отсюда.

Создавайте подкоманды

Другой кандидат на роль пилотного проекта – специально сформированная подкоманда в составе существующей команды. Почти в любой команде имеется «черная дыра» – компонент, который нужно сопровождать и который содержит много ошибок, хотя в основном работает правильно. Добавление новых функций в такой компо-

нент – трудная задача, и именно ее трудность может подвигнуть людей на эксперимент с пилотным проектом.

Принимайте во внимание осуществимость проекта

Выбирая пилотный проект, не пытайтесь откусить больше, чем сможете проглотить. Для выполнения сложных проектов нужно больше опыта, поэтому лучше иметь два варианта – проект посложнее и попроще – чтобы было из чего выбирать.

Теперь, когда вы внутренне подготовились к стоящей перед вами задаче, давайте посмотрим, что можно сделать, чтобы все шло гладко (или хоть как-то шло).

Применяйте анализ кода и тестов как средство обучения

Если вы являетесь техническим руководителем небольшой команды (до восьми человек), то один из лучших способов организовать обучение – ввести в практику анализ кода, в том числе и тестов. Идея в том, что анализируя код и тесты, написанные другими людьми, вы сможете объяснить им, на что сами обращаете внимание в тестах и как сами подходите к написанию тестов или к методике TDD. Дам несколько советов.

- Проводите анализ лично, не пользуясь программами удаленного доступа. В ходе личного общения можно передать гораздо больше невербальной информации, поэтому обучение проходит быстрее и эффективнее.
- В первые две недели подвергайте анализу каждую строчку кода, сохраняемую в системе управления версиями. Тогда никто не будет говорить «мы не думали, что этот код нуждается в анализе». Если красной черты нет вообще (то есть анализируется весь код), то и поднимать ее не придется.
- Привлекайте к анализу кода третьего участника, который будет сидеть молча и наблюдать, как вы анализируете код. Тогда впоследствии он сможет проводить анализ кода сам и обучать других, и вы перестанете быть узким местом – единственным человеком, умеющим это делать. Идея в том, чтобы другие тоже учились анализировать код и брать на себя ответственность.

Об этой методике я написал статью в своем блоге для руководителей команд, она находится по адресу http://5whys.com/blog/what-should-a-good-code-review-look-and-feellike.html.

9.2. Пути к успеху

Есть два подхода к изменению процесса в организации или команде: снизу вверх и сверху вниз (иногда оба комбинируются). Как вы скоро увидите, они сильно различаются, но для вашей команды или компании может подойти любой. Ни один путь не является единственно правильным.

По ходу дела вы должны научиться убеждать начальников в том, что *ваши* усилия должны стать и *их* усилиями, или определять момент, когда имеет смысл пригласить на помощь кого-то со стороны. Очень важно ставить ясные цели, допускающие измерение. Не меньшее значение имеет наглядная демонстрация прогресса. Задача выявления и устранения препятствий также должна стоять на одном из первых мест. Дел много, и нужно выбирать из них самые важные.

9.2.1. Партизанское внедрение (снизу вверх)

Партизанское внедрение – это когда вы тишком внедряете новую методику в команде, получаете результаты и лишь затем начинаете убеждать других, что методику стоит взять на вооружение. Обычно закоперщиком партизанского внедрения является команда, которой надоело работать по старинке. Они договариваются между собой работать по-новому, сами учатся и сами внедряют перемены. После того как команда предъявит достигнутые результаты, другие коллективы могут захотеть внедрить аналогичные изменения у себя.

Иногда партизанское внедрение означает, что процесс сначала *принимается* разработчиками, потом руководством. А иногда – что процесс сначала *пропагандируется* разработчиками, потом руководством. Разница в том, что в первом случае вы можете получить результаты тайно, не ставя в известность начальство. Во втором случае все делается при участии начальства.

Как поступить в конкретной ситуации, решать вам. Иногда тайные операции – единственный способ что-то изменить. Лучше этого избегать, но если другого пути нет, а вы уверены, что перемены необходимы, – действуйте.

Не думайте, что этот шаг обязательно поставит вашу карьеру под угрозу. Разработчики постоянно делают что-то, не спрашивая разрешения: отлаживают код, читают почту, пишут комментарии к коду,

рисуют блок-схемы и т. д. Все это часть их обычной работы. То же относится и к автономному тестированию. Большинство разработчиков все равно пишут какие-то тесты (автоматизированные или нет). Идея в том, чтобы усилия, которые так или иначе тратятся на создание тестов, переориентировать на что-то такое, что даст эффект в долгосрочной перспективе.

9.2.2. Обеспечение поддержки руководства (сверху вниз)

Движение сверху вниз обычно начинается одним из двух способов. Руководитель или разработчик инициирует процесс, а остальная организация начинает постепенно двигаться в том же направлении. Или какой-нибудь руководитель среднего звена может посмотреть презентацию, прочитать книгу (например, эту) или пообщаться с коллегой на тему преимуществ, которые сулят конкретные перемены в привычном способе работы. Затем этот руководитель инициирует процесс – проводит презентацию перед различными командами или даже вводит изменения в приказном порядке.

9.2.3. Привлечение организатора со стороны

Я горячо рекомендую привлекать стороннего консультанта для помощи во внедрении автономного тестирования и сопутствующих изменений. У него имеются следующие преимущества по сравнению с работниками компании.

- *Свобода слова.* Консультант может говорить вещи, которые сотрудники компании не хотели бы слышать от своих коллег («целостность кода никуда не годится», «ваши тесты неудобочитаемы» и т. д.).
- *Опыт.* У консультанта больше опыта по преодолению сопротивления изнутри, он может дать хорошие ответы на трудные вопросы и знает, на какие кнопки нажимать, чтобы дело пошло.
- *Наличие времени.* Для консультанта это работа. В отличие от работников компании, которым есть чем заняться, помимо борьбы за перемены (например, писать программы), консультант посвящает этой задаче все свое время.

Целостность кода

Термином *целостность кода* я описываю цель, стоящую перед командной разработкой: получение стабильного кода, удобного для сопровождения и дающего обратную связь. По существу, это означает, что код делает то, для чего задуман, и команда знает, когда это не так. Все перечисленные ниже виды работ имеют непосредственное отношение к целостности кода:

- автоматизированная сборка;
- непрерывная интеграция;
- автономное тестирование и разработка через тестирование;
- единообразие кода и согласованные стандарты качества;
- максимальное сокращение времени исправления ошибок (так чтобы отказавшие тесты стали проходить).

Некоторые считают, что это полезные свойства разработки, их можно найти в таких методологиях, как экстремальное программирование, но я предпочитаю говорить: «У нас хорошая целостность кода», а не что, по моему мнению, мы все делаем правильно.

Я часто видел, как задуманные перемены кончались пшиком, потому что перегруженный работой инициатор просто не мог уделять процессу достаточно времени.

9.2.4. Наглядная демонстрация прогресса

Важно, чтобы ход и состояние изменений были всем видны. Развесьте доски или плакаты на стенах коридоров и в местах приема пищи, где люди часто собираются. Представленные данные должны соотноситься с поставленными целями.

Например, покажите, сколько тестов прошло и не прошло в ходе последней ночной сборки. Нарисуйте диаграмму, где показано, сколько команд уже практикуют процедуру автоматизированной сборки. Повесьте диаграмму сгорания задач (методология Scrum), на которой отражен ход итераций, или отчет о покрытии кода тестами (рис. 9.1), если именно в этих терминах сформулированы цели. (Прочитать о Scrum можно на сайте www.controlchaos.com.) Опубликуйте контактные данные – свои и всех ваших сподвижников – чтобы люди знали, к кому обращаться с вопросами. Повесьте большой жидкокристаллический экран, на котором постоянно отображается четкая графическая картина состояния сборок – что сейчас выполняется и какие обнаружены ошибки. Этот экран должен находиться в таком месте, чтобы его могли видеть все разработчики, – например, в оживленном коридоре или сверху на капитальной стене помещения, где работает команда.

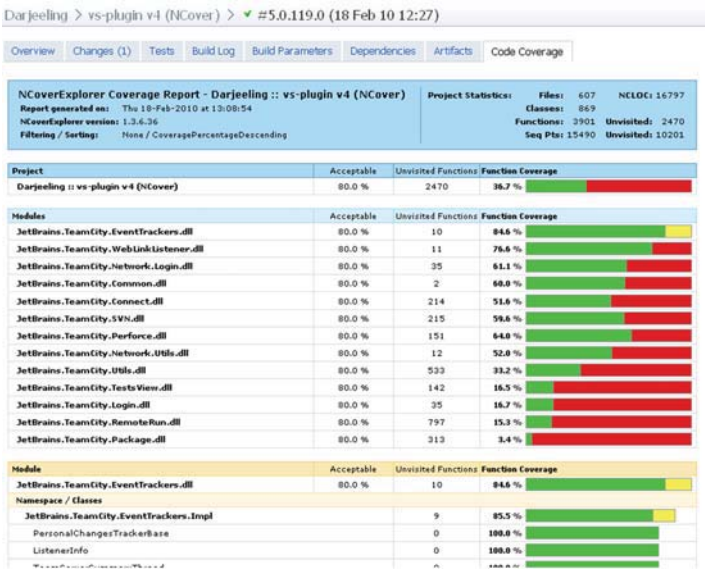


Рис. 9.1. Пример отчета о покрытии кода тестами, сформированного с помощью NCover в TeamCity

Смысл всех этих диаграмм состоит в том, чтобы навести мосты между двумя группами:

- *Группой, которая внедряет изменения*. Люди из этой группы будут испытывать чувство гордости за свои достижения, отображаемые на обновляемых диаграммах (видных всем), и ощущать, что они обязаны успешно завершить процесс, – ведь все же видят. Они также смогут сопоставлять ход работ у себя и у других. Возможно, их подстегнет информация о том, что какая-то другая группа внедряет определенные части методологии быстрее.
- *Работники, которые не участвуют в процессе*. Вы возбудите интерес и любопытство у этих людей, начнутся слухи и толки. Сформируется движение за «присоединение», которое никому не заказано.

9.2.5. Постановка конкретных целей

Без четких целей трудно измерить эффект перемен и довести его до сведения других. Это будет расплывчатое «нечто», которое легко прикрыть при первых признаках затруднений.

Ниже перечислено несколько возможных целей.

- *Увеличить объем покрытия кода тестами одновременно с проведением анализа кода и тестов.*

  В исследовании Бориса Бейзера (Boris Beizer) показано, что разработчики, которые пишут тесты, но не пользуются инструментами определения покрытия кода, выказывают наивный оптимизм относительно того, какая часть кода покрыта тестами. Из результатов другого исследования, опубликованных в книге Karl Wiegers «Peer Reviews in Software: A Practical Guide» (Addison-Wesley Professional, 2001), следует, что тестирование без применения инструментов определения покрытия кода приводит к покрытию лишь 50–60 % кода. (Существуют примечательные свидетельства в пользу того, что применение TDD позволяет достичь 95 и даже 100 %-ного покрытия логического кода.)

  Простая цель заключается в измерении процентной доли кода, покрытого тестами. Чем больше покрытие, тем выше шансы обнаружить ошибки. Хотя это и не панацея. Очень просто достичь почти 100 %-ного покрытия плохими тестами, которые ровным счетом ничего не значат. Низкое покрытие – заведомо плохой знак, высокое – знак того, что, возможно, дело обстоит лучше.

  Что действительно нужно, так это высокое покрытие в сочетании с постоянным анализом кода и тестов (о чем я говорил выше в этой главе). Тогда будет уверенность, что тесты пишутся не просто для того, чтобы формально выполнить требования к покрытию (например, безо всяких утверждений), а на самом деле что-то значат.

Примечание. Исследование Бориса Бейзера обсуждается в статье Марка Джонсона (Mark Johnson) «Dr. Boris Beizer on Software Testing: An Interview. Part I» в журнале «The Software QA Quarterly» (лето 1994).

- *Увеличить объем покрытия кода тестами сравнительно с объемом модификации кода.*

  Некоторые системы позволяют измерять объем *модификации кода* – сколько строк изменилось между сборками. Чем меньше строк изменилось, тем меньше вероятность внесения ошибок в систему. Вычисление этой метрики не всегда осмысленно, особенно в системах, где в процессе сборки много кода генерируется автоматически, но эту проблему можно решить,

игнорируя генерируемый код. В частности, измерять модификацию кода умеет Microsoft Team System. (См. статью «Analyze and Report on Code Churn and Code Coverage Using the Code Churn and Run Coverage Perspectives» по адресу http://msdn.microsoft.com/en-us/library/vstudio/ms244661.aspx.)

- *Уменьшить количество вновь внесенных ошибок.*

  Очень просто, исправляя одно, случайно сломать что-то другое. Если это происходит редко, значит, вы умеете исправлять дефекты и сопровождать систему, не нарушая ранее сделанных допущений.

- *Уменьшить среднее время исправления ошибки (время от обнаружения ошибки до закрытия проблемы).*

  В системе с хорошими тестами и высоким покрытием ошибки обычно исправляются быстрее (если, конечно, тесты удобны для сопровождения). Это, в свою очередь, означает, что оборотное время уменьшается, а циклы выпуска версий менее напряженные.

В книге «Code Complete» (Microsoft Press, второе издание, 2004) Стив Макконелл (Steve McConnell) приводит несколько метрик для количественной оценки хода работ, а именно:

- количество дефектов на один класс по приоритетам;
- количество дефектов на стандартное количество часов тестирования на одну найденную ошибку;
- среднее количество дефектов на один тестовый сценарий.

Я настоятельно рекомендую прочитать главу 22 книги Макконелла всем, кто занимается тестированием в процессе разработки.

9.2.6. Осознание неизбежности препятствий

Препятствия есть всегда. По большей части они проистекают из организационной структуры, но есть и технические. Технические устранить легче, так как всего-то и требуется что найти подходящее решение. Организационные требуют внимания и учета психологии.

Важно не считать, что все пропало, при первой же временной неудаче, когда не складывается с итерацией, тесты работают медленнее, чем ожидалось, и т. п. Иногда наладить процесс сложно и нужно потерпеть хотя бы пару месяцев, а потом вы освоитесь, и все морщинки

разгладятся. Нужно убедить руководство не отказываться от идеи по меньшей мере три месяца, даже если не все идет по плану. И согласием важно заручиться заранее. У вас не будет времени на убеждения в разгар самого напряженного первого месяца.

Советую также усвоить мудрую мысль, которой Тим Оттингер (Tim Ottinger) поделился в Твиттере (@Tottinge): «Даже если ваши тесты отлавливают не все дефекты, они все равно упрощают исправление ошибок, которые не отловили. Это чистая правда».

Итак, мы обсудили, как вести себя, чтобы все было хорошо. А теперь посмотрим, что может привести к провалу.

9.3. Пути к провалу

В предисловии к этой книге я упомянул об одном проекте с моим участием, который провалился отчасти из-за неправильного внедрения автономного тестирования. Это один из способов потерпеть неудачу. Здесь я приведу еще несколько, в том числе и способ, который стоил мне того проекта, а также посоветую, что с этим можно сделать.

9.3.1. Отсутствие движущей силы

Всюду, где я встречался с безуспешными попытками внедрения перемен, основной причиной было отсутствие движущей силы. Быть инициатором и постоянной движущей силой перемен нелегко. Чтобы учить других, помогать им и бороться с внутренними политическими склоками, нужно отвлекаться от своей обычной работы. Вы должны быть готовы жертвовать своим временем, иначе никаких изменений не будет. В разделе 9.2.3 я уже отмечал, что приглашение консультанта со стороны может решить проблему неослабевающей движущей силы.

9.3.2. Отсутствие политической поддержки

Если начальник прямо говорит, что никаких изменений не потерпит, то мало что можно сделать – разве что попытаться убедить его взглянуть на вещи вашими глазами. Но иногда отсутствие поддержки принимает более тонкие формы и нужно еще осознать, что вы столкнулись с оппозицией.

Например, вам могут сказать: «Давай, вперед, внедряй свои тесты. Можешь тратить на это 10 % своего времени». Но если не можешь

уделять хотя бы 30 % времени, то нечего даже и пытаться внедрить автономное тестирование. Это один из способов, который начальник может использовать, чтобы положить конец нежелательной инициативе, – задушить ее на корню.

Вы должны заподозрить, что имеет место противодействие, а подтвердить подозрение уже нетрудно. Возразив, что такие ограничения нереалистичны, вы в ответ услышите: «Ну что ж, тогда не делай».

9.3.3. Плохая организация внедрения и негативные первые впечатления

Если вы собрались внедрять автономное тестирование, не зная толком, как писать хорошие автономные тесты, сделайте себе одолжение: привлеките кого-нибудь с опытом и следуйте проверенным рекомендациям (например, изложенным в этой книге).

Я встречал разработчиков, с разбега прыгающих в воду, не понимая, что делать и с чего начать. Я бы не хотел оказаться на их месте. Мало того что придется потратить уйму времени, чтобы понять, как внедрить изменения применительно к вашей конкретной ситуации, так еще, начав с неудачной организации, вы утратите доверие. Это может кончиться закрытием пилотного проекта.

Если вы читали предисловие к этой книге, то знаете, что со мной такое случалось. У вас есть всего два месяца, чтобы набрать темп и убедить начальство, что результаты налицо. Составьте график и устраните все риски, какие сможете. Если вы не знаете, как писать хорошие тесты, почитайте книжку или наймите консультанта. Если не знаете, как сделать код тестопригодным, поступите точно так же. Не тратьте время на изобретение методов тестирования заново.

9.3.4. Отсутствие поддержки со стороны команды

Если команда не поддерживает ваши усилия, то добиться успеха практически невозможно, потому что вам будет исключительно сложно сочетать дополнительную работу над новым процессом с повседневными обязанностями. Вам придется бороться за то, чтобы команда приняла участие в новом процессе или хотя бы не мешала.

Поговорите с членами команды о переменах. Иногда лучше начать с перетягивания их на свою сторону поодиночке, но и беседа о ваших устремлениях со всеми вместе – с ответами на трудные вопро-

сы – тоже имеет свою ценность. Что бы вы ни делали, не думайте, что поддержка команды – нечто само собой разумеющееся. Убедитесь, что точно знаете, во что ввязываетесь; это те люди, с которыми вам предстоит работать каждый день.

В любом случае вам будут задавать трудные вопросы об автономном тестировании. Ниже приведены вопросы, к которым вы должны быть готовы в дискуссиях с людьми, от которых зависит успех или провал ваших планов. И ответы на эти вопросы.

9.4. Факторы влияния

Даже больше, чем автономные тесты, меня очаровывают люди и мотивы их поведения. До чего обидно, когда вы пытаетесь убедить человека начать что-то делать (к примеру, TDD), а он, несмотря на все ваши усилия, не хочет и все тут. Вы пытаетесь втолковать ему свою точку зрения, но видите, что он ничего не воспринимает.

По поводу оказания влияния на людей есть замечательная книга: Kerry Patterson, Joseph Grenny, David Maxfield, Ron McMillan, Al Switzler «Influencer: The Power to Change Anything»[1] (McGraw-Hill, 2007). Ссылка на нее имеется на странице http://5whys.com/recommendedbooks/. В этой книге снова и снова повторяется одна глубокая мысль: какое бы поведение ни взять, мир скроен так, что оно может случиться. Это означает, что, помимо желания или умения человека что-то делать, существуют и другие факторы, влияющие на его поведение. Только мы редко пытаемся рассмотреть что-то, кроме этих двух факторов. В книге перечисляются шесть факторов влияния.

Личное умение	Обладает ли человек знаниями и навыками для выполнения требуемого?
Личная мотивация	Получает ли человек удовольствие от правильного поведения и неудовольствие от неправильного? Способен ли человек к самоконтролю в достаточной мере, чтобы вести себя правильно, когда это очень трудно?
Социальная поддержка	Вы или еще кто-то предоставляете помощь, информацию и ресурсы человеку, который в них нуждается, особенно в критические моменты?

[1] Кэрри Паттерсон, Джозеф Гренни, Дэвид Максфилд, Рон Макмиллан, Ал Свайтзле «Как влиять на других. Принципы, методы, примеры». Вильямс, 2008. – *Прим. перев.*

Социальная мотивация	Верно ли, что окружающие активно поддерживают правильное поведение и осуждают неправильное? Подаете ли вы и другие люди пример правильного поведения?
Структурная поддержка (со стороны окружения)	Существуют ли в окружающей среде (здание, бюджет и т. п.) факторы, которые делают поведение удобным, легким и безопасным? Достаточно ли подсказок и напоминаний, чтобы не сбиться с курса?
Структурная мотивация	Существует ли четкое и значимое вознаграждение (например, в виде зарплаты, премий или иных материальных стимулов) за правильное поведение и наказание за неправильное? Соответствует ли краткосрочное воздаяние желательным долгосрочным результатам или видам поведения, которые требуется подкрепить или предотвратить?

Задумайтесь над этим коротким списком, когда захотите понять, почему все идет не так, как вы хотите. А потом учтите еще один важный факт: возможно, есть какой-то фактор, которого вы не видите. Чтобы изменить поведение, нужно изменить все влияющие на него факторы. Если вы измените только один, поведение останется прежним.

Ниже приведен пример воображаемого контрольного списка, который я составил применительно к человеку, не желающему работать в соответствии с TDD. (Имейте в виду, что для каждого человека в каждой организации потребуется свой список.)

Личное умение	Обладает ли человек знаниями и навыками для выполнения требуемого?	Да. Он прошел трехдневный курс по TDD под руководством Роя Ошероува.
Личная мотивация	Получает ли человек удовольствие от правильного поведения и неудовольствие от неправильного? Способен ли человек к самоконтролю в достаточной мере, чтобы вести себя правильно, когда это очень трудно?	Я с ним говорил, ему нравится TDD.
Социальная поддержка	Вы или еще кто-то предоставляете помощь, информацию и ресурсы человеку, который в них нуждается, особенно в критические моменты?	Да.

Социальная мотивация	Верно ли, что окружающие активно поддерживают правильное поведение и осуждают неправильное? Подаете ли вы и другие люди пример правильного поведения?	Настолько, насколько это возможно.
Структурная поддержка (со стороны окружения)	Существуют ли в окружающей среде (здание, бюджет и т. п.) факторы, которые делают поведение удобным, легким и безопасным? Достаточно ли подсказок и напоминаний, чтобы не сбиться с курса?	* В организации нет денег для приобретения системы сборки.
Структурная мотивация	Существует ли четкое и значимое вознаграждение (например, в виде зарплаты, премий или иных материальных стимулов) за правильное поведение и наказание за неправильное? Соответствует ли краткосрочное воздаяние желательным долгосрочным результатам или видам поведения, которые требуется подкрепить или предотвратить?	* Когда человек пытается заняться автономным тестированием, начальник говорит, что он зря тратит время. Если продукт будет поставлен досрочно, пусть даже отвратительного качества, отдел получит премию.

Я поставил звездочки против пунктов, требующих принятия мер. В данном случае я выявил два таких пункта. Добиться выделения денег на систему сборки недостаточно, поведение при этом не изменится. Необходимо установить систему сборки *и* покончить с практикой выплаты премий за досрочную поставку никуда не годной программы.

В книге «Notes to a Software Team Leader», посвященной руководству технической командой, я уделяю гораздо больше внимания этой теме. Вы можете найти книгу на сайте 5whys.com.

9.5. Трудные вопросы и ответы на них

В этом разделе обсуждаются некоторые вопросы, которые мне задавали в разных местах. Обычно в основе любого вопроса лежит предположение, что внедрение автономного тестирования может ущемить чьи-то интересы – начальника, озабоченного соблюдением сроков, или работника отдела контроля качества, опасающегося, не окажется ли он ненужным. Поняв, с чем связан вопрос, вы должны разрешить исходную проблему, прямо или косвенно. В противном случае столкнетесь со скрытым сопротивлением.

9.5.1. Насколько автономное тестирование замедлит текущий процесс?

Руководители команд, менеджеры проектов и клиенты – вот кто задает вопрос о дополнительных временных затратах. Эти люди в первую очередь обеспокоены соблюдением графиков.

Начнем с фактов. Исследования показывают, что подъем общего качества кода повышает продуктивность и сокращает сроки поставки. Как это совмещается с тем, что написание тестов замедляет кодирование? В основном, экономия достигается за счет более удобного сопровождения и простоты исправления ошибок.

Примечание. Результаты исследований по качеству кода и продуктивности см. в книгах «Programming Productivity» (McGraw-Hill College, 1986) и «Software Assessments, Benchmarks, and Best Practices» (Addison-Wesley Professional, 2000). Обе написал Кейперс Джонс (Capers Jones).

Когда руководитель команды спрашивает о времени, он на самом деле хочет знать: «Что мне говорить менеджеру проекта, если мы не уложимся в сроки?» Возможно, процесс ему нравится, но он хочет вооружиться для будущей битвы. И быть может, его интересует не продукт в целом, а какая-то конкретная функциональность.

С другой стороны, тот же вопрос из уст менеджера проекта или заказчика обычно относится к графику выпуска полных версий продукта.

Поскольку смысл вопроса зависит от спрашивающего, ответ должен учитывать это различие. Например, автономное тестирование, возможно, вдвое увеличит время, необходимое для реализации конкретной функции, но сроки выпуска всего продукта могут даже сократиться. Чтобы понять, как такое возможно, рассмотрим реальный пример проекта, в котором я принимал участие.

Рассказ о двух функциях

Крупная компания, которую я консультировал, захотела внедрить в свой процесс автономное тестирование, начав с пилотного проекта. В пилотном проекте участвовала группа разработчиков, перед которыми стояла задача добавить новую функцию в большое существующее приложение. Основным источником дохода компании была сложная программа по выставлению счетов, которая адаптировалась под нужды различных клиентов. На компанию работали тысячи программистов в разных частях света.

Для оценки успешности пилотного проекта были выбраны следующие метрики:

- время, потраченное командой на каждом этапе разработки;
- общее время до передачи продукта заказчику;
- количество ошибок, найденных заказчиком после выпуска версии.

Аналогичная статистика собиралась для проекта создания похожей функции другой командой по заказу другого клиента. Обе функции были примерно равнозначны по размеру, навыки и квалификация команд тоже были примерно одинаковы. В обоих случаях нужно было адаптировать программу под заказчика – с автономными тестами и без них. В табл. 9.1 приведены данные о времени.

Таблица 9.1. Время выполнения различных этапов с автономными тестами и без

Этап	Без тестов	С тестами
Реализация (кодирование)	7 дней	14 дней
Интеграция	7 дней	2 дня
Тестирование и исправление ошибок	Тестирование – 3 дня Исправление – 3 дня Тестирование – 3 дня Исправление – 2 дня Тестирование – 1 день Всего: 12 дней	Тестирование – 3 дня Исправление – 1 день Тестирование – 2 дня Исправление – 1 день Тестирование – 1 день Всего: 8 дней
Общее время до выпуска	26 дней	24 дня
Количество ошибок в готовой системе	71	11

Общее время до выпуска оказалось меньше у команды, работавшей с тестами. Тем не менее, менеджеры поначалу сочли пилотный проект неудачным, потому что рассматривали в качестве критерия только статистику кодирования (первая строка в табл. 9.1), а не числа в последней строке. На кодирование ушло вдвое больше времени (потому что автономные тесты тоже нужно писать). Но дополнительно затраченное время более чем компенсировалось сокращением числа найденных ошибок, с которыми пришлось разбираться отделу контроля качества.

Поэтому важно подчеркивать, что хотя при использовании автономного тестирования время реализации функции может возрастать, общее время на протяжении всего цикла разработки не увеличивается благодаря более высокому качеству и удобству сопровождения.

9.5.2. Не станет ли автономное тестирование угрозой моей работе в отделе контроля качества?

Автономное тестирование не устраняет необходимость в контроле качества. Инженерам по контролю качества будут передаваться комплекты автономных тестов, то есть они могут удостовериться, что все автономные тесты проходят, перед тем как приступить к собственному процессу тестирования. Наличие автономных тестов сделает их работу даже интереснее. Вместо отладки в пользовательском интерфейсе (где каждое второе нажатие кнопки приводит к какому-нибудь исключению) они смогут сосредоточиться на поиске ошибок логического характера в реальных сценариях. Автономные тесты – это лишь первый эшелон защиты от ошибок, а отдел контроля качества обеспечивает второй эшелон – приемку пользователем. Как и в случае безопасности, приложение нуждается в нескольких уровнях защиты. Раз отдел контроля качества может сконцентрировать внимание на более крупных вопросах, приложение получится лучше.

В некоторых компаниях инженеры по контролю качества сами пишут код, тогда они могут принять участие в создании автономных тестов. При этом они трудятся вместе с разработчиками, а не вместо них. Как разработчики, так и инженеры по контролю качества имеют право писать автономные тесты.

9.5.3. Откуда нам знать, что автономные тесты и вправду работают?

Чтобы определить, работают автономные тесты или нет, создайте какую-нибудь метрику вроде тех, что обсуждались в разделе 9.2.5. Если вы сможете произвести замеры, то будете знать точно. К тому же, вы почувствуете это интуитивно.

На рис. 9.2 показан пример отчета о покрытии кода тестами (покрытие в зависимости от номера сборки). Автоматическое создание подобного отчета (с помощью инструмента типа NCover для .NET) во время сборки может продемонстрировать прогресс в одном аспекте разработки.

Покрытие кода – неплохая отправная точка для тех, кто интересуется, не пропущены ли какие-то автономные тесты.

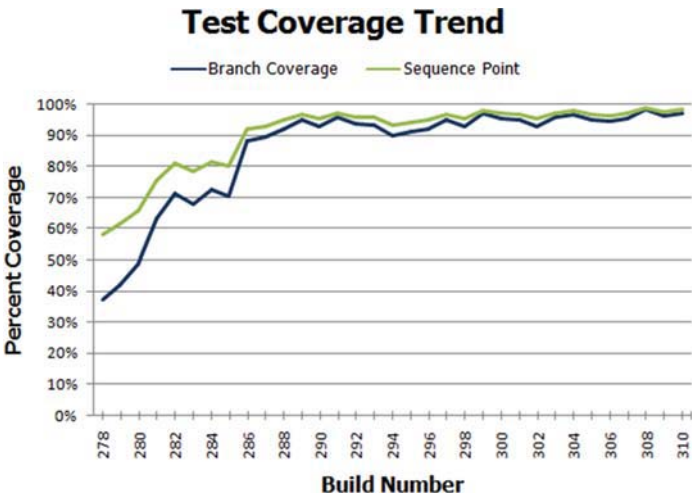


Рис. 9.2. Пример отчета о тенденции покрытия кода тестами

9.5.4. Есть ли доказательства, что автономное тестирование действительно помогает?

Я не могу указать специальных исследований, в которых изучалось бы, помогают автономные тесты улучшить качество кода или нет. В большинстве исследований речь идет о внедрении тех или иных методик гибкой разработки, в том числе автономного тестирования. Но в Сети можно найти эмпирические свидетельства в пользу того, что компании и отдельные разработчики добивались впечатляющих результатов и не желали возвращаться к кодированию без тестов.

Некоторые исследования на тему TDD можно найти на странице http://biblio.gdinwiddie.com/biblio/Studies-OfTestDrivenDevelopment.

9.5.5. Почему отдел контроля качества по-прежнему находит ошибки?

Работа инженера по контролю качества – искать ошибки на разных уровнях, атаковать приложение с разных направлений. Обычно это интеграционное тестирование, в ходе которого можно находить ошибки, недоступные автономным тестам. Например, ошибки могут

проявиться на стыке разных компонентов в готовой системе, хотя каждый компонент в отдельности успешно проходит все автономные тесты (работающие в изоляции друг от друга). Кроме того, инженер по контролю качества может проводить тестирование в терминах вариантов использования и законченных сценариев, что автономным тестам обычно неподвластно. При таком подходе выявляются логические ошибки и ошибки, препятствующие приемке заказчиком, и это тоже немало способствует повышению качества проекта.

В исследовании Гленфорда Майерса (Glenford Myres) показано, что разработчики, которые писали тесты, задачи поиска ошибок перед собой не ставили, поэтому обнаруживали только от половины до двух третей ошибок, имеющихся в приложении. Короче говоря, это означает, что для инженеров по контролю качества работа всегда найдется. И хотя этому исследованию уже больше 34 лет, я полагаю, что и сегодня сохраняется тот же менталитет, поэтому его результаты все еще значимы, по крайней мере, для меня.

Примечание. Результаты исследования Гленфорда Майерса обсуждаются в работе «A controlled experiment in program testing and code walkthroughs/inspections», Communications of the ACM 21, № 9 (сентябрь 1979), 760–69.

9.5.6. У нас полно кода без тестов: с чего начать?

Исследования, проведенные в 1970-е и 1990-е годы, показали, что 90 % ошибок содержатся примерно в 20 % кода. Проблема в том, чтобы найти тот код, где больше всего ошибок. Как правило, любая команда может сказать, какие компоненты вызывают наибольшие опасения. Вот с них и начните. Можно добавить метрики, например упомянутые в разделе 9.2.5, показывающие количество ошибок на класс.

Примечание. Исследования, доказывающие, что 90 % ошибок сосредоточено в 20 % кода, опубликованы в следующих статьях: Albert Endres «An analysis of errors and their causes in system programs» IEEE Transactions on Software Engineering 2 (июнь 1975), 140–49; Lee L. Gremillion «Determinants of program repair maintenance requirements» Communications of the ACM 27, № 9 (август 1994), 926–32; Barry W. Boehm «Industrial software metrics top 10 list» IEEE Software 4, № 9 (сентябрь 1997), 94–95; Shull и др. «What we have learned about fighting defects» Proceedings of the 9th International Symposium on Software Metrics (2002), 249–59.

Тестирование унаследованного кода требует иных подходов, чем написание нового параллельно с тестами. Подробнее об этом см. главу 10.

9.5.7. Мы работаем на нескольких языках, возможно ли при этом автономное тестирование?

Иногда на одном языке можно написать тесты для кода на другом языке, особенно если речь идет о разных языках на платформе .NET. Например, на C# можно писать тесты для кода на VB.NET. Бывает, что каждая команда пишет тесты на том языке, на котором ведет разработку: программисты на C# пишут тесты на C#, пользуясь одним из многочисленных доступных каркасов (MSTest, NUnit приходят на ум первыми), и программисты на C++ могут писать тесты с помощью какого-нибудь каркаса, ориентированного на C++, например CppUnit. Мне встречались случаи, когда программисты, работавшие на C++, писали вокруг своего кода обертки на управляемом C++, а затем на C# – тесты этих оберток. Это упрощало как написание, так и сопровождение тестов.

9.5.8. А что, если мы разрабатываем программно-аппаратные решения?

Если ваше приложение включает комбинацию программного и аппаратного обеспечения, то тесты нужно писать только для программной части. Скорее всего, у вас уже имеется какой-то имитатор оборудования, которым могут воспользоваться тесты. Работы немного больше, но это, безусловно, осуществимо, и многие компании давно так делают.

9.5.9. Откуда нам знать, что в тестах нет ошибок?

Вы должны следить за тем, чтобы тесты падали, когда должны падать, и проходили, когда должны проходить. Применение TDD – прекрасный способ гарантировать, что такие вещи не будут забыты. (См. краткий обзор TDD в главе 1.)

9.5.10. Мой отладчик показывает, что код работает правильно. К чему мне еще тесты?

Отладчики – плохие помощники при работе с многопоточным кодом. Кроме того, в своем коде вы, может, и уверены, а как насчет чужого? Откуда вам знать, что он работает? Откуда им знать, что работает ваш код, и что они ничего не сломают, внося изменения? Напомню, что кодирование – лишь первый этап в жизненном цикле программы. Большую часть своей жизни она проводит в режиме сопровождения. Вы должны позаботиться о том, чтобы код сообщал людям о том, что сломался. Для этого и служат автономные тесты.

Исследование Кэртиса, Краснера и Айскоу доказало, что большинство дефектов связано не с кодом как таковым, а является результатом недопонимания, постоянного изменения требований и недостаточного знания предметной области. Даже лучший в мире кодировщик, скорее всего, напишет неправильный код, если ему так поставят задачу. А когда потребуется внести в программу изменения, вы будете рады, что для всего остального имеются тесты, гарантирующие, что ничего не сломалось.

Примечание. Упомянутое исследование опубликовано в статье Bill Curtis, H. Krasner, N. Iscoe «A field study of the software design process for large systems» Communications of the ACM 31, № 11 (ноябрь 1999), 1269–97.

9.5.11. Мы обязательно должны вести разработку через тестирование?

Выбор TDD – вопрос стиля. Лично я вижу в TDD много полезного, и многие считают, что эта методика повышает продуктивность и вообще обладает целым рядом преимуществ, но есть люди, которые пишут тесты после кода и прекрасно себя чувствуют. Решайте сами.

Если этот вопрос возникает из опасения менять слишком много сразу, то разбейте процесс обучения на несколько шагов:

- изучите автономное тестирование по книгам вроде этой и пользуйтесь инструментами типа Typemock Isolator ил JMockIt, чтобы не нужно было думать о проблемах проектирования в ходе тестирования;
- изучите достойную технику проектирования, например SOLID (обсуждается в главе 11);

- изучите методику разработки через тестирование (могу рекомендовать книгу Kent Beck «Test-Driven Development: By Example»[2]).

При таком подходе обучение будет проще, и вы сможете приступить к делу быстрее, не отрывая много времени от работы над проектом.

9.6. Резюме

Внедрение автономного тестирования в организации – вещь, с которой рано или поздно придется столкнуться многим читателям этой книги. Будьте готовы. Убедитесь, что сможете ответить на вопросы, которые вам, вероятно, зададут. Не отталкивайте людей, которые могут вам помочь. Готовьтесь к возможной скачке с препятствиями. Помните о факторах влияния.

В следующей главе мы поговорим об унаследованном коде и рассмотрим имеющиеся инструменты и методы работы с ним.

[2] Кент Бек «Экстремальное программирование. Разработка через тестирование». Питер, 2003. – *Прим. перев.*

ГЛАВА 10. Работа с унаследованным кодом

В этой главе:

- Типичные проблемы унаследованного кода.
- С чего начинать написание тестов.
- Обзор полезных инструментов для работы с унаследованным кодом.

Как-то мне довелось консультировать крупную компанию, которая изготавливала ПО для выставления счетов. В компании работало более 10 000 разработчиков, которые писали на смеси .NET, Java и C++. Были там и продукты, и субпродукты и переплетенные проекты. В той или иной форме программа существовала свыше пяти лет, и большая часть разработчиков занималась сопровождением и надстройками над существующей функциональностью.

В мою задачу входило оказание помощи нескольким подразделениям (в которых использовались все языки) в освоении методики TDD. Примерно 90 % программистов, с которыми я работал, так и не перешли на TDD по разным причинам, часть из которых была связана с унаследованным кодом:

- было трудно писать тесты для проверки существующего кода;
- было почти невозможно переработать существующий код (или на это не хватало времени);
- некоторые не хотели менять структуру кода;
- мешал инструментарий (или его отсутствие);
- было трудно понять, с чего начать.

Всякий, кто хоть раз пытался добавить тесты в существующую систему, понимает, что для большинства таких систем написать тесты

почти невозможно. Обычно при их разработке никто не думал о местах (зазорах), в которые можно было бы вставить расширения или другие компоненты взамен существующих.

Имея дело с унаследованным кодом, нужно разрешить несколько проблем.

- Работы уйма, с чего начинать тестирование? На чем сосредоточить усилия?
- Как можно безопасно подвергать код рефакторингу, если никаких тестов еще нет?
- Какие существуют инструменты для работы с унаследованным кодом?

В этой главе мы постараемся ответить на эти трудные вопросы: перечислим технические приемы, а также приведем ссылки на ресурсы и инструменты.

10.1. С чего начать добавление тестов?

В предположении, что существующий код находится внутри компонентов, нужно создать упорядоченный список компонентов, которые полезнее всего протестировать. На место компонента в списке влияет несколько факторов.

- *Логическая сложность*. Речь идет о количестве логических конструкций в компоненте: вложенных `if`, предложений `switch`, рекурсии. Для определения этой характеристики можно использовать инструменты, вычисляющие цикломатическую сложность.
- *Уровень зависимости*. Под этим понимается количество других компонентов, от которых зависит данный. Сколько зависимостей придется разорвать, чтобы протестировать класс? Быть может, он взаимодействует с внешним почтовым компонентом или вызывает какой-нибудь статический метод протоколирования?
- *Приоритет*. Это общий приоритет компонента в проекте.

Каждому компоненту можно назначить оценку по каждому из этих факторов, от 1 (наименее важный) до 10 (наиболее важный).

В табл. 10.1 показано несколько классов с оценками указанных факторов. Я называю ее *таблицей целесообразности тестов*.

Таблица 10.1. Простая таблица целесообразности тестов

Компонент	Логическая сложность	Уровень зависимости	Приоритет	Примечания
`Utils`	6	1	5	В этом служебном классе мало зависимостей, но много логики. Протестировать его будет легко, и это принесет большую пользу.
`Person`	2	1	1	Это класс для хранения данных. В нем мало логики и нет никаких зависимостей. От его тестирования пользы мало (хотя и есть).
`TextParser`	8	4	6	В этом классе много логики и зависимостей. Ко всему прочему, он решает в проекте высокоприоритетную задачу. Его тестирование принесет много пользы, но это трудное и долгое дело.
`ConfigManager`	1	6	1	В этом классе хранятся конфигурационные параметры, прочитанные из файла на диске. В нем мало логики, но много зависимостей. Его тестирование не принесет особой пользы проекту, к тому же обещает быть трудным и долгим.

На основе табл. 10.1 мы можем нарисовать диаграмму, показанную на рис. 10.1, где компоненты изображены по значимости для проекта и по числу зависимостей.

Можно спокойно игнорировать все, что оказалось ниже выбранного вами порога логической сложности (я обычно задаю его равным 2 или 3), поэтому компоненты `Person` и `ConfigManager` нас интересовать не будут. Остаются только два верхних компонента на рис. 10.1.

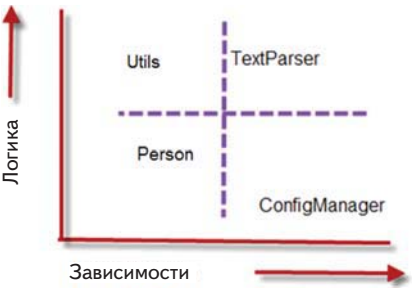


Рис. 10.1. Графическое представление компонентов для оценки целесообразности тестов

Есть два основных способа интерпретации этой диаграммы с целью решить, что тестировать сначала (см. рис. 10.2).

- Выбирать логически сложные компоненты, которые проще тестировать (левый верхний квадрант).
- Выбирать логически сложные компоненты, которые труднее тестировать (правый верхний квадрант).

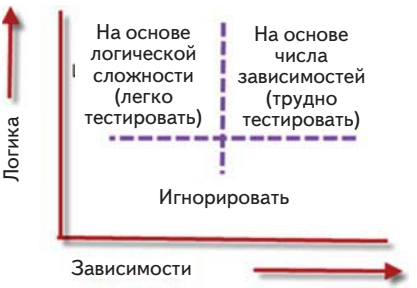


Рис. 10.2. Классификация компонентов на простые, трудные и несущественные, исходя из логической сложности и числа зависимостей

Теперь возникает вопрос, по какому пути идти. Начать с простого или с трудного?

10.2. На какой стратегии выбора остановиться

В предыдущем разделе было сказано, что можно начать с компонентов, которые легко тестировать, или с тех, которые тестировать трудно (потому что у них много зависимостей). У каждой стратегии есть свои проблемы.

10.2.1. Плюсы и минусы стратегии «сначала простые»

Если начать с компонентов, у которых меньше зависимостей, то первые тесты можно будет написать легко и быстро. Но тут есть подвох, изображенный на рис. 10.3.

На рис. 10.3 показано, сколько времени занимает подготовка компонентов к тестированию на протяжении срока жизни проекта. Поначалу тесты писать легко, но чем дальше, тем труднее тестировать каждый следующий компонент, а самые трудные поджидают в конце цикла, когда все уже с нетерпением ждут выпуска продукта в мир.

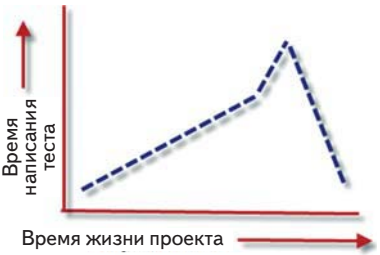


Рис. 10.3. Если начинать с простых компонентов, то время создания тестов при переходе к более трудным будет возрастать

Если команда еще не поднаторела в автономном тестировании, то имеет смысл начинать с легких компонентов. Со временем команда освоит приемы, необходимые для тестирования более трудных компонентов с большим числом зависимостей.

Такой команде лучше поначалу вообще избегать компонентов, для которых число зависимостей больше определенного порога (я считаю разумным порог 4).

10.2.2. Плюсы и минусы стратегии «сначала трудные»

Предложение начинать с более трудных компонентов может показаться бессмысленным, но с точки зрения команды, имеющих богатый опыт автономного тестирования, у него есть достоинства.

На рис. 10.4 показано среднее время написания теста одного компонента на протяжении срока жизни проекта в случае, когда сначала тестируются компоненты с большим числом зависимостей.

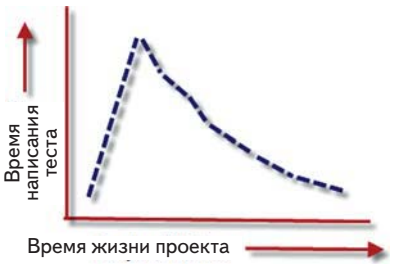


Рис. 10.4. Если начинать с трудных компонентов, то время создания тестов поначалу велико, но уменьшается по мере устранения зависимостей в ходе рефакторинга

При такой стратегии можно потратить целый день, а то и больше на написание даже простейших тестов для самых сложных компонентов. Но обратите внимание, как быстро спадает время создания одного теста по сравнению с медленным ростом на рис. 10.3. Тестируя оче-

редной компонент и перерабатывая его, чтобы сделать более тестопригодным, вы заодно решаете проблемы зависимостей, встречающихся в нем и в других компонентах. Точнее, рефакторинг компонента с большим числом зависимостей может улучшить положение дел в других частях системы. Отсюда и быстрый спад.

Стратегия «сначала трудные» применима, только если команда имеет достаточный опыт автономного тестирования, потому что реализовать ее сложнее. Если такой опыт есть, ориентируйтесь на приоритет компонентов, решая, с каких – трудных или легких – начать. Можно выбрать и комбинированную стратегию, но важно заранее понимать, сколько потребуется усилий и каковы возможные последствия.

10.3. Написание интеграционных тестов до рефакторинга

Если вы таки планируете переработать код, сделав его тестопригодным (чтобы можно было писать автономные тесты), то реальный способ гарантировать, что в ходе рефакторинга ничего не сломается, – написать интеграционные тесты для работающей системы.

Я был консультантом в крупном унаследованном проекте и работал с программистом, которому был поручен менеджер конфигурации на базе XML-файлов. Тестов в проекте не было, и тестопригодным его можно было назвать с большим трудом. Проект к тому же был написан на C++, поэтому мы не могли воспользоваться инструментом типа Typemock Isolator для изоляции компонентов без рефакторинга кода.

Программист должен был добавить в XML-файл еще одну пару атрибут–значение и обеспечить чтение и изменение значения с помощью имеющегося компонента конфигурации. В конце концов мы написали два интеграционных теста, которые пользовались реальной системой для сохранения и загрузки конфигурационных данных и содержали утверждения относительно значений, которые компонент читал и записывал. Эти тесты определили «исходное» поведение менеджера конфигурации, от которого мы могли плясать.

Затем мы написали интеграционный тест, который показывал, что сразу после чтения файла компонентом в памяти не было атрибута, который мы собирались добавить. Мы доказали, что эта функциональность действительно раньше отсутствовала, и получили тест, который должен был бы пройти, после того как мы добавим новый атрибут в XML-файл и корректно запишем файл из компонента.

Написав код, который сохранял и загружал дополнительный атрибут, мы прогнали все три интеграционных теста (два первоначальных и новый, который пытался читать новый атрибут). Все три прошли, так что мы убедились, что, добавляя новую функциональность, ничего не сломали.

Как видите, процесс сравнительно прост:

- добавить один или несколько интеграционных тестов (никаких подставок и заглушек), доказывающих, что исходная система работает правильно;
- провести рефакторинг или добавить падающий тест для функциональности, которую вы собираетесь добавить в систему;
- проводить рефакторинг и понемногу вносить в систему изменения, как можно чаще прогоняя интеграционные тесты, которые проверяют, что ничего не сломалось.

Иногда может показаться, что интеграционные тесты писать проще, чем автономные, потому что не надо возиться с внедрением зависимостей. Но прогон таких тестов на локальной машине может оказаться докучным и трудоемким делом, потому что необходимо сконфигурировать систему, так чтобы все оказалось на своих местах.

Штука в том, чтобы работать только с теми частями системы, которые требуется исправить или дополнить. Не трогайте ничего другого. Тогда система будет расти в нужных местах, и новые проблемы можно будет решать по мере поступления.

Добавляя все новые и новые тесты, вы будете перерабатывать систему, включать в нее автономные тесты, постепенно приводя ее к удобному для сопровождения и тестопригодному виду. Это требует времени (иногда много месяцев), но результат того стоит.

Я, помнится, говорил, что нужны хорошие инструменты. Познакомлю вас со своими любимыми.

10.4. Инструменты, важные для автономного тестирования унаследованного кода

Ниже приводится несколько рекомендаций по поводу выбора инструментов, которые помогут приступить к тестированию существующего кода на платформе .NET.

- Изолируйте зависимости с помощью JustMock или Typemock Isolator.
- Используйте JMockit при работе с унаследованным кодом на Java.
- Используйте Vise для рефакторинга кода на Java.
- Используйте FitNesse для написания приемочных тестов перед началом рефакторинга.
- Прочитайте книгу Майкла Фэзерса об унаследованном коде.
- Используйте NDepend для исследования продуктового кода.
- Используйте ReSharper для навигации и рефакторинга продуктового кода.
- Используйте Simian и TeamCity для обнаружения повторяющегося кода (и ошибок).

Рассмотрим эти рекомендации более подробно.

10.4.1. Изолируйте зависимости с помощью JustMock или Typemock Isolator

Неограниченные каркасы типа Typemock Isolator описывались в главе 6. Уникальная особенность таких каркасов, делающая их пригодными для решения рассматриваемой задачи, – умение подделывать зависимости в имеющемся коде, вообще не подвергая его рефакторингу. Это экономит время для начальной подготовки к тестированию, которого всегда так не хватает.

Примечание. Полное раскрытие информации: во время подготовки первого издания этой книги, я трудился разработчиком в компании Typemock, занимаясь другим продуктом. Я также принимал участие в проектировании API Isolator 5.0. В декабре 2010 года я уволился из Typemock.

Почему Typemock, а не Microsoft Fakes?

Хотя каркас Microsoft Fakes бесплатен, а Isolator и JustMock – нет, я полагаю, что использование Microsoft Fakes приведет к появлению в проекте большого объема несопровождаемого кода, потому что дизайн и особенности использования (генерация кода и разбросанные повсюду делегаты) порождают очень хрупкий API, который трудно сопровождать. Эта проблема даже отмечена в документе группы ALM Rangers, посвященном Microsoft Fakes, который можно найти по адресу http://vsartesttoolingguide.codeplex.com/releases/view/102290. Там говорится: «если вы подвергнете рефакторингу тестируемый код, то автономные тесты, написанные с помощью прокладок (Shims) и заглушек (Stubs) из прежних сборок Fakes, компилироваться не будут. В настоящее время не существует простого

решения этой проблемы за исключением разве что модификации тестов с помощью специализированных регулярных выражений. Имейте это в виду, оценивая затраты на рефакторинг кода, для которого существует много автономных тестов. Затраты могут оказаться велики».

В следующих далее примерах я буду использовать Typemock Isolator, потому что работать с ним мне наиболее комфортно. В Isolator (на момент написания этой книги последней была версия 7.0) используется только термин *fake*, а слова *mock* и *stub* из API исключены. Этот каркас позволяет подделывать (fake) интерфейсы, запечатанные и статические типы, невиртуальные и статические методы. Это означает, что вам не нужно думать об изменении проекта (для этого может не оказаться времени или возможности ввиду соображений безопасности). Начинать тестирование можно почти сразу. Существует также бесплатная ограниченная версия Typemock, так что вы можете скачать продукт и поэкспериментировать с ним. Только помните, что, будучи ограниченной, эта версия работает лишь со стандартным тестопригодным кодом.

В листинге ниже приведены два примеры использования Isolator API для подделывания экземпляров классом.

Листинг 10.1. Подделывание статических методов и создание поддельных классов с помощью Isolator

```
[Test]
public void FakeAStaticMethod()
{
  Isolate
    .WhenCalled(()=>MyClass.SomeStaticMethod())
    .WillThrowException(new Exception());
}

[Test]
public void FakeAPrivateMethodOnAClassWithAPrivateConstructor()
{
  ClassWithPrivateConstructor c =
     Isolate.Fake.Instance<ClassWithPrivateConstructor>();
  Isolate.NonPublic.WhenCalled(c,"SomePrivateMethod").WillReturn(3);
}
```

Как видите, API прост и понятен, для возврата поддельных значений в нем используются универсальные методы и делегаты. Существует также API, рассчитанный специально на VB.NET, в котором синтаксис приближен к стилистике VB. Но в обоих случаях в дизайн тестируемых классов не нужно вносить никаких изменений.

10.4.2. Используйте JMockit при работе с унаследованным кодом на Java

JMockit и PowerMock – это проекты с открытым исходным кодом, в которых для реализации некоторых вещей, которые Typemock Isolator делает в .NET, используется Java Instrumentation API. Для изоляции компонентов от зависимостей в существующий проект не нужно вносить никаких изменений.

В JMockit используется подход на основе *подмены* (swap). Сначала вручную пишется класс, которые должен будет заменить класс, выступающей в роли зависимости для тестируемого компонента (допустим, класс `FakeDatabase` призван заменить класс `Database`). Затем с помощью JMockit мы подменяем обращения к исходному классу обращениями к нашему поддельному. Можно также переопределить методы класса, реализовав их в виде анонимных методов в тесте.

В следующем листинге приведен пример теста, в котором используется JMockit.

Листинг 10.2. Использование JMockit для подмены реализации класса

```
public class ServiceATest extends TestCase {
  private boolean serviceMethodCalled;

  public static class MockDatabase {
     static int findMethodCallCount;
     static int saveMethodCallCount;

     public static void save(Object o) {
        assertNotNull(o);
        saveMethodCallCount++;
     }

     public static List find(String ql, Object arg1) {
        assertNotNull(ql);
        assertNotNull(arg1);
        findMethodCallCount++;
        return Collections.EMPTY_LIST;
     }
  }

  protected void setUp() throws Exception {
     super.setUp();
     MockDatabase.findMethodCallCount = 0;
     MockDatabase.saveMethodCallCount = 0;
```

```
        Mockit.redefineMethods(Database.class,
             MockDatabase.class);
    }

    public void testDoBusinessOperationXyz() throws Exception {
        final BigDecimal total = new BigDecimal("125.40");

        Mockit.redefineMethods(ServiceB.class,
             new Object()
        {
            public BigDecimal computeTotal(List items)
            {
                assertNotNull(items);
                serviceMethodCalled = true;
                return total;
            }
        });

        EntityX data = new EntityX(5, "abc", "5453-1");
        new ServiceA().doBusinessOperationXyz(data);

        assertEquals(total, data.getTotal());
        assertTrue(serviceMethodCalled);
        assertEquals(1, MockDatabase.findMethodCallCount);
        assertEquals(1, MockDatabase.saveMethodCallCount);
    }
}
```

Все волшебство вершится здесь

JMockit – неплохая отправная точка для тестирования унаследованного кода на Java.

10.4.3. Используйте Vise для рефакторинга кода на Java

Майкл Фэзерс написал для Java интересный инструмент, который позволяет убедиться, что по ходу рефакторинга не повреждаются значения, которые могут изменяться в методе. Например, если метод изменяет массив значений, то можно проверить, что в процессе рефакторинга никакое значение случайно не затирается.

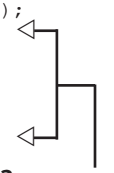
В листинге ниже показано, как для этой цели применяется метод `Vise.grip()`.

Листинг 10.3. Использование Vise в программе на Java для проверки того, что во время рефакторинга не изменились значения

```
import vise.tool.*;

public class RPRequest {
  ...
```

```
    public int process(int level, RPPacket packet) {
       if (...) {
          if (...) {
             ...
          } else {
             ...
             bar_args[1] += list.size();
             Vise.grip(bar_args[1]);
             packet.add(new Subpacket(list, arrivalTime));
             if (packet.calcSize() > 2)
               bar_args[1] += 2;
             Vise.grip(bar_args[1]);
          }
       } else {
          int reqLine = -1;
          bar_args[0] = packet.calcSize(reqLine);
          Vise.grip(bar_args[0]);
          ...
       }
    }
}
```



Примечание. Код в листинге 10.3 скопирован с разрешения владельца со страницы www.artima.com/weblogs/viewpost.jsp?thread=171323.

Vise заставляет добавлять строки в рабочий код, и весь его смысл – в поддержке рефакторинга. Для .NET такого инструмента не существует, но написать его сравнительно просто. При каждом вызове метод `Vise.grip()` проверяет, что переданное ему значение по-прежнему такое, каким должно быть. В некотором роде это добавление внутреннего утверждения с упрощенным синтаксисом. Vise может также выдать отчет о «закрепленных» (gripped) переменных и их текущих значениях.

Получить дополнительные сведения о Vise и загрузить его можно в блоге Майкла Фэзерса: www.artima.com/weblogs/viewpost.jsp?thread=171323.

10.4.4. Используйте приемочные тесты перед началом рефакторинга

Очень разумно будет добавить интеграционные тесты перед началом рефакторинга кода. FitNesse – один из таких инструментов, он позволяет создавать интеграционные и приемочные тесты. Из других подобных средств упомяну Cucumber и SpecFlow. (Для работы с Cucumber нужно хотя бы немного знать язык Ruby. SpecFlow ориентирован на .NET и предназначен для разбора сценариев, написанных

на Cucumber.) FitNesse позволяет писать интеграционные тесты приложения (на Java или .NET), а затем легко изменять их или добавлять новые без необходимости писать код.

Использование каркаса FitNesse состоит из трех шагов.

1. Написать классы-адаптеры (они называются фикстурами), которые обертывают продуктовый код и представляют действия пользователя, в которых этот код участвует. Например, в банковском приложении можно было бы написать класс `bankingAdapter` с методами `withdraw` и `deposit`.
2. Создать HTML-таблицы с помощью специального синтаксиса, который понимает и разбирает движок FitNesse. В этих таблицах хранятся значения, которые будут использоваться во время прогона тестов. Таблицы размещаются на страницах специального вики-сайта, в основе которого лежит движок FitNesse, поэтому ваш комплект тестов представляется внешнему миру через сайт. Каждую страницу, содержащую таблицу (которая видна в любом браузере), можно редактировать как обычную страницу вики, и на каждой имеется кнопка **Execute Tests** (Выполнить тесты). Таблицы разбираются во время выполнения и преобразуются в прогоны тестов.
3. Нажать кнопку **Execute Tests** на одной из вики-страниц. В результате движку FitNesse передаются параметры, заданные в таблице. В конечном итоге движок вызывает ваши классы-обертки, которые обращаются к целевому приложению, и высказывает утверждения о значениях, возвращенных этими классами

На рис. 10.5 показан пример таблицы FitNesse в браузере. Подробнее узнать о каркасе FitNesse можно на сайте http://fitnesse.org/. Об интеграции FitNesse с .NET написано на странице http://fitnesse.org/FitNesse.DotNet.

Лично я всегда считал, что использование FitNesse – сплошная головная боль: и интерфейс неудобный, и в половине случаев вообще не работает, особенно с кодом для .NET. Рекомендую вместо этого познакомиться с Cucumber на сайте http://cukes.info/.

10.4.5. Прочитайте книгу Майкла Фэзерса об унаследованном коде

Книга Майкла Фэзерса «Эффективная работа с унаследованным кодом» – единственный известный мне источник, где рассматривают-

ся вопросы работы с унаследованным кодом (помимо этой главы). В ней детально разобраны многочисленные приемы рефакторинга и подводные камни, о которых я даже не упоминаю. Я ценю ее на вес золота. Приобретите ее.

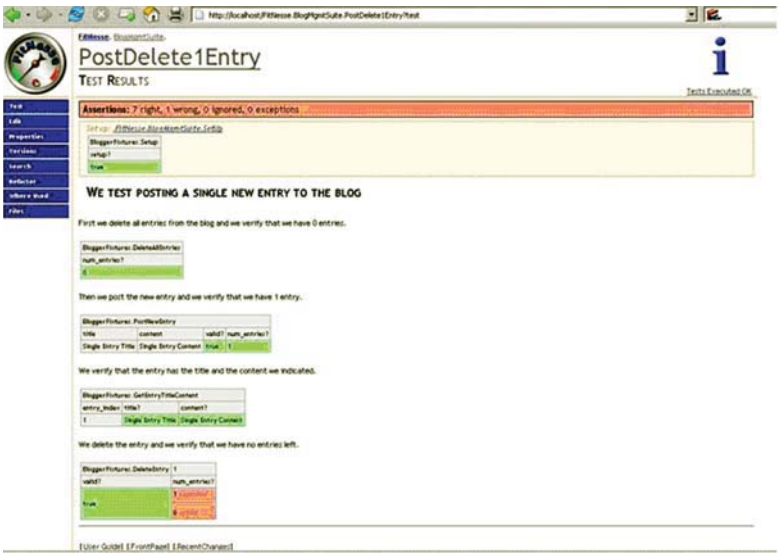


Рис. 10.5. Использование FitNesse для интеграционного тестирования

10.4.6. Используйте NDepend для исследования продуктового кода

NDepend – сравнительно новый коммерческий анализатор кода для .NET, который умеет создавать наглядные представления различных аспектов откомпилированных сборок, в том числе деревья зависимостей, сложность кода, отличия между разными версиями одной и той же сборки и многое другое. Потенциал у этого инструмента огромный, я рекомендую научиться им пользоваться.

Самое мощное средство NDepend – это специальный язык запросов (он называется CQL), на котором можно формулировать запросы о структуре кода, получая в ответ различные метрики компонентов. Например, нетрудно запросить все компоненты, в которых имеется закрытый конструктор.

Купить NDepend можно на сайте www.ndepend.com.

10.4.7. Используйте ReSharper для навигации и рефакторинга продуктового кода

ReSharper – одна из лучших надстроек над VS.NET для повышения продуктивности. Помимо мощных автоматизированных средств рефакторинга (гораздо мощнее встроенных в Visual Studio 2008) этот продукт знаменит своими навигационными возможностями. При работе с существующим проектом ReSharper может легко осуществлять навигацию по коду с помощью комбинаций клавиш для перехода из любой точки решения в любую другую, как-то связанную с исходной.

Приведем несколько примеров навигационных действий.

- Находясь в объявлении класса или метода, мы можем легко перейти к любому наследнику класса или переопределенному методу либо к реализации текущего члена в базовом классе, если таковой существует.
- Можно найти все места, где используется данная переменная (подсвечиваются в текущем редакторе).
- Можно найти все места, где используется общий интерфейс или реализующий его класс.

Есть еще много комбинаций клавиш, которые позволяют с меньшими усилиями осуществлять навигацию по коду и исследовать его структуру.

ReSharper работает для VB.NET и C#. Пробную версию можно скачать с сайта www.jetbrains.com.

10.4.8. Используйте Simian и TeamCity для обнаружения повторяющегося кода (и ошибок)

Допустим, вы нашли ошибку в своем коде и хотите удостовериться, что больше нигде она не повторяется.

В программу TeamCity встроен обнаружитель дубликатов для .NET. Дополнительные сведения о нем можно найти на странице http://confluence.jetbrains.com/display/TCD6/Duplicates+Finder+(.NET).

С помощью Simian легко найти повторяющийся код, узнать, сколько вам еще предстоит работы, а заодно произвести рефакторинг для

устранения дублирования. Simian – коммерческий продукт, имеющий версии для .NET, Java, C++ и других языков. Купить его можно на сайте www.harukizaemon.com/simian/.

10.5. Резюме

В этой главе я рассказал о том, как подступиться к унаследованному коду. Важно ранжировать компоненты по количеству зависимостей, сложности логики и приоритету для проекта. Имея эту информацию, вы сможете решить, с каких компонентов начать, исходя из легкости (или трудности) их подготовки к тестированию.

Если у команды мало или совсем нет опыта автономного тестирования, то лучше начинать с легких компонентов, тогда по мере добавления все новых и новых тестов уверенность команды в своих силах будет возрастать. Опытной команде рекомендуется начинать с трудных компонентов, тогда тестировать остальные части будет проще.

Если команда не хочет производить рефакторинг кода, добиваясь тестопригодности, а собирается просто написать автономные тесты того, что есть, то будут полезны неограниченные изолирующие каркасы, потому что они позволяют изолировать зависимости, не изменяя существующий код. Подумайте об этом, если имеете дело с унаследованным кодом для .NET. При работе с Java обратите внимание на JMockit или PowerMock – по тем же причинам.

Я также описал ряд инструментов, которые могут оказаться полезными в борьбе за улучшение качества существующего кода. Они предназначены для разных этапов проекта, а решать, какой использовать (и использовать ли вообще), предстоит вашей команде.

И напоследок, как сказал один мой приятель, имея дело с унаследованным кодом, никогда не вредно запастись изрядной бутылкой водочки.

ГЛАВА 11. Проектирование и тестопригодность

В этой главе:

- Преимущества учета тестопригодности при проектировании.
- Плюсы и минусы проектирования с учетом тестопригодности.
- Как быть с проектами, трудными для тестирования.

Перепроектирование программы с целью сделать ее более тестопригодной – больная тема для некоторых разработчиков. В этой главе мы рассмотрим основные идеи и методы проектирования с учетом тестопригодности. Мы также поговорим о плюсах и минусах этого подхода и о том, когда он имеет смысл.

Но сначала зададимся вопросом, зачем вообще нужно учитывать тестопригодность при проектировании.

11.1. Почему я должен думать о тестопригодности в своем проекте?

Законный вопрос. Проектируя программу, мы должны думать о том, что она должна делать и каких результатов может ожидать конечный пользователь, – так нас учили. Но тесты программы – еще один вид пользователя. Этот пользователь предъявляет строгие требования к программе, но все они вытекают из одного и того же пожелания:

тестопригодности. Это пожелание может изменять проект разными способами, по большей части – к лучшему.

В тестопригодном проекте каждая логическая конструкция (циклы, предложения `if`, `switch` и т. д.) должна быть устроена так, чтобы для нее можно было легко и быстро написать автономный тест, обладающий следующими свойствами:

- работает быстро;
- изолирован, т. е. может быть выполнен независимо или в составе группы тестов, до или после любого другого теста;
- не требует конфигурирования внешней системы;
- при любом прогоне дает один и тот же результат – прошел или не прошел.

Эту совокупность свойств иногда называют FICC: fast (быстрый), isolated (изолированный), configuration-free (без конфигурирования), consistent (стабильный). Если писать такой тест трудно или долго, система не может считаться тестопригодной.

Если представлять себе тесты как пользователя системы, то проектирование с учетом тестопригодности становится образом мыслей. Если бы вы разрабатывали систему через тестирование, то она автоматически получилась бы тестопригодной, потому что в этом случае тесты пишутся сначала и в значительной степени определяют структуру API системы, вынуждая делать ее такой, чтобы тесты могли с ней работать.

Теперь, понимая, что такое тестопригодное проектирование, посмотрим, что оно влечет за собой, обсудим его плюсы и минусы, рассмотрим альтернативы и приведем пример проекта, который трудно тестировать.

11.2. Цели проектирования с учетом тестопригодности

Существует несколько моментов, способных сделать проект существенно более тестопригодным. Роберт С. Мартин составил хороший список целей проектирования объектно-ориентированных систем, легший в основу проектов, рассматриваемых в этой главе. См. его статью «Principles of OOD» на странице http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

Мои рекомендации по большей части сводятся к включению в код зазоров – мест, куда можно будет внедрить другой код или подменить

поведение, не изменяя исходный класс. (О зазорах часто говорят в связи с принципом открытости-закрытости, который упоминается в статье «Principles of OOD».) Например, в методе, который обращается к веб-службе, API этой службы можно скрыть за интерфейсом и тем самым оставить возможность подменить настоящую веб-службу заглушкой, которая возвращает нужные нам значения, или подставным объектом. В главах 3–5 подделки, заглушки и подставки обсуждаются во всех деталях.

В табл. 11.1 приведены некоторые рекомендации по проектированию и вытекающие из них преимущества. Ниже мы обсудим эти рекомендации более подробно.

Таблица 11.1. Рекомендации по тестопригодному проектированию

Рекомендация	Преимущества
По умолчанию делайте методы виртуальными.	Это позволит переопределить методы в производном классе для целей тестирования. Переопределение дает возможность изменить поведение или разорвать внешнюю зависимость.
Проектируйте на основе интерфейсов.	Это позволит использовать полиморфизм для подмены зависимостей заглушками и подставками.
По умолчанию делайте классы незапечатанными.	Если класс запечатан (`final` в Java), то переопределить его виртуальные методы невозможно.
Избегайте создания экземпляров конкретных классов внутри методов, содержащих логику. Получайте экземпляры классов от вспомогательных методов, фабрик, DI-контейнеров типа Unity или иных мест, но не создавайте их напрямую.	Это позволит подсовывать поддельные экземпляры классов методам, которые в них нуждаются, не связывая себя внутренней реализацией порождения экземпляра.
Избегайте прямых обращений к статическим методам. Предпочитайте вызовы методов экземпляра, из которых уже вызываются статические методы.	Это позволит разорвать зависимость от статических методов путем переопределения методов экземпляра (переопределить статический метод невозможно).
Избегайте конструкторов и статических конструкторов, содержащих логику.	Реализовать переопределение конструкторов трудно. Чем проще конструктор, тем легче унаследовать классу в тестах.

Рекомендация	Преимущества
Отделяйте логику объектов-одиночек (синглтонов) от логики их создания	Если имеется одиночка, то должен быть способ подменить его экземпляр, чтобы можно было внедрить заглушку или вернуть объект в исходное состояние.

11.2.1. По умолчанию делайте методы виртуальными

В Java методы по умолчанию виртуальны, но на платформе .NET разработчики не такие везучие. В .NET, чтобы можно было переопределить поведение метода, необходимо явно сделать его виртуальным. В таком случае можно будет воспользоваться техникой «выделить и переопределить», описанной в главе 3.

Альтернативой этому способу является вызов пользовательского делегата из класса. Этот делегат можно будет подменить извне, установив свойство или передав параметр конструктору или методу. Это не самый распространенный подход, но некоторые системные проектировщики считают его приемлемым. В листинге ниже приведен пример класса с делегатом, который в тесте можно подменить.

Листинг 11.1. Класс, вызывающий делегат, который можно подменить в тесте

```
public class MyOverridableClass
{
  public Func<int,int> calculateMethod=delegate(int i)
  {
     return i*2;
  };

  public void DoSomeAction(int input)
  {
     int result = calculateMethod(input);
     if (result==-1)
     {
        throw new Exception("input was invalid");
     }
     // сделать что-то еще
  }
}

[Test]
[ExpectedException(typeof(Exception))]
public void DoSomething_GivenInvalidInput_ThrowsException()
{
```

```
    MyOverridableClass c = new MyOverridableClass();
    int SOME_NUMBER=1;

    // заглушить метод вычисления, возвращая "недопустимое" значение
    c.calculateMethod = delegate(int i) { return -1; };

    c.DoSomeAction(SOME_NUMBER);
}
```

Использовать виртуальные методы удобно, но проектирование на основе интерфейсов – не менее удачное решение, как поясняется в следующем разделе.

11.2.2. Проектируйте на основе интерфейсов

Выявление в приложении «ролей» и их абстрагирование в виде интерфейсов – важная часть процесса проектирования. Абстрактный класс не должен вызывать никакие конкретные классы, а конкретные классы могут вызывать только конкретные классы, являющиеся объектами данных (т. е. объекты, не имеющего никакого поведения, а служащие лишь для хранения данных). Это позволяет организовать в приложении многочисленные зазоры, куда можно подставить собственную реализацию.

Примеры такой подмены реализаций интерфейсов приведены в главах 3–5.

11.2.3. По умолчанию делайте классы незапечатанными

Некоторым программистам психологически трудно делать классы незапечатанными, потому что они предпочитают полностью управлять тем, кто и кому может наследовать в приложении. Беда в том, что если классу нельзя унаследовать, то не получится и переопределить его виртуальные методы.

Иногда этому правилу не удается следовать из соображений безопасности, но вообще его соблюдение должно стать правилом, а не исключением.

11.2.4. Избегайте создания экземпляров конкретных классов внутри методов, содержащих логику

Иногда психологически трудно избежать создания экземпляров конкретных классов внутри методов, содержащих логику, просто потому

что вы к этому привыкли. Причина же в том, что тестам может понадобиться управлять тем, какой экземпляр создается в тестируемом классе. Если не существует зазора, из которого возвращается экземпляр, то задача существенно усложняется – приходится использовать неограниченные изолирующие каркасы типа Typemock Isolator. Например, если методу нужен объект протоколирования, не создавайте его внутри метода. Получите объект от простого фабричного метода и сделайте этот метод виртуальным, чтобы впоследствии его можно было переопределить и таким образом предоставить методу то, что удобно тесту. Или же воспользуйтесь не виртуальным методом, а инверсией зависимости через конструктор. Эти и другие способы внедрения обсуждаются в главе 3.

11.2.5. Избегайте прямых обращений к статическим методам

Попытайтесь абстрагировать прямые зависимости, которые было бы трудно подменить во время выполнения. В большинстве статических языков, в частности в VB.NET и C#, подменить поведение статического метода трудно или громоздко. Абстрагирование статических методов с помощью техники «выделить и переопределить» (см. раздел 3.4 в главе 3) – один из способов решения проблемы.

Более радикальный подход – вообще избегать статических методов. Тогда любая логика является частью экземпляра класса, и подменить ее становится проще. Невозможность подмены – одна из причин, почему некоторые программисты, практикующие автономное тестирование или TDD, не любят объекты-одиночки – они выступают в роли открытого разделяемого ресурса, по природе своей статического, который трудно переопределить.

Полностью отказаться от статического методов тоже трудно, но, попытавшись свести к минимуму число одиночек и статических методов в приложении, вы упростите себе жизнь во время тестирования.

11.2.6. Избегайте конструкторов и статических конструкторов, содержащих логику

Классы, предназначенные для конфигурирования, часто делаются статическими или одиночками, потому что они используются в самых разных частях приложения. Поэтому подменить такие классы

во время тестирования трудно. Один из способов решения этой проблемы – воспользоваться той или иной формой контейнеров инверсии управления (IoC) (Microsoft Unity, Autofac, Ninject, StructureMap, Spring.NET или Castle Windsor – все это каркасы для .NET с открытым исходным кодом).

Эти контейнеры много чего умеют, но все они предоставляют общую умную фабрику, которая позволяет получать экземпляры классов, не зная, является ли экземпляр объектом-одиночкой и как он реализован. Вы просите интерфейс (обычно в конструкторе) и автоматически получаете объект, который этот интерфейс реализует.

Использование IoC-контейнера (их еще называют DI-контейнерами) позволяет абстрагировать управление временем жизни объекта и упрощает создание объектной модели, основанной главным образом на интерфейсах, поскольку контейнер автоматически разрешает все зависимости класса.

Подробное обсуждение контейнеров выходит за рамки этой книги, но их полный перечень и некоторые отправные точки можно найти в статье «List of .NET Dependency Injection Containers (IOC)» в блоге Скотта Ханселмана (Scott Hanselman): http://www.hanselman.com/blog/ListOfNETDependencyInjection ContainersIOC.aspx.

11.2.7. Отделяйте логику объектов-одиночек от логики их создания

Планируя использовать в проекте класс-одиночку, отделяйте логику самого класса от логики создания его экземпляра (например, той части, где инициализируется статическая переменная), помещая то и другое в разные классы. Тогда вы, с одной стороны, не отступите от принципа единственной обязанности (SRP), а, с другой, оставите способ переопределить логику одиночки.

Например, в листинге 11.2 показан класс-одиночка, а в листинге 11.3 – результат его рефакторинга для повышения тестопригодности.

Листинг 11.2. Нетестопригодный класс-одиночка

```
public class MySingleton
{
  private static MySingleton _instance;

  public static MySingleton Instance
  {
     get
```

```
        {
            if (_instance == null)
            {
                _instance = new MySingleton();
            }
            return _instance;
        }
    }
}
```

Листинг 11.3. Тестопригодный класс-одиночка после рефакторинга

```
public class RealSingletonLogic                    <-- Тестопригодная логика
{
  public void Foo()
  {
     // сложная логика
  }
}

public class MySingletonHolder                     <-- Контейнер одиночки
{
  private static RealSingletonLogic _instance;

  public static RealSingletonLogic Instance
  {
     get
     {
        if (_instance == null)
        {
           _instance = new RealSingletonLogic();
        }
        return _instance;
     }
  }
}
```

Рассмотрев некоторые способы обеспечения тестопригодности проекта, вернемся к общей картине. А надо ли все это делать, нет ли каких-нибудь негативных последствий?

11.3. Плюсы и минусы проектирования с учетом тестопригодности

Проектирование с учетом тестопригодности – для многих больной вопрос. Одни считают, что тестопригодность должна быть одним из

обязательных свойств проекта, другие – что проект не должен «страдать» от того, что кому-то взбредет в голову его тестировать.

Важно понять, что сама по себе тестопригодность – не конечная цель, а лишь побочный продукт определенной школы мысли, которая практикует принципы тестопригодного объектно-ориентированного проектирования, изложенные Робертом С. Мартином (и упомянутые в начале раздела 11.2). В проекте, где расширяемость и абстракции классов занимают одно из первых мест, легко найти зазоры для действий, относящихся к тестированию. Все рассмотренные в этой главе приемы прекрасно согласованы с принципами Мартина: классы, чье поведение можно изменять путем наследования и переопределения или путем внедрения интерфейса, являются «открытыми для расширения, но закрытыми для модификации» – это принцип открытости-закрытости. Такие классы обычно отвечают также принципам внедрения зависимости и инверсии управления, чтобы поддержать внедрение через конструктор. Применяя принцип единственной обязанности, мы можем, например, отделить сам класс-одиночку от логики его создания. И лишь принцип подстановки Лисков одиноко стоит в сторонке, потому что я не смог придумать ни одного примера, где его нарушение нарушает также тестопригодность. Но из того факта, что ваш тестопригодный проект по видимости коррелирует с принципами SOLID, еще *не* следует, что этот проект хорош или что вы в совершенстве овладели искусством проектирования. Вовсе нет. Скорее всего, ваш проект, как и мой, можно улучшить. Возьмите какую-нибудь достойную книгу по этому предмету, например, Eric Evans «Domain-Driven Design: Tackling Complexity in the Heart of Software»[1] (Addison-Wesley Professional, 2003) или Joshua Kerievsky «Refactoring to Patterns»[2] (Addison-Wesley Professional, 2004). А как насчет «Чистого кода» Роберта Мартина? Тоже годится!

Я встречал чертову уйму ужасно спроектированного и в высшей степени тестопригодного кода. Лишнее доказательство того, что TDD без должных знаний о проектировании не обязательно приносит хорошие плоды.

Вопрос, тем не менее, остается – верно ли, что это лучший способ работы? Каковы недостатки методики тестопригодного проектирования? Как быть, если код унаследован? И так далее.

---

[1] Эрик Эванс «Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем», Вильямс, 2010. – *Прим. перев.*

[2] Джошуа Кериевски «Рефакторинг с использованием шаблонов», Вильямс, 2006. – *Прим. перев.*

11.3.1. Объем работы

В большинстве случаев проектирование с учетом тестопригодности занимает больше времени, потому что обычно при этом приходится писать больше кода. Даже дядюшка Боб в своих пространных и временами забавных видеороликах на сайте http://cleancoders.com любит говаривать (голосом Шерлока Холмса, держа в руке трубку), что он начинает с упрощенного дизайна, который делает нечто простейшим из возможных способов, а рефакторингом занимается, только когда видит в этом необходимость.

Вы можете возразить, что дополнительное проектирование, требуемое для обеспечения тестопригодности, высвечивает проблемы, которые вы раньше не рассматривали и которые в любом случае стоило бы учесть в проекте (разделение функций, принцип единственной обязанности и т. д.).

С другой стороны, в предположении что вы и так довольны своим проектом, вносить изменения, не относящиеся к функциональности продукта, только ради тестопригодности может оказаться проблематично. Но и тут можно возразить, что код тестов важен не меньше, чем продуктовый, так как раскрывает характеристики API, описывающего модель предметной области, и заставляет думать о том, как другие люди будут использовать ваш код.

Начиная с этого места, дискуссии на эту тему редко оказываются продуктивными. Остановимся на том, что для обеспечения тестопригодности действительно требуется больше кода и больше работы, но зато при этом мы вынуждены задумываться о пользователях своего API – и это хорошо.

11.3.2. Сложность

При проектировании с учетом тестопригодности иногда возникает ощущение, что вы чуточку (а, может, вовсе и не чуточку) переусложняете. То ловите себя на добавлении интерфейсов туда, где им вроде бы не место, то раскрывается семантика поведения класса, хотя вы этого вовсе не хотели. А когда абстрагируется слишком много интерфейсов, искать в коде, где же все-таки находится истинная реализация метода, становится мукой мученической.

Можно было бы возразить, что есть инструменты типа ReSharper, которые значительно упрощают навигацию, и потому этот аргумент уже не актуален. Согласен, что большинство проблем навигации

они решают. Наличие подходящего для работы инструмента вообще очень помогает.

11.3.3. Раскрытие секретной интеллектуальной собственности

Во многих проектах воплощена интеллектуальная собственность, которая должна оставаться секретной, но при проектировании с учетом тестопригодности что-то неизбежно раскрывается, например, информация о защите или лицензировании или запатентованные алгоритмы. Есть, конечно, обходные пути – сделать соответствующие части программы внутренними и воспользоваться атрибутом `[InternalsVisibleTo]`, но они по сути своей подрывают саму идею тестопригодного проектирования. Мы изменяем проект, но все равно сохраняем часть логики скрытой. И зачем тогда было все затевать?

В этом отношении идея проектирования с учетом тестопригодности действительно дает небольшой сбой. Иногда обойти проблемы, связанные с безопасностью или патентной защитой, не получается. Тогда нужно что-то изменить или пойти на компромисс.

11.3.4. Иногда нет никакой возможности

Иногда существуют политические или иные причины, навязывающие определенный способ проектирования, и поделать с этим ничего нельзя (есть кто из «Выматываем душу, Инк.»?). Бывает и так, что просто нет времени на переработку проекта или проект настолько хрупок, что перерабатывать его боязно. Это еще один случай, когда проектирование с учетом тестопригодности не годится – мешает окружение. Тут мы имеем пример факторов влияния, которые рассматривались в главе 9.

Рассмотрев аргументы за и против, обратимся к альтернативам тестопригодному проектированию.

11.4. Альтернативы проектированию с учетом тестопригодности

Интересно посмотреть, как решается проблема в других языках.

Код, написанный на динамических языках типа Ruby или Smalltalk, изначально тестопригоден, потому что во время выполнения можно

подменить все, что угодно. Работая с таким языком, можно проектировать, как вам нравится, не задумываясь о тестопригодности. Чтобы подменить что-то, не нужно заводить интерфейс, а допускающий переопределение метод не обязан быть открытым. Можно даже динамически изменять поведение базовых типов, и никто не станет ругаться, что код не компилируется.

В мире, где все тестопригодно, нужно ли учитывать это при проектировании? Вы, конечно, ожидаете ответа «нет». Мол, в таком мире мы вольны выбирать проект, сообразуясь с собственным вкусом.

11.4.1. К вопросу о проектировании в динамически типизированных языках

Интересно, однако, что, начиная с 2010 года, в сообществе Ruby, к которому я тоже когда-то принадлежал, все чаще ведутся разговоры о принципах проектирования SOLID (единственной обязанности, открытости-закрытости, подстановки Лисков, разделения интерфейсов и инверсии зависимости). «Могу – не значит должен», говорят некоторые рубисты, например Авди Гримм (Avdi Grimm), автор книги «Objects on Rails», выложенной на сайте http://objectsonrails.com. В блогах можно найти немало размышлений о состоянии дел с проектированием в сообществе Rails, например, на странице http://jamesgolick.com/2012/5/22/objectify-a-better-way-to-build-rails-applications.html. Другие рубисты на это отвечают: «Отстаньте от нас со своим переинженирингом». Так, Дэвид Хейнемейер Ханссон (David Heinemeier Hansson), он же DHH, создатель каркаса Ruby on Rails, обсуждает эту тему в своей статье «Dependency injection is not a virtue» (Внедрение зависимости – не добродетель) по адресу http://david.heinemeierhansson.com/2012/dependency-injection-is-not-avirtue.html.

Можете представить, какое оживление это вызывает в Твиттере.

Забавно, что все эти споры напоминают мне то, что происходило в 2008–2009 годах в сообществе .NET, а особенно в рядах недавно почившего движения ALT.NET (большая часть активистов ALT.NET открыла для себя Ruby или Node.js и отдалилась от .NET, правда, через год вернулась – «ради денег». Каюсь, сам виновен!). Но существенная разница состоит в том, что мы говорим о Ruby. В сообществе .NET хотя бы были пусть и не вполне убедительные аргументы в пользу сторонников лозунга «За SOLID-ное проектирование»: например, невозможно протестировать проект без классов, отвечающих принципу

открытости-закрытости, потому что компилятор нарявкает. Так что проектировщики говорили: «Ну видишь? Даже компилятор говорит, что твой дизайн – отстой». В ретроспективе это звучит довольно глупо, потому что есть сколько угодно тестопригодных проектов, являющихся, тем не менее, в высшей степени отстойными. А теперь приходят ребята, пишущие на Ruby, и говорят, что хотят использовать принципы SOLID? На хрена им это нужно?

Похоже, использование SOLID сулит кое-какие дополнительные выгоды: код становится проще понять и сопровождать, а в мире Ruby это может оказаться очень серьезной проблемой. Иногда даже более серьезной, чем в статически типизированных языках, потому что в Ruby динамическая программа может вызывать глубоко скрытый переадресованный код, и тогда сам черт ногу сломит. Тесты помогают, но лишь до определенного предела.

Ну, а что я думал на этот счет? Сложилось так, что поначалу программисты, пишущие на Ruby, вообще не задумывались о тестопригодности дизайна, потому что код и так уже был тестопригодным. И все было хорошо, пока они не открыли для себя идеи *проектирования* кода; оказалось, что *проектирование* – самостоятельная деятельность, которая отнюдь не сводится к простому рефакторингу, связанному с тестопригодностью.

Но вернемся к .NET и к статически типизированным языкам: возьмем какую-нибудь технологию на платформе .NET, которая показывает, как использование инструментов может изменить способ осмысления задач и иногда сделать трудную проблему тривиальной. В мире, где памятью управляют за вас, нужно ли учитывать в проекте управление памятью? Как правило, нет. При работе с языком, в котором автоматического управления памятью нет (например, C++), необходимо помнить об оптимизации и своевременном освобождении памяти, иначе приложению будет худо. Но это не помешает хорошо спроектировать программу, не в управлении памятью дело. Важны совсем другие вещи: удобочитаемость кода, удобство пользования и т. д. Вы не прибегаете к воображаемому противнику, аргументируя выбор проектных решений, потому что, обосновывая свою позицию, возможно, опираетесь не на ту подпорку (слишком много метафор? Да-да, я знаю. Это как... да нет, ничего).

Точно так же, следуя принципам объектно-ориентированного проектирования, обеспечивающим тестопригодность, вы можете получить тестопригодный проект в качестве побочного продукта, хотя цели такой и не ставили. Проект предназначается для решения кон-

кретной задачи. Если под рукой имеется инструмент, который решает за вас проблему тестопригодности, то специально учитывать этот аспект при проектировании нет нужды. У проектов есть и другие достоинства, но они должны быть результатом осознанного выбора, а не просто данностью.

Основная проблема нетестопригодного проекта состоит в невозможности подменить зависимости во время выполнения. Именно поэтому мы создаем интерфейсы, делаем методы виртуальными и занимаемся другими подобными вещами. На платформе .NET существуют инструменты, которые умеют подменять зависимости без рефакторинга с целью обеспечения тестопригодности, – неограниченные изолирующие каркасы.

Означает ли наличие неограниченных каркасов, что учитывать тестопригодность при проектировании не нужно? В каком-то смысле да. Это избавляет от необходимости включать тестопригодность в состав целей проектирования. У объектно-ориентированных принципов, выдвинутых Бобом Мартином, есть замечательные достоинства, и следовать им нужно не потому, что они обеспечивают тестопригодность, а потому что в применении к проектированию они разумны и целесообразны. Они упрощают разработку, чтение и сопровождение кода, даже если тестопригодность не ставится во главу угла.

Завершим это обсуждение примером проекта, который с трудом поддается тестированию.

11.5. Пример проекта, трудного для тестирования

Легко найти интересные проекты, в которых можно покопаться. Один такой проект – BlogEngine.NET, исходный код которого находится по адресу http://blogengine.codeplex.com/SourceControl/latest. Можно сразу определить, использовалась ли при создании проекта методика разработки через тестирование и задумывались ли авторы о тестопригодности. В данном случае по всему коду разбросаны статические классы, статические методы, статические конструкторы. С точки зрения проектирования, это не так уж плохо. Но напомню, эта книга не о проектировании. А вот если говорить о тестопригодности, то такой подход неудачен.

Взгляните всего на один класс из решения: класс Manager в пространстве имен Ping (http://blogengine.codeplex.com/SourceControl/latest#BlogEngine/BlogEngine. Core/Ping/Manager.cs):

```
namespace BlogEngine.Core.Ping
{
  using System;
  using System.Collections.Generic;
  using System.Linq;
  using System.Text.RegularExpressions;

  public static class Manager
  {
     private static readonly Regex TrackbackLinkRegex = new Regex(
        "trackback:ping=\"([^\"]+)\"", RegexOptions.IgnoreCase |
                                        RegexOptions.Compiled);
     private static readonly Regex UrlsRegex = new Regex(
        @"<a.*?href=[""'](?<url>.*?)[""'].*?>(?<name>.*?)</a>",
        RegexOptions.IgnoreCase | RegexOptions.Compiled);

     public static void Send(IPublishable item, Uri itemUrl)
     {
        foreach (var url in GetUrlsFromContent(item.Content))
        {
           var trackbackSent = false;
           if (BlogSettings.Instance.EnableTrackBackSend)
           {
            // ignoreRemoteDownloadSettings должно быть равно true
            // для обратной совместимости с Utils.DownloadWebPage.
            var remoteFile = new RemoteFile(url, true);
            var pageContent = remoteFile.GetFileAsString();
            var trackbackUrl = GetTrackBackUrlFromPage(pageContent);

            if (trackbackUrl != null)
            {
              var message =
                new TrackbackMessage(item, trackbackUrl, itemUrl);
              trackbackSent = Trackback.Send(message);
            }
           }
           if (!trackbackSent &&
               BlogSettings.Instance.EnablePingBackSend)
           {
            Pingback.Send(itemUrl, url);
           }
        }
     }

     private static Uri GetTrackBackUrlFromPage(string input)
     {
        var url =
          TrackbackLinkRegex.Match(input).Groups[1].ToString().Trim();
        Uri uri;
```

```
            return
            Uri.TryCreate(url, UriKind.Absolute, out uri) ? uri : null;
        }

        private static IEnumerable<Uri> GetUrlsFromContent(string content)
        {
            var urlsList = new List<Uri>();
            foreach (var url in
                UrlsRegex.Matches(content).Cast<Match>().Select(myMatch =>
                  myMatch.Groups["url"].ToString().Trim()))
            {
                Uri uri;
                if (Uri.TryCreate(url, UriKind.Absolute, out uri))
                {
                  urlsList.Add(uri);
                }
            }
            return urlsList;
        }
    }
}
```

Мы сосредоточимся на методе `Send` класса `Manager`. Предполагается, что этот метод посылает какой-то сигнал прозвона или трассировки (что это такое, нам сейчас не важно), если обнаруживает в посте пользователя какие-то из перечисленных URL-адресов.

В этом методе реализовано много требований:

- посылать отклик, только если в глобальном объекте конфигурации некоторое свойство равно `true`;
- если сигнал прозвона не послан, попробовать послать сигнал трассировки;
- посылать сигнал прозвона или трассировки для любого URL-адреса, обнаруженного в тексте поста.

Почему я считаю, что этот метод трудно тестировать? По нескольким причинам.

- Все зависимости (например, от конфигурации) – статические методы, поэтому без неограниченного каркаса их трудно подделать и подменить.
- Даже если бы удалось подделать зависимости, их невозможно внедрить через параметры или свойства. Они используются непосредственно.
- Можно было бы попробовать воспользоваться техникой «выделить и переопределить» (см. главу 3), чтобы вызывать зависимости с помощью виртуальных методов, которые перео-

пределяются в производном классе, только вот класс `Manager` статический, поэтому никаких нестатических, а уж тем более виртуальных методов в нем быть не может. Так что выделить и переопределить не удастся.

- Даже если бы этот класс не был статическим, статическим является метод, который мы хотим протестировать. А чтобы можно было применить технику «выделить и переопределить», нам нужен метод экземпляра. Опять облом.

Вот как я подошел бы к рефакторингу этого класса (в предположении, что имеются интеграционные тесты).

1. Сделать класс нестатическим.
2. Создать нестатическую копию метода `Send()` с теми же самыми параметрами. Я бы назвал ее `InstanceSend()`, чтобы при компиляции не возникало конфликтов с исходным статическим методом.
3. Удалить весь код из исходного статического метода, заменив его на `new Manager().InstanceSend(item, itemUrl);`, так что статический метод станет просто механизмом перенаправления. При этом прежний код, из которого этот метод вызывается, не сломается (это же рефакторинг!).
4. Имея нестатический класс и метод экземпляра в нем, я могу применить технику «выделить и переопределить» к частям метода `InstanceSend()` и тем самым разорвать зависимости, например, выделить обращение к `BlogSettings.Instance.EnableTrackBackSend` в отдельный виртуальный метод, который затем можно будет переопределить в классе, производном от `Manager`.
5. Я еще не закончил, но просвет уже виден. Теперь можно продолжить рефакторинг и выделение с переопределением.

Вот как выглядит класс, перед тем как я начну выделять и переопределять.

```
public static class Manager
{
  …
  public static void Send(IPublishable item, Uri itemUrl)
  {
    new Manager().InstanceSend(item,itemUrl);
  }

  public static void InstanceSend(IPublishable item, Uri itemUrl)
  {
```

```
        foreach (var url in GetUrlsFromContent(item.Content))
        {
            var trackbackSent = false;
            if (BlogSettings.Instance.EnableTrackBackSend)
            {
                // ignoreRemoteDownloadSettings должно быть равно true
                // для обратной совместимости с Utils.DownloadWebPage.
                var remoteFile = new RemoteFile(url, true);
                var pageContent = remoteFile.GetFileAsString();
                var trackbackUrl = GetTrackBackUrlFromPage(pageContent);

                if (trackbackUrl != null)
                {
                    var message =
                      new TrackbackMessage(item, trackbackUrl, itemUrl);
                    trackbackSent = Trackback.Send(message);
                }
            }
            if (!trackbackSent && BlogSettings.Instance.EnablePingBackSend)
            {
                Pingback.Send(itemUrl, url);
            }
        }
    }

    private static Uri GetTrackBackUrlFromPage(string input)
    {
        …
    }

    private static IEnumerable<Uri> GetUrlsFromContent(string content)
    {
        …
    }
}
```

Вот несколько вещей, которые помогли бы сделать этот класс более тестопригодным:

- По умолчанию классы должны быть нестатическими. В C# вообще мало причин для создания чисто статических классов.
- Использовать методы экземпляра, а не статические методы.

На сайте http://tddcourse.osherove.com имеется онлайновый курс по TDD, в котором показано, как я провожу рефакторинг этого класса.

11.6. Резюме

В этой главе мы обсуждали идею проектирования с учетом тестопригодности: что это означает с точки зрения приемов проектирования,

плюсы и минусы и альтернативы. Простых ответов тут не существует, но вопросы интересные. Будущее автономного тестирования зависит от того, как люди станут решать эти проблемы и какие инструменты появятся в качестве альтернатив.

Тестопригодность проекта обычно существенна только в статических языках типа C# и VB.NET, где приходится заранее думать, как можно будет подменить различные объекты. В динамических языках, которые изначально гораздо более тестопригодны, учет тестопригодности на этапе проектирования играет куда меньшую роль. В таких языках почти все можно подменить вне зависимости от исходных проектных решений. Это избавляет соответствующие сообщества от необходимости спорить с воображаемым противником о том, означает ли нетестопригодность программы, что она плохо спроектирована, и сосредоточиться на более глубоких аспектах хорошего проектирования.

В тестопригодном проекте имеются виртуальные методы, незапечатанные классы, интерфейсы и четкое разделение функций. В нем меньше статических классов и методов и больше экземпляров классов с логикой. На самом деле, тестопригодный проект следует принципам SOLID, но это еще не значит, что проект обязательно хороший. Быть может, пришло время сказать, что конечной целью должна быть не тестопригодность, а исключительно хороший дизайн.

Мы рассмотрели небольшой пример совершенно нетестопригодного проекта и шаги, которые нужно выполнить, что привести его к тестопригодному виду. Подумайте, настолько легко было бы его тестировать, если бы при разработке применялась методика TDD! Он был бы тестопригоден с самой первой строчки кода, и нам не пришлось бы устраивать эти танцы с бубнами.

Мне пора закругляться. Но мир прекрасен и полон вещей, в которые, я уверен, вы с удовольствием запустили бы зубки.

11.7. Дополнительные ресурсы

Я знаю, что многие читатели этой книги проходят следующие стадии.

- Освоившись с соглашениями об именовании, они начинают перенимать другие или создавать свои собственные. Это хорошо. Мои соглашения хороши для начинающих, а сам я пользуюсь ими до сих пор, но это не единственный путь. Выбирайте имена тестов, так чтобы вам было комфортно.

- Начинают искать другие формы написания тестов, например каркасы в стиле разработки через поведение (BDD) типа MSpec или NSpec. Это тоже хорошо, потому что коль скоро сохранены три важных аспекта (что тестируется, при каких условиях и каков ожидаемый результат), удобочитаемость не страдает. В API, построенных на основе BDD, проще организовать единственную точку входа и высказывать утверждения о нескольких конечных результатах в разных требованиях – получается в высшей степени понятно. Объясняется это тем, что большинство API на основе BDD допускают иерархическое написание.
- Начинают автоматизировать больше интеграционных и комплексных тестов, потому что находят, что уровень автономного тестирования слишком низок. И это замечательно, так как хорошо все, что позволяет вам уверенно вносить изменения в код. Если в проекте вообще не останется автономных тестов, но вы тем не менее сохраните способность вести разработку с высокой скоростью и качеством, это просто отлично, и хорошо бы мне у вас поучиться. (Это возможно, но, начиная с какого-то момента, прогон тестов начинает занимать уж очень много времени. Мы пока еще не нашли волшебного способа получить все и сразу.)

Как насчет книг?

В части проектирования эту книгу прекрасно дополняет книга Steve Freeman, Nat Pryce «Growing Object-Oriented Software, Guided by Tests».

Хорошим справочником по паттернам и антипаттернам автономного тестирования может служить книга Gerard Meszaros «xUnit Test Patterns: Refactoring Test Code»[3].

Книга Michael Feathers «Working Effectively with Legacy Code»[4] обязательна для чтения, если вам приходится сталкиваться с унаследованным кодом.

На сайте ArtOfUnitTesting.com приведен более полный и постоянно (дважды в год) обновляемый список интересных книг.

Что касается анализа тестов, рекомендую также посмотреть мои видео, в которых я рассказываю о том, как можно улучшить проек-

[3] Джерард Месарош «Шаблоны тестирования xUnit. Рефакторинг кода тестов», Вильямс, 2009. – *Прим. перев.*

[4] Майкл Физерс «Эффективная работа с унаследованным кодом», Вильямс, 2009. – *Прим. перев.*

ты с открытым исходным кодом. Они выложены по адресу http://artofunittesting.com/test-reviews/.

Я разместил на сайтах http://ArtOfUnitTesting.com и http://Osherove.com/Videos еще немало бесплатных видео, анализов тестов, сеансов парного программирования и докладов на конференциях по разработке через тестирование. Надеюсь, эта дополнительная информация будет вам полезна.

Возможно, вы захотите записаться на мой мастер-класс по TDD (доступен также в формате онлайнового потокового видео) на сайте http://TDDCourse.Osherove.com.

Вы всегда можете связаться со мной через Твиттер (@RoyOsherove) или напрямую по адресу http://Contact.Osherove.com.

С нетерпением жду весточки от вас!

ПРИЛОЖЕНИЕ. Инструменты и каркасы

Эта книга была бы неполной без обзора некоторых инструментов и основных приемов написания автономных тестов. В приложении описываются инструменты, подходящие для самых разных видов тестирования: работы с базами данных, пользовательского интерфейса, веб-приложений. Одни используются для интеграционного тестирования, другие – и для автономного тоже. Особо я отмечаю те, что, на мой взгляд, хороши для начинающих.

Инструменты и методики разбиты на следующие категории:

- изолирующие каркасы;
- каркасы тестирования;
  - исполнители тестов;
  - API тестирования;
- вспомогательные средства тестирования;
- DI и IoC-контейнеры;
- тестирование работы с базами данных;
- тестирование веб-приложений;
- тестирование пользовательского интерфейса;
- тестирование многопоточных приложений;
- приемочное тестирование.

Совет. Актуальный вариант списка инструментов и методик можно найти на сайте этой книги http://ArtOfUnitTesting.com.

Итак, приступим.

A.1. Изолирующие каркасы

Изолирующие каркасы, или каркасы подставных объектов – это плоть и кровь автономного тестирования в достаточно сложных случаях. Выбирать есть из чего, и это прекрасно:

- Moq

- Rhino Mocks
- Typemock Isolator
- JustMock
- Moles/Microsoft Fakes
- NSubstitute
- FakeItEasy
- Foq

В предыдущем издании этой книги описывались еще два инструмента, которые я удалил, поскольку они устарели:

- NMock
- NUnit.Mocks

Ниже приводятся краткие описания каждого каркаса.

A.1.1. Moq

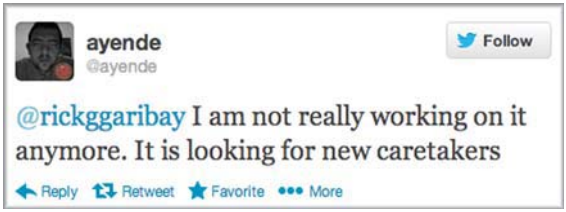
Moq – изолирующий каркас с открытым исходным кодом и API, претендующим на простоту изучения и использования. Это один из первых API, построенных в стиле подготовка–действие–утверждение (в противоположность модели запись–воспроизведение, применявшейся в более ранних каркасах) и опирающихся на возможности, появившиеся в .NET 3.5 и 4, в частности лямбда-выражения и методы расширения. Чтобы им пользоваться, необходимо уверенно владеть синтаксисом лямбда-выражений. Впрочем, это относится и ко всем остальным каркасам из списка.

Осваивается каркас легко. Могу лишь придраться к чрезмерному использованию слова *mock* в API, что только вносит путаницу. Я хотел бы, по крайней мере, чтобы было проведено различие между созданием заглушек или подставок. А еще лучше, чтобы в обоих случаях употреблялось слово *fake*, что устранило бы сам источник путаницы.

О Moq можно прочитать на сайте http://code.google.com/p/moq/ и установить пакет через NuGet.

A.1.2. Rhino Mocks

Rhino Mocks – широко используемый открытый каркас для создания заглушек и подставок. В предыдущем издании я его рекомендовал, сейчас не рекомендую. Его разработка практически прекращена, к тому же существуют более легкие, простые и удачно спроектированные каркасы. Если есть возможность, не пользуйтесь им. Автор, Айенде (Ayende), написал в Твиттере, что больше его не поддерживает.



Скачать Rhino Mocks можно со страницы http://ayende.com/projects/rhino-mocks.aspx.

A.1.3. Typemock Isolator

Typemock Isolator – коммерческий *неограниченный* (умеет подделывать все, см. главу 6) изолирующий каркас, в котором сделана попытка убрать из лексикона термины *заглушки* и *подставки* в пользу более простого и лаконичного API.

Isolator отличается от других каркасов тем, что позволяет изолировать компоненты от зависимостей, как бы ни была спроектирована система (хотя при этом поддерживает все функции, имеющиеся в других каркасах). Это делает его идеальным выбором для людей, которые только приступают к автономному тестированию и хотят изучать вопросы проектирования и тестопригодности постепенно. Поскольку каркас не заставляет учитывать тестопригодность при проектировании, вы можете сначала научиться правильно писать тесты, а потом заняться повышением квалификации в части проектирования, а не смешивать все сразу. Но это и самое дорогое из неограниченных решений, что, правда, окупается удобством работы и возможностью использования для унаследованного кода.

Есть два варианта Typemock Isolator: бесплатная базовая редакция со всеми недостатками, присущими ограниченному каркасу (возможность подделывания только нестатических виртуальных методов и т. д.), и неограниченная платная редакция, умеющая подделывать почти все.

Примечание. Полное раскрытие информации: я работал в Typemock в период между 2008 и 2010 годом.

Скачать Typemock Isolator можно на сайте http://www.typemock.com.

A.1.4. JustMock

JustMock от компании Telerik – сравнительно новый изолирующий каркас, явно нацеленный на конкуренцию с Typemock Isolator. API обоих каркасов похожи по структуре, так что при выполнении простых действий можно сравнительно легко переключаться между ними. Как и Typemock, JustMock поставляется в двух вариантах: ограниченная бесплатная редакция и неограниченная платная, способная подделывать почти все.

В API есть небольшие шероховатости, и, насколько я знаю, пока не поддерживаются рекурсивные подделки – возможность изготовить подделку, которая возвращает поддельный объект, который возвращает поддельный объект..., не описывая все это явно. Получить продукт можно на странице www.telerik.com/products/mocking.aspx.

A.1.5. Microsoft Fakes (Moles)

Проект Microsoft Fakes был запущен в исследовательском центре Microsoft для ответа на вопрос: «Как мы можем подделывать файловую систему и другие вещи, например SharePoint, не покупая компанию типа Typemock?». В результате появился каркас Moles. Позже он перерос в Microsoft Fakes и теперь включается в некоторые версии Visual Studio.

MS Fakes – еще один неограниченный изолирующий каркас, который не имеет API для проверки того, что нечто вызывалось. По существу, он предоставляет средства для создания заглушек. Высказать утверждение о том, что некоторый объект вызывался, можно, но тестовый код при этом будет совершенно неудобочитаемым.

Как и описанные выше неограниченные каркасы, MS Fakes позволяет создавать поддельные объекты двух типов: вы либо генерируете неограниченные классы, которые наследуют и переопределяют уже тестопригодный код, либо используете прокладки (shim). *Прокладки* не ограничены, а *заглушки*, сгенерированные классы, ограничены. Запутались? Я тоже. Одна из причин, по которым я никому, кроме особо героических личностей, которым нечего терять, не рекомендую использовать MS Fakes – чудовищно неудобный API. Все запутанно до невозможности. К тому же, сопровождение тестов, в которых используются прокладки или заглушки, остается под вопросом. Сгенерированные заглушки необходимо перегенерировать при каждом изменении тестируемого кода, после чего необходимо изменить тесты. А код, где используются прокладки, очень длинный, его трудно

читать, а, значит, и сопровождать. Пусть даже MS Fakes бесплатный и включен в Visual Studio, но работа с ним обойдется очень дорого, если выражать цену ее в часах, потраченных разработчиком на попытки разобраться в ваших тестах и исправить их.

И еще один немаловажный момент: если вы пользуетесь MS Fakes, то в качестве каркаса тестирования обязаны использовать MSTest. Никакой другой не подойдет.

Если вам нужен неограниченный каркас для написания тестов, которые будут жить дольше, чем неделя-другая, то берите JustMock или Typemock Isolator.

Дополнительные сведения о MS Fakes читайте на странице http://msdn.microsoft.com/en-us/library/hh549175.aspx.

A.1.6. NSubstitute

NSubstitute – ограниченный изолирующий каркас с открытым исходным кодом. API очень прост для освоения и использования, имеется отличная документация. Еще плюс: сообщения об ошибках очень подробны. Наряду с FakeItEasy именно этот ограниченный каркас я бы в первую очередь рекомендовал для нового проекта.

Дополнительные сведения о NSubstitute см. на сайте http://nsubstitute.github.com/. Пакет устанавливается через NuGet.

A.1.7. FakeItEasy

У каркаса FakeItEasy не только очень удачное название, но и весьма симпатичный API. Он и NSubstitute на сегодня мои любимые ограниченные каркасы, но документация у FakeItEasy похуже. Что мне особенно нравится в API, так это буква `A` в начале любой операции, например:

```
var foo = A.Fake<IFoo>();
A.CallTo(() => foo.Bar()).MustHaveHappened();
```

Дополнительные сведения о FakeItEasy см. на странице https://github.com/FakeItEasy/FakeItEasy/wiki. Пакет устанавливается через NuGet.

A.1.8. Foq

Каркас Foq появился в ответ на потребность программистов на F# создавать подделки, удобные для использования из F#. Этот ограни-

ченный изолирующий каркас умеет подделывать абстрактные классы и интерфейсы. Лично я им не пользовался, т. к. никогда не работал на F#, но, похоже, в этом сегменте это единственное приемлемое решение. Дополнительные сведения о Foq см. на сайте https://foq.codeplex.com/. Пакет устанавливается через NuGet.

A.1.9. Isolator++

Isolator++ — распространяемый компанией Typemock неограниченный изолирующий каркас для C++. Он умеет подделывать статические методы, закрытые методы и многое другое в унаследованном коде на C++. Продукт коммерческий и, похоже, единственный в этом сегменте, обладающий такими возможностями. Дополнительные сведения см. на странице www.typemock.com/what-is-isolator-pp.

A.2. Каркасы тестирования

Каркасы тестирования предлагают функциональность двух типов:

- исполнители тестов выполняют написанные вами тесты, отображают результаты и позволяют узнать, где имели место ошибки;
- API тестирования включают атрибуты или базовые классы, которым вы должны унаследовать, а также API для высказывания утверждений.

Рассмотрим по очереди.

Исполнители тестов, работающие внутри Visual Studio:

- исполнитель MS, встроенный в Visual Studio;
- TestDriven.NET;
- ReSharper;
- NUnit;
- DevExpress;
- Typemock Isolator;
- NCrunch;
- ContinuousTests (Mighty Moose).

API тестирования и утверждений:

- NUnit.Framework;
- Microsoft.VisualStudio.TestPlatform.UnitTestFramework;
- Microsoft.VisualStudio.TestTools.UnitTesting;

- FluentAssertions;
- Shouldly;
- SharpTestEx;
- AutoFixture.

A.2.1. Непрерывный исполнитель тестов Mighty Moose (он же ContinuousTests)

Когда-то коммерческий, а ныне бесплатный, инструмент, Mighty Moose предназначен для того, чтобы непрерывно информировать о тестах и покрытии кода, как и NCrunch.

- Он прогоняет тесты в фоновом потоке.
- Тесты автоматически прогоняются после сохранения и компиляции кода.
- Встроенный алгоритм определяет, какие тесты прогонять в зависимости от того, какой код изменился.

К сожалению, развитие этого инструмента, похоже, прекращено. Дополнительные сведения см. на сайте http://continuoustests.com.

A.2.2. Непрерывный исполнитель тестов NCrunch

NCrunch – коммерческий продукт, предназначенный для того, чтобы непрерывно информировать о тестах и покрытии кода. Появившись сравнительно недавно, NCrunch завоевал мое сердце (я купил лицензию) благодаря нескольким удобным функциям.

- Он прогоняет тесты в фоновом потоке.
- Тесты запускаются автоматически при изменении кода, даже сохранять необязательно.
- Зеленые и красные точки слева от тестов и продуктового кода информируют о том, покрыта ли текущая строка каким-нибудь тестом и проходит в данный момент этот тест или нет.
- Он очень гибко конфигурируется, иногда это даже раздражает. Просто запомните: когда в начале работы над простым проектом открывается мастер, жмите **Esc**, принимая подразумеваемый по умолчанию режим прогона всех тестов.

Дополнительные сведения см. на сайте www.ncrunch.net/.

A.2.3. Исполнитель тестов Typemock Isolator

Этот исполнитель тестов является частью коммерческого изолирующего каркаса Typemock Isolator.

Пытается прогонять тесты и одновременно показывать покрытие при каждой компиляции. Пока в стадии бета-версии, ведет себя нестабильно. Быть может, когда-нибудь станет более полезным, но сейчас я его обычно отключаю и пользуюсь API изолирующих каркасов.

Дополнительные сведения см. на сайте http://Typemock.com.

A.2.4. Исполнитель тестов CodeRush

Этот исполнитель тестов является частью коммерческого продукта CodeRush – хорошо известной надстройки над Visual Studio.

Как и у ReSharper, у этого инструмента есть ряд достоинств:

- он хорошо интегрирован с редактором кода Visual Studio – показывает против каждого теста отметку, щелкнув по которой можно этот тест выполнить;
- поддерживает большинство API тестирования, имеющихся для .NET.
- если вы уже пользуетесь CodeRush, то этого исполнителя будет вполне достаточно.

Как и в случае ReSharper, визуальное представление результатов тестов может показаться помехой опытным практикам TDD. Дерево выполняющихся тестов и показ по умолчанию всех результатов, даже прошедших тестов, – пустая трата времени, когда работа над TDD в самом разгаре. Но некоторым нравится. Дело вкуса.

Дополнительные сведения см. на странице www.devexpress.com/Products/_Visual_Studio_Add-in/Coding_Assistance/unit_test_runner.xml.

A.2.5. Исполнитель тестов ReSharper

Этот исполнитель тестов является частью коммерческого продукта ReSharper – хорошо известной надстройки над Visual Studio.

У этого исполнителя есть ряд достоинств:

- он хорошо интегрирован с редактором кода Visual Studio – показывает против каждого теста отметку, щелкнув по которой можно этот тест выполнить;

- поддерживает большинство API тестирования, имеющихся для .NET.
- если вы уже пользуетесь ReSharper, то этого исполнителя будет вполне достаточно.

Недостатком я считаю излишне навязчивое визуальное представление результатов тестов. Дерево выполняющихся тестов очень красивое и расцвеченное. Но эта красота и показ по умолчанию всех результатов, даже прошедших тестов, только отвлекает, когда работа над TDD в самом разгаре. Но некоторым нравится. Дело вкуса.

Дополнительные сведения см. на странице www.jetbrains.com/resharper/features/unit_testing.html.

А.2.6. Исполнитель TestDriven.NET

Это коммерческий исполнитель тестов (бесплатный для личного использования). Пока я не начал пользоваться NCrunch, он был моим любимым. И его есть за что любить.

- Он умеет прогонять тесты для большинства, если не всех, каркасов тестирования, существующих на платформе .NET, в том числе NUnit, MSTest, xUnit.net, а также для некоторых каркасов на основе BDD.
- Пакет совсем небольшой с минималистским интерфейсом. Вывод результатов прост: они появляются в окне вывода Visual Studio плюс текст в нижних боковых панелях Visual Studio.
- Он очень быстрый, один из самых быстрых исполнителей тестов.
- У него есть уникальная функция: можно щелкнуть правой кнопкой мыши по *любому* участку кода (не только по тесту) и выбрать из контекстного меню пункт **Test with → Debugger** (Тестировать с помощью → Отладчик). После этого вы сможете пошагово отлаживать любой код (даже продуктовый, даже если для него нет тестов). Под капотом TD.NET вызывает метод, в котором вы находитесь, с помощью механизма отражения и, если ему нужны параметры, подставляет значения по умолчанию. Это экономит уйму времени при работе с унаследованным кодом.

Рекомендуется назначить комбинацию клавиш команде TD.NET `ReRunTests` в Visual Studio, чтобы, применяя методику TDD, не терять концентрации.

A.2.7. Исполнитель NUnit GUI

Исполнитель NUnit GUI бесплатный и распространяется с открытым исходным кодом. Он не интегрирован с Visual Studio, так что запускать его придется с рабочего стола. Поэтому им почти никто не пользуется, если имеются перечисленные здесь интегрированные альтернативы. Интерфейс плохо продуман, содержит много шероховатостей, так что не рекомендую.

A.2.8. Исполнитель MSTest

Исполнитель MSTest встроен во все версии Visual Studio. В платных версиях в нем имеется механизм надстроек, который позволяет добавлять поддержку прогона тестов с другими API, например NUnit или xUnit.net, с помощью специальных адаптеров, устанавливаемых как расширения Visual Studio.

Очком в пользу этого исполнителя является интеграция с комплектом инструментов Visual Studio Team System и хорошие готовые средства отчетности, покрытия и автоматизации сборки. Если в вашей компании для автоматизации сборки используется Team System, попробуйте MSTest в качестве исполнителя тестов в ночных и НИ-сборках, так как его возможности интеграции, в частности средства формирования отчетов, очень неплохи.

Из недостатков MSTest отмечу низкую производительность и наличие зависимостей.

- *Зависимости.* Для прогона тестов с помощью mstest.exe на машине сборке должна быть установлена Visual Studio. Кому-то это удобно, особенно если вы компилируете код там же, где прогоняете тесты. Но если вы хотите прогонять тесты в относительно чистом окружении, в уже откомпилированном виде, то это перебор, и уж совсем проблематично, если тесты требуется прогонять в окружении, где Visual Studio быть не должно.
- *Низкое быстродействие.* В MSTest многое происходит под капотом до и после выполнения каждого теста: копирование файлов, запуск внешних процессов, профилирование и прочее. В результате по ощущениям MSTest воспринимается как самый медленный исполнитель из всех, с которыми мне доводилось работать.

A.2.9. Pex

Pex (сокращение от «program exploration» – исследование программы) – интеллектуальный помощник программиста. Из параметризованного автономного теста он автоматически порождает традиционный комплект автономных тестов с высоким покрытием кода. Кроме того, он подсказывает программисту, как исправить ошибки.

Pex позволяет создать специальные тесты с параметрами, пометив их особыми атрибутами. Движок Pex генерирует новые тесты, которые затем можно прогонять как часть комплекта. Он очень хорош, когда нужно найти граничные условия и редкие случаи, которые программа обрабатывает неправильно. Pex следует использовать в дополнение к обычному каркасу тестирования, например NUnit или MbUnit.

Скачать Pex можно со страницы http://research.microsoft.com/projects/pex/.

A.3. API тестирования

Далее мы рассмотрим инструменты, предлагающие высокоуровневые абстракции и обертки для базовых каркасов автономного тестирования.

A.3.1. MSTest API – каркас автономного тестирования от Microsoft

Поставляется в составе редакции Visual Studio .NET Professional или старше. Включает ряд средств, аналогичных NUnit.

Но из-за некоторых проблем MSTest не столь хорош для автономного тестирования, как NUnit или xUnit.net:

- расширяемость;
- отсутствие `Assert.Throws`.

Расширяемость

Серьезная проблема этого каркаса – то, что расширить его не так просто, как другие каркасы тестирования. Хотя в прошлом в Сети велись дискуссии на тему того, как сделать MSTest более расширяемым за счет специальных атрибутов, похоже разработчики Visual Studio отказались от мысли победить в конкуренции с NUnit и другими продуктами.

Вместо этого в VS 2012 включен механизм надстроек, который позволяет назначить NUnit или еще что-нибудь каркасом тестирования по умолчанию, и тогда MSTest будет прогонять тесты, написанные для NUnit. Уже существуют адаптеры для NUnit и xUnit.net, если вас устраивает исполнитель MSTest с другими каркасами. К сожалению, в бесплатной редакции Visual Studio Express этот механизм отсутствует, что заставляет использовать не столь качественный MSTest. (Кстати говоря, почему Microsoft вынуждает нас платить за Visual Studio, чтобы мы разрабатывали код, который сделает господство платформы MS еще более всеобъемлющим?)

Отсутствие Assert.Throws

Тут все просто. В MSTest имеется атрибут `ExpectedException`, но нет метода `Assert.Throws`, который позволил бы проверить, что некоторая строка возбуждает исключение. Спустя шесть лет после появления на рынке и четыре года после того, как это было сделано в большинстве других каркасов, разработчики MSTest так и не удосужились добавить реализацию, насчитывающую буквально 10 строчек кода. Это заставляет меня задуматься, а интересны ли им вообще автономные тесты.

A.3.2. MSTest для приложений Metro (магазин Windows)

MSTest для приложений Metro – это API для создания приложений, выставляемых в магазине Windows. Выглядит он как MSTest, но, похоже, отношение к автономным тестам в нем изменилось в лучшую сторону. Например, появился метод `Assert.ThrowsException()`.

Создается впечатление, что при написании приложений для магазина Windows мы вынуждены использовать этот каркас, но решение существует, если воспользоваться связанными проектами. Дополнительные сведения см. на странице http://stackoverflow.com/questions/12924579/testing-a-windows-8-store-app-with-nunit.

A.3.3. NUnit API

В настоящее время NUnit – стандарт API тестирования де факто для разработчиков автономных тестов на платформе .NET. Он распространяется с открытым исходным кодом и используется практически всеми. В главе 2 я рассказывал о нем подробно. NUnit легко расширя-

ется и имеет обширное сообщество пользователей и форумы. Я рекомендую его всем, кто начинает заниматься автономным тестированием в .NET. Я пользуюсь им и по сей день.

Скачать NUnit можно с сайта www.Nunit.org.

A.3.4. xUnit.net

xUnit.net – каркас тестирования с открытым исходным кодом, разработанный в сотрудничестве с одним из авторов NUnit, Джимом Ньюкирком (Jim Newkirk). Это элегантный минималистский каркас, который стремится вернуться к основам, реализуя не больше, а меньше функций, чем в других каркасах. К тому же, названия атрибутов в нем другие.

Что в этом такого радикально отличающегося? Ну, во-первых, нет методов подготовки и очистки. В тестовом классе должен быть реализован конструктор и метод `Dispose`. Другое существенное отличие – простота расширения.

Поскольку xUnit.net так сильно отличается от других каркасов, требуется время, чтобы привыкнуть к нему после работы, скажем, с NUnit или MbUnit. Если вы никогда прежде не работали ни с какими каркасами, то освоить и использовать xUnit.net будет легко, и он достаточно стабилен для применения в реальном проекте.

Получить дополнительную информацию и скачать xUnit.net можно на сайте www.codeplex.com/xunit.

A.3.5. Вспомогательный API Fluent Assertions

Вспомогательный API Fluent Assertions – новая разновидность API тестирования. Это элегантная библиотека, предназначенная для одной-единственной цели: позволить высказывать утверждения о чем угодно безотносительно к используемому API тестирования. Например, с ее помощью можно получить функциональность типа `Assert.Throws()` в MSTest.

Дополнительные сведения см. на сайте http://fluentassertions.codeplex.com/.

A.3.6. Вспомогательный API Shouldly

Этот API очень похож на Fluent Assertions, но меньше. Он также предназначен для единственной цели: высказывать утверждения о

чем угодно безотносительно к используемому API тестирования. Дополнительные сведения см. на сайте http://shouldly.github.com.

A.3.7. Вспомогательный API SharpTestsEx

Как и Fluent Assertions, вспомогательный API SharpTestsEx предназначен для того, чтобы высказывать утверждения о чем угодно безотносительно к используемому API тестирования. Дополнительные сведения см. на сайте http://sharptestex.codeplex.com.

A.3.8. Вспомогательный API AutoFixture

AutoFixture – не API для утверждений. Он предназначен для того, чтобы упростить создание тестируемых объектов, специфика которых вам не интересна. Например, вам нужно какое-то число или строка. Считайте AutoFixture умной фабрикой, которая умеет внедрять объекты и входные данные в ваш тест.

Я знакомился с этим продуктом и больше всего мне понравилось его умение создавать экземпляр тестируемого класса, не зная сигнатуры его конструктора, т. к. это делает мои тесты более пригодными к длительному сопровождению. Но все же одного этого недостаточно, чтобы я начал его всерьез использовать, потому что то же самое я могу сделать, написав небольшой фабричный метод.

Кроме того, меня немного пугает тот факт, что он внедряет в мои тесты случайные значения, потому что получается, что при каждом прогоне я выполняю новый тест. Это также усложняет утверждения, т. к. мне приходится вычислять ожидаемое значение, исходя из случайных параметров внедренного объекта, а это может окончиться дублированием логики продуктового кода в тестах.

Дополнительные сведения можно найти на странице https://github.com/AutoFixture/AutoFixture.

A.4. IoC-контейнеры

IoC-контейнеры можно использовать для улучшения архитектурных свойств объектно-ориентированной системы за счет сокращения ручных технических издержек, связанных с применением методов добротного проектирования (использование параметров конструктора, управление временем жизни объектов и т. д.).

Контейнеры ослабляют связанность классов и их зависимостей, улучшают тестопригодность структуры классов и обеспечивают об-

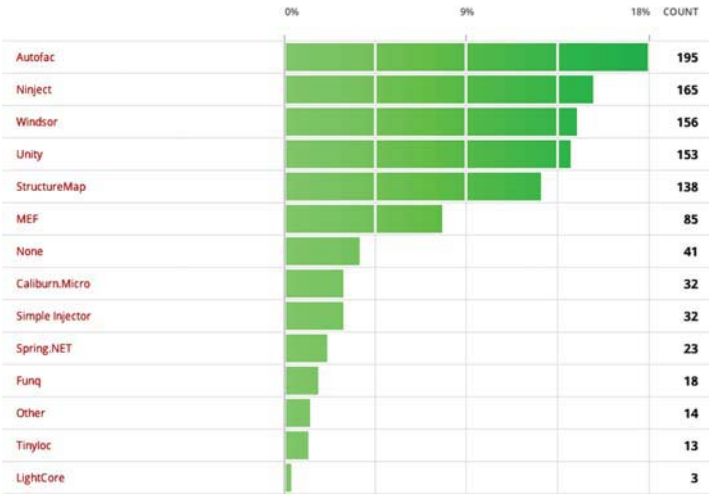
щие механизмы повышения гибкости. При благоразумном использовании контейнеры могут существенно расширить возможности повторного использования кода благодаря устранению прямых связей между классами и механизмами конфигурирования (например, за счет использования интерфейсов).

На платформе .NET в контейнерах нет недостатка. Они очень разнообразны и представляют немалый интерес для исследователя. Если вас волнует производительность, то на странице http://www.palmmedia.de/Blog/2011/8/30/ioc-container-benchmark-performance-comparison имеются результаты сравнения контейнеров. Лично мне никогда не казалось, что IoC-контейнеры – первопричина проблем с производительностью, но если бы это было так, то я бы первым делом заглянул на эту страницу.

Как бы то ни было, контейнеров много, но мы рассмотрим те, которые наиболее часто употребляются в сообществе.

Я отобрал их на основе результатов опроса, который провел в своем блоге в марте 2013 года. Вот как выглядит шестерка лучших:

- Autofac (Auto Factory)
- Ninject
- Castle Windsor
- Microsoft Unity
- StructureMap
- Microsoft Managed Extensibility Framework



Рассмотрим их по очереди.

A.4.1. Autofac

Autofac одним из первых предложил новый подход к IoC в .NET, хорошо согласующийся с синтаксическими нововведениями в C# 3 и 4. С точки зрения API, принятый в нем метод можно назвать минималистским. Его API радикально отличается от других каркасов, к нему нужно привыкнуть. Кроме того, для работы требуется .NET 3.5 и свободное владение синтаксисом лямбда-выражений. Рассказать об Autofac в двух словах довольно трудно, зайдите на сайт и посмотрите, на что он похож. Я рекомендую этот каркас тем, кто уже имеет опыт с другими DI-каркасами.

Домашняя страница Autofac находится по адресу http://code.google.com/p/autofac/.

A.4.2. Ninject

У Ninject простой синтаксис, работать с ним удобно. Больше особо сказать нечего, кроме того, пожалуй, что я горячо рекомендую познакомиться с ним.

Найти дополнительные сведения о Ninject можно на сайте http://ninject.org/.

A.4.3. Castle Windsor

Castle – большой проект с открытым исходным кодом, охватывающий самые разные территории. Windsor – одна из них, это зрелая и мощная реализация DI-контейнера.

Castle Windsor содержит большую часть функций, которые могут потребоваться от контейнера, и еще многое другое, но из-за этого освоить его нелегко.

Дополнительные сведения о контейнере Castle Windsor см. на странице http://docs.castleproject.org/Windsor.MainPage.ashx.

A.4.4. Microsoft Unity

Unity пришел на территорию DI-контейнеров довольно поздно, но предлагает простой минималистский подход, легко доступный начинающим. Опытным пользователям может чего-то не хватать, но нет сомнений, что каркас следует правилу 80–20: он предоставляет 80 % наиболее востребованных возможностей.

Каркас Unity разработан Microsoft, распространяется с открытым исходным кодом и хорошо документирован. Рекомендую его в качестве отправной точки для работы с контейнерами.

Домашняя страница Unity находится по адресу www.codeplex.com/unity.

A.4.5. StructureMap

StructureMap – контейнер с открытым исходным кодом, написанный Джереми Д. Миллером (Jeremy D. Miller). Его текучий API пытается моделировать естественный язык, в нем очень широко используются универсальные конструкции.

Документации маловато, зато имеются такие интересные возможности, как встроенный контейнер автоподставок (который может автоматически создавать заглушки, запрашиваемые тестом), развитый механизм управления временем жизни, конфигурирование без XML, интеграция с ASP.NET и многое другое.

Домашняя страница StructureMap находится по адресу http://structuremap.net.

A.4.6. Microsoft Managed Extensibility Framework

Managed Extensibility Framework (MEF) – на самом деле, не контейнер, но попадает в ту же общую категорию поставщиков служб создания экземпляров классов в программе. Он задуман как нечто гораздо большее, чем простой контейнер; это полноценная модель надстроек для небольших и крупных приложений. MEF включает облегченный IoC-контейнер, чтобы можно было без труда внедрять зависимости в разные места программы с помощью специальных атрибутов. Кривая обучения MEF довольно крута, и я не рекомендую использовать его просто в качестве IoC-контейнера. Но если ваше приложение нуждается в механизмах расширяемости, то MEF заодно может послужить и DI-контейнером.

Домашняя страница MEF находится по адресу http://mef.codeplex.com/.

A.5. Тестирование работы с базами данных

Как тестировать работу с базами данных – жгучий вопрос для многих начинающих. Например, следует ли заглушать базу данных в тестах? В этом я разделе я приведу некоторые рекомендации.

Но прежде несколько слов об интеграционном тестировании с участием базы данных.

A.5.1. Использование интеграционных тестов для уровня данных

Как тестировать уровень данных? Следует ли абстрагировать интерфейсы работы с базой данных? Нужно ли использовать настоящую базу данных?

Я обычно пишу для уровня данных (той части приложения, которая непосредственно обращается к базе данных) интеграционные тесты, потому что логика работы с данными почти всегда распределена между логикой приложения и самой базой (триггеры, правила защиты, ссылочная целостность и т. д.). Если вы не можете протестировать логику базы данных в полной изоляции (а я пока не встречал достойного каркаса для этой цели), то единственный способ проверить ее работу в тестах – подключить логику уровня данных к настоящей базе.

После совместного тестирования уровня данных и базы данных в проекте остается мало сюрпризов. Но само тестирование базы сопряжено с проблемами, из которых самая серьезная состоит в том, что база данных – это состояние, разделяемое многими тестами. Если один тест вставил в базу строку, то следующий ее увидит.

Поэтому необходим какой-то способ отката сделанных изменений, и, к счастью, в .NET Framework такой способ есть и притом несложный.

A.5.2. Использование TransactionScope для отката изменений данных

Класс `TransactionScope` достаточно развит для обработки очень сложных транзакций, в том числе вложенных, когда тестируемый код сам фиксирует локальную транзакцию.

Следующий простой пример показывает, как просто добавить в тесты механизм отката.

```
[TestFixture]
public class TrannsactionScopeTests
{
  private TransactionScope trans = null;

  [SetUp]
```

```
    public void SetUp()
    {
       trans = new TransactionScope(TransactionScopeOption.Required);
    }

    [TearDown]
    public void TearDown()
    {
       trans.Dispose();
    }

    [Test]
    public void TestServicedSameTransaction()
    {
       MySimpleClass c = new MySimpleClass();

       long id = c.InsertCategoryStandard("whatever");
       long id2 = c.InsertCategoryStandard("whatever");
       Console.WriteLine("Got id of " + id);
       Console.WriteLine("Got id of " + id2);
       Assert.AreNotEqual(id, id2);
    }
}
```

В методе подготовки мы создаем область видимости транзакции, а в методе очистки уничтожаем ее.

Поскольку на уровне тестового класса транзакция не фиксируется, то все изменения в базе откатываются, потому что метод `Dispose()` инициирует откат, если до этого не вызывался метод `Commit()`.

Некоторые считают разумным другой подход – прогонять тесты, подключаясь к базе данных в памяти. Я испытываю смешанные чувства по этому поводу. С одной стороны, такие тесты ближе к реальности, потому что тестируется логика базы данных. С другой стороны, если приложение работает с совершенно другой СУБД, имеющей собственные механизмы, то велик шанс, что какие-то вещи в тестах будут работать одним способом, а в производственной системе — другим. Я предпочитаю то, что максимально близко к реальной среде. Обычно это означает работу с одной и той же СУБД.

Но если база данных в памяти обладает точно такими же функциями и логикой, то идея может оказаться неплохой.

A.6. Тестирование веб-приложений

«Как тестировать веб-страницы?» – еще один часто поднимаемый вопрос. Вот несколько инструментов, которые могут помочь в этом предприятии:

- Ivonna;
- Team System Web Test;
- Watir;
- Selenium.

Дадим краткое описание каждого.

A.6.1. Ivonna

Ivonna – вспомогательный каркас для автономного тестирования, который устраняет необходимость использовать настоящие сеансы и страницы HTTP при прогоне тестов, относящихся к ASP.NET. За кулисами он делает удивительные вещи, в том числе компилирует тестируемые страницы и позволяет проверять содержащиеся в них элементы управления, не имея сеанса в браузере. Каркас полностью подделывает всю модель HTTP времени выполнения.

Автономные тесты пишутся точно так же, как при тестировании любых других объектов в памяти. Не нужно ни веб-сервера, ни прочей гадости.

Ivonna разрабатывается в партнерстве с Typemock и работает как надстройка над каркасом Typemock Isolator. Домашняя страница Ivonna находится по адресу http://ivonna.biz.

A.6.2. Тестирование веб-приложений в Team System

В редакции Visual Studio Team Test и Team Suite встроена возможность записывать и воспроизводить запросы к веб-страницам и попутно проверять различные вещи. Строго говоря, это интеграционное тестирование, но очень эффективное. В последних версиях появилась также возможность записывать Ajax-действия на странице, и, кроме того, работать стало гораздо удобнее.

Дополнительные сведения о Team System можно найти нас странице http://msdn.microsoft.com/en-us/teamsystem/default.aspx.

A.6.3. Watir

Watir (произносится так же, как «water») расшифровывается как «web application testing in Ruby» (тестирование веб-приложений в Ruby). Этот каркас с открытым исходным кодом позволяет описывать последовательность операций в браузере на языке программирования Ruby. Многие рубисты горячо рекомендуют его, но для этого потребуется изучить совсем новый язык. Впрочем, в ряде проектов для .NET Watir успешно используется, так что не такая уж это большая проблема.

Домашняя страница Watir находится по адресу http://watir.com/.

A.6.4. Selenium WebDriver

Selenium – это набор инструментов для автоматизации тестирования веб-приложений на разных платформах. Он старше всех остальных каркасов в моем списке, и его API имеет обертку для .NET. WebDriver – это расширение Selenium, поддерживающее самые разные браузеры, в том числе мобильные. Очень мощная штука.

Selenium – каркас интеграционного тестирования, весьма популярный. Рекомендую начинать с него. Но предупреждаю: его функциональность очень богата, а кривая обучения крута.

Домашняя страница Selenium находится по адресу http://docs.seleniumhq.org/ projects/webdriver/.

A.6.5. Coypu

Coypu – абстракция .NET поверх Selenium и других инструментов, относящихся к тестированию веб-приложений. На момент написания этой книги она только-только появилась, но потенциал велик. Возможно, стоит познакомиться поближе.

Дополнительные сведения можно найти по адресу https://github.com/featurist/coypu.

A.6.6. Capybara

Capybara – написанный на Ruby инструмент для автоматизации браузера. Для этой цели используется API RSpec (в стиле BDD), который многие находят прелестным.

Каркас Selenium более зрелый, но Capybara заманчивее и быстро прогрессирует. Если я пишу на Ruby, то пользуюсь именно этим каркасом.

Дополнительные сведения можно найти по адресу https://github.com/jnicklas/capybara.

A.6.7. Тестирование JavaScript

Для тех, кто собирается писать автономные или приемочные тесты для JavaScript-кода есть несколько инструментов. Отметим, что для работы с большинством из них требуется установить на машину Node.JS, но в наши дни это перестало быть сложной проблемой. Просто зайдите на страницу http://nodejs.org/download/.

Вот неполный перечень каркасов, на которые стоит взглянуть.

- *JSCover*. Используется для проверки покрытия JavaScript-кода тестами. http://tntim96.github.com/JSCover/.
- *Jasmin*. Очень известный каркас в стиле BDD, с которым мне доводилось работать. Рекомендую. http://pivotal.github.io/jasmine/.
- *Sinon.JS*. Создает подделки на JS. http://sinonjs.org/.
- *CasperJS + PhantomJS*. Используется для тестирования поведения JavaScript в браузере на машине без монитора и клавиатуры. Да, это не опечатка – реальный браузер не нужен (под капотом используется node.js). http://casperjs.org/.
- *Mocha*. Также широко известен и используется во многих проектах. http://visionmedia.github.com/mocha/.
- *QUnit*. Не первой молодости, но все еще хороший каркас тестирования. http://qunitjs.com/.
- *Buster.JS*. Совсем новый каркас. http://docs.busterjs.org/en/latest/.
- *Vows.js*. Перспективный, многообещающий каркас. https://github.com/cloudhead/vows.

A.7. Тестирование пользовательского интерфейса (персональных приложений)

Тестировать пользовательский интерфейс всегда трудно. Я не очень верю в автономные или интеграционные тесты пользовательского интерфейса, потому что отдача очень мала по сравнению с потраченным временем. На мой взгляд, интерфейсы слишком изменчивы для надежного тестирования. Поэтому я обычно стараюсь вынести всю

логику из пользовательского интерфейса на более низкий уровень, который можно автономно протестировать стандартными способами.

Я не могу порекомендовать никаких инструментов в этом сегменте (которые не побудили бы вас разбить клавиатуру через три месяца использования).

A.8. Тестирование многопоточных приложений

Потоки всегда были костью в горле для автономного тестирования. Ну не тестируются они – и все тут. Именно поэтому появляются новые каркасы, которые позволяют тестировать относящуюся к потокам логику (взаимоблокировки, состояния гонки и т. д.). Приведу два примера.

- Microsoft CHESS;
- Osherove.ThreadTester.

A.8.1. Microsoft CHESS

CHESS – перспективный инструмент, исходный код которого частично открыт Microsoft на сайте Codeplex.com. CHESS пытается обнаружить в коде ошибки, связанные с многопоточностью (взаимоблокировки, зависания, активные блокировки и т. д.), пробуя всевозможные перестановки порядка выполнения потоков. Такие тесты пишутся как обычные автономные тесты.

Домашняя страница находится по адресу http://chesstool.codeplex.com.

A.8.2. Osherove.ThreadTester

Это небольшой каркас с открытым исходным кодом, который я разработал некоторое время назад. Он позволяет запускать в одном тесте несколько потоков и смотреть, не произошло ли в коде что-то странное (к примеру, взаимоблокировка). Его функциональность не очень богата, это лишь попытка реализовать многопоточный тест (а не тест многопоточного кода).

Скачать каркас можно из моего блога по адресу http://osherove.com/blog/2007/6/22/multithreaded-unit-tests-with-osherovethreadtester.html.

A.9. Приемочное тестирование

Приемочное тестирование способствует более тесному сотрудничеству заказчиков и разработчиков программного обеспечения. В ходе этой деятельности заказчики, тестировщики и разработчики начинают лучше понимать, что должна делать программа, а тесты автоматически сравнивают полученные результаты с ожидаемыми. Это прекрасный способ совместной работы над сложными задачами (и нахождения правильных решений) на ранних стадиях разработки.

К сожалению, существует не так много каркасов для автоматизированного приемочного тестирования, и лишь один действительно работает! Надеюсь, что скоро положение дел изменится. Вот список достойных внимания каркасов:

- FitNesse;
- SpecFlow;
- Cucumber;
- TickSpec.

A.9.1. FitNesse

FitNesse — не требовательный к ресурсам каркас с открытым исходным кодом, который по идее должен упростить разработчикам определение приемочных тестов – в виде веб-страниц, содержащих простые таблицы входных данных и ожидаемых результатов, – прогон тестов и отображение результатов.

FitNesse содержит много ошибок, но тем не менее используется в разных местах с переменным успехом. Лично мне не удалось заставить его работать нормально.

Дополнительные сведения о FitNesse можно найти на сайте www.fitnesse.org.

A.9.2. SpecFlow

SpecFlow – попытка привнести в мир .NET то, чем Cucumber облагодетельствовал мир Ruby: инструмент, с помощью которого можно писать спецификации в виде простых текстовых файлов, привлекая к работе заказчиков и отдел контроля качества.

С этой работой он неплохо справляется. Подробности см. на сайте http://www.specflow.org.

A.9.3. Cucumber

Cucumber – основанный на Ruby инструмент, позволяющий писать спецификации на специальном языке Gherkin (согласен, не звучит). Это обычные текстовые файлы, для которых затем нужно написать связующий код, который будет исполнять фактический код, воздействующий на код приложения.

Кажется сложным, но на самом деле это не так.

Но что здесь делает инструмент для Ruby? Дело в том, что он послужил источником вдохновения для целого семейства новых инструментов в мире .NET, из которых на данный момент, похоже, выжил только один – SpecFlow.

Однако существует способ исполнять Cucumber на платформе .NET, если вы используете IronRuby – язык, который Microsoft начала разрабатывать и бросила, отдав на растерзание поклонникам открытого исходного кода, чтобы никогда больше о нем не слышать (отличная работа!).

Как бы то ни было, Cucumber достаточно важен, чтобы знать о нем, пусть даже вы и не собираетесь им пользоваться. Он поможет понять, почему на платформе .NET пытаются сделать то же самое.

Кроме того, Cucumber лежит в основе языка Gherkin, который пытаются реализовать некоторые нынешние и будущие инструменты. Читайте подробности на http://cukes.info/.

A.9.4. TickSpec

TickSpec интересен тем, кто работает на F#. Сам я его не использовал, потому что не пишу на F#, но знаю, что он из того же ряда каркасов приемочного тестирования в стиле BDD, что и все вышеупомянутые. Я не слыхал, чтобы кто-то им пользовался, но, быть может, это потому, что я не вращаюсь в кругах, близких к F#. Подробности см. на сайте https://tickspec.codeplex.com/.

A.10. Каркасы с API в стиле BDD

В последние годы появилась целая плеяда каркасов, имитирующих еще один инструмент из мира Ruby – *RSpec*. Этот инструмент возвестил миру о том, что, быть может, автономное тестирование – на такое уж удачное название, и что, заменив его на BDD (behaviour-driven development – разработка через поведение), мы сможем повы-

сить удобочитаемость и, чем черт не шутит, даже более осмысленно общаться с заказчиками на тему поведения программы.

На мой взгляд, идея реализовать эти каркасы просто как другие API, с помощью которых можно писать автономные или интеграционные тесты, уже отрицает саму возможность больше (чем раньше) общаться с заказчиками, потому что заказчики вряд ли будут читать или изменять ваш код. Я полагаю, что каркасы приемочного тестирования, описанные в предыдущем разделе, куда лучше отвечают этому направлению мыслей.

Таким образом, остаются только кодировщики, пытающиеся использовать новые API.

Поскольку эти API черпают вдохновение из языка Cucumber, построенного в стиле BDD, иногда они получаются более удобочитаемыми, но, на мой взгляд, не в простых случаях, для которых лучше подходит незамысловатый стиль утверждений. Вы можете думать иначе.

Перечислю несколько наиболее известных каркасов в стиле BDD. Я не стал описывать их в специальных подразделах, потому что не использовал ни один из них в реальном проекте сколько-нибудь долго.

- NSpec – самый почтенный, но все еще, похоже, в хорошей форме. Подробности на http://nspec.org/.
- StoryQ – еще один старичок-бодрячок. Порождает в высшей степени удобочитаемый результат и включает также инструмент для трансляции написанных на Gherkin историй в компилируемый код тестов. Подробности на http://storyq.codeplex.com/.
- MSpec, или Machine.Specifications, пытается как можно ближе соответствовать источнику (RSpec), применяя многочисленные трюки с лямбда-выражениями. Затягивает. Подробности на https://github.com/machine/machine.specifications.
- TickSpec – та же идея, но реализованная для F#. Подробности на http://tickspec.codeplex.com/.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Ц

Ш

Я

A

B

C

D

E

P

R

S

T

V

W

X

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) 725-54-09, 725-50-27**;

Электронный адрес: **books@alians-kniga.ru**.

Рой Ошероув

Искусство автономного тестирования с примерами на C#

Второе издание

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. 20,05.
Тираж 200 экз.

Web-сайт издательства: www.дмк.рф

Вы знаете, что должны заниматься автономным тестированием, так почему же не занимаетесь? Если вы новичок в этом деле, если считаете, что автономное тестирование — это скучно, или просто не получаете отдачи от затраченных усилий, читайте дальше.

Во втором издании книги «Искусство автономного тестирования с примерами на C#» автор шаг за шагом проведет вас по пути от первого простенького автономного теста до создания полного комплекта тестов — понятных, удобных для сопровождения и заслуживающих доверия. Вы и не заметите, как перейдете к более сложным вопросам — заглушкам и подставкам — и попутно научитесь работать с изолирующими каркасами типа Moq, FakeItEasy или Typemock Isolator. Вы узнаете о паттернах тестирования и организации тестов, о том, как проводить рефакторинг приложений и тестировать «нетестопригодный» код. Не забыл автор и об интеграционном тестировании и тестировании работы с базами данных.

Внутри книги:

- Создание удобочитаемых, пригодных для сопровождения и заслуживающих доверия тестов;
- Подделки, заглушки, подставки и изолирующие каркасы;
- Простые приемы внедрения зависимости;
- Рефакторинг унаследованного кода.

Эта книга — нечто особенное. Из глав, опирающихся друг на друга, возводится конструкция, поражающая своей глубиной. Готовьтесь получить удовольствие.

— Из предисловия Роберта С. Мартина, cleancoder.com

Самый лучший способ изучить автономное тестирование по книге, ставшей уже классической в этой области.

— Рафаэль Фариа, LG Electronics

Учит философии в не меньшей степени, чем технике эффективного автономного тестирования.

— Прадип Челлаппан, Microsoft

Когда коллеги спрашивают меня, как правильно писать автономные тесты, отвечаю просто: «Купите эту книгу»!

— Алессандро Кампеис, Vimar SpA

Самый лучший источник по автономному тестированию.

— Калеб Педерсон , Next IT Corporation

Примеры в книге написаны на C#, но будут понятны всем, кто владеет любым статически типизированным языком, например Java или C++.

Рой Ошеров пишет программы свыше 15 лет, а также занимается консультированием и обучением коллективов разработчиков тонкому искусству автономного тестирования и разработки через тестирование. Он ведет блог на сайте ArtOfUnitTesting.com.

ISBN 978-5-94074-945-5

Интернет-магазин:
www.dmkpress.com
Книга – почтой:
orders@alians-kniga.ru
Оптовая продажа:
“Альянс-книга”
тел.(499)725-54-09
books@alians-kniga.ru

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф