ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ КО ВТОРОМУ ИЗДАНИЮ РОБЕРТА С. МАРТИНА

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, что 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 может 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 КО ВТОРОМУ ИЗДАНИЮ МАЙКЛА ФЭЗЕРСА

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.

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 изоляции, 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.

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.

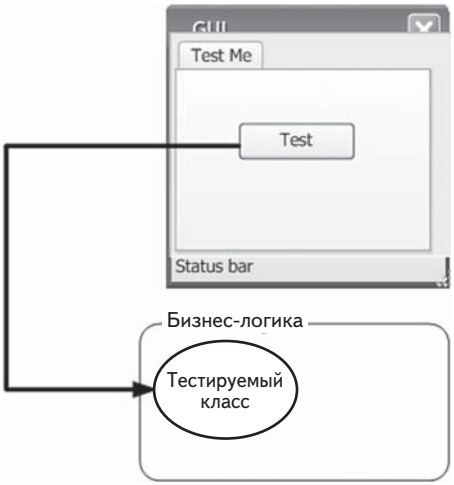

Рис. 1.1. При классическом тестировании разработчик использует графический интерфейс пользователя (ГИП) для активации некоторого действия в тестируемом классе. Затем проверяет результат.

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.

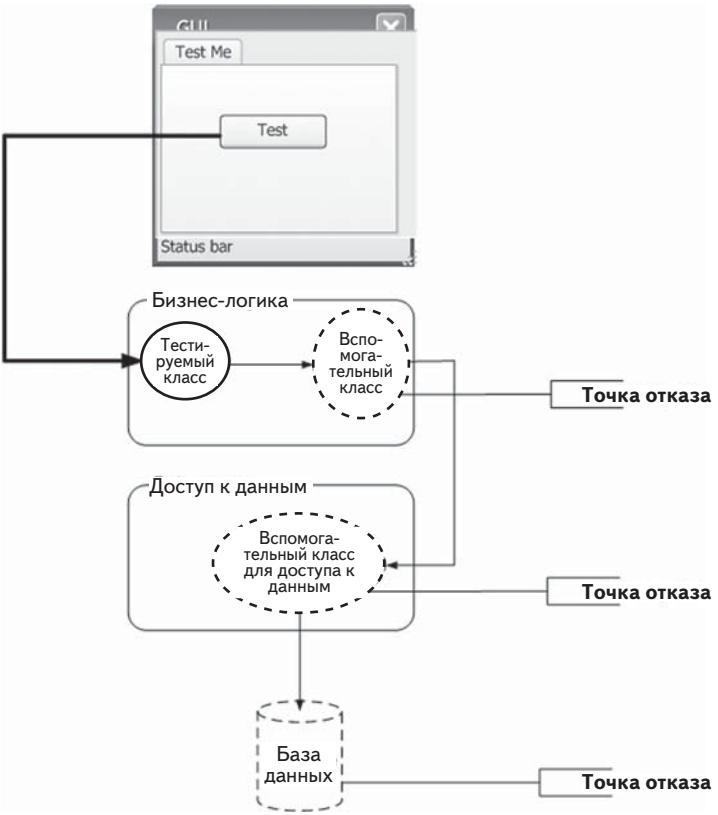

Рис. 1.2. В интеграционном тесте может быть много точек отказа. Все автономные единицы должны работать совместно, и каждый может отказать, поэтому найти источник ошибки не всегда просто.

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.