Искусство автономного тестирования

с примерами на C#

Рой Ошероув

Второе издание

ДМК ИЗДАТЕЛЬСТВО

MANNING

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

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.

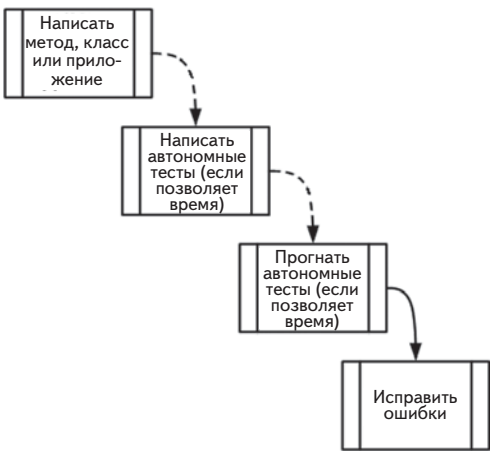

Рис. 1.3. Традиционный способ написания автономных тестов. Пунктирными линиями представлены действия, которые принято считать необязательными

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.

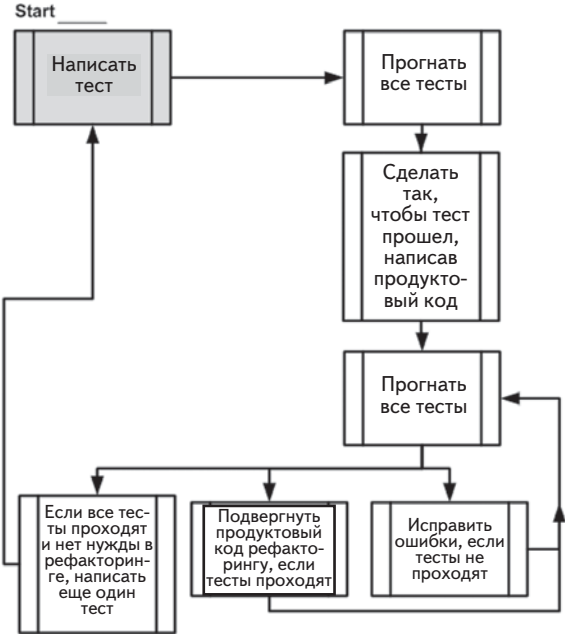

Рис. 1.4. Разработка через тестирование – взгляд с высоты птичьего полета. Обратите внимание на спиральный характер процесса: написать тест, написать код, выполнить рефакторинг, написать следующий тест. Это демонстрация инкрементной природы TDD: мелкие шаги приводят к высококачественному конечному результату

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.

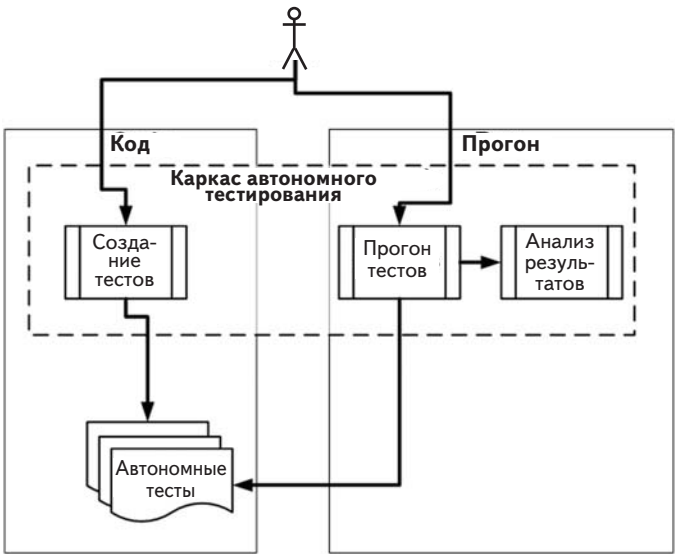

Рис. 2.1. При написании автономных тестов используются библиотеки, входящие в состав каркаса тестирования. Затем тесты прогоняются с помощью специального инструмента или непосредственно в IDE, а результаты (представленные в виде текста или в графическом интерфейсе каркаса) анализируются разработчиком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

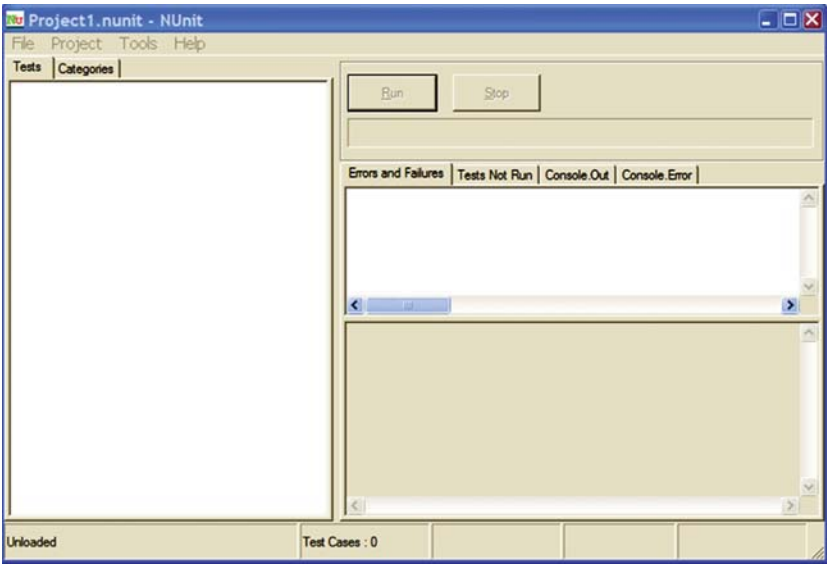

Рис. 2.2. Пользовательский интерфейс NUnit состоит из трех основных частей: дерева тестов слева, области сообщений и ошибок справа вверху и трассировки стека справа внизу

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):