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.

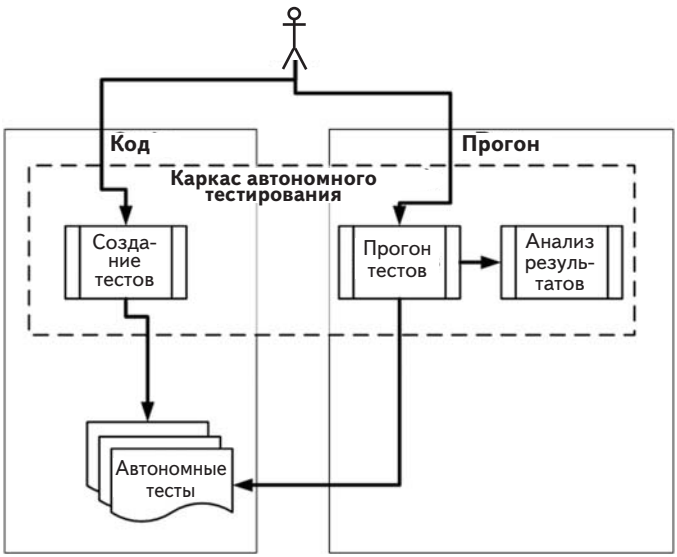

Рис. 2.1. При написании автономных тестов используются библиотеки, входящие в состав каркаса тестирования. Затем тесты прогоняются с помощью специального инструмента или непосредственно в IDE, а результаты (представленные в виде текста или в графическом интерфейсе каркаса) анализируются разработчиком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

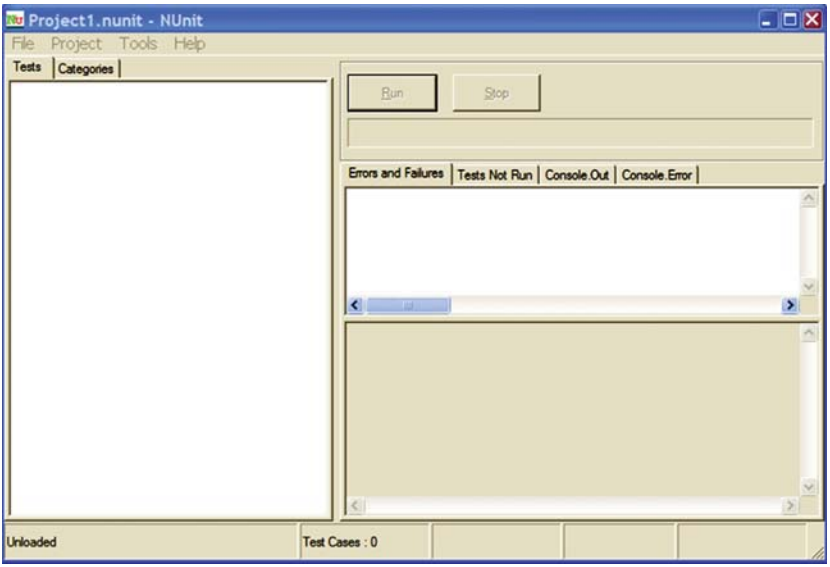

Рис. 2.2. Пользовательский интерфейс NUnit состоит из трех основных частей: дерева тестов слева, области сообщений и ошибок справа вверху и трассировки стека справа внизу

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.)

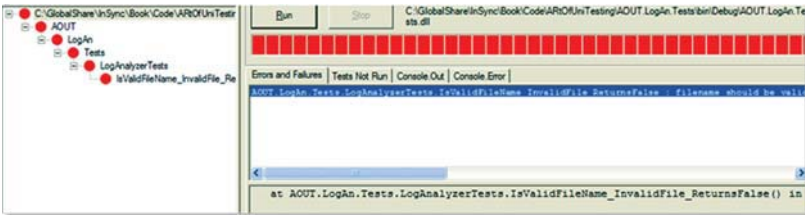

Рис. 2.3. Информация о том, что некоторые тесты не прошли, в NUnit показывается в трех местах: иерархия тестов слева и индикатор хода выполнения сверху становятся красными, а в область справа выводятся сообщения об ошибках

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.

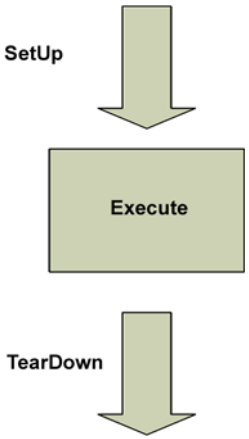

Рис. 2.4. NUnit производит подготовку и очистку соответственно до и после выполнения каждого метода

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.

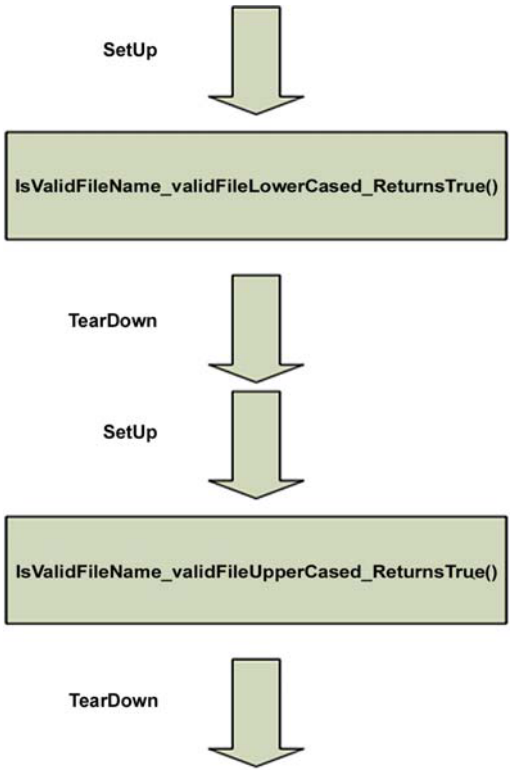

Рис. 2.5. Как NUnit вызывает методы с атрибутами `SetUp` и `TearDown` при наличии нескольких тестов в одном классе: до начала выполнения каждого теста вызывается `SetUp`, а после его завершения – `TearDown`

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.

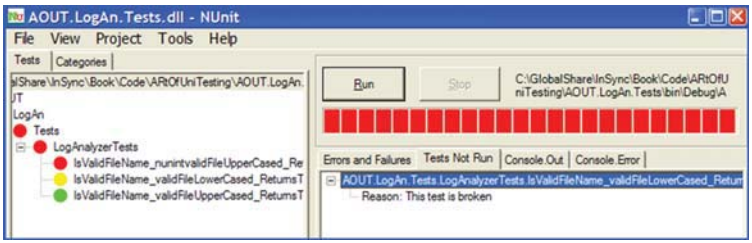

Рис. 2.6. В NUnit игнорируемый тест обозначается желтым цветом (средний тест), а причина, по которой он не выполняется, отображается на вкладке Tests Not Run в правой области

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**.

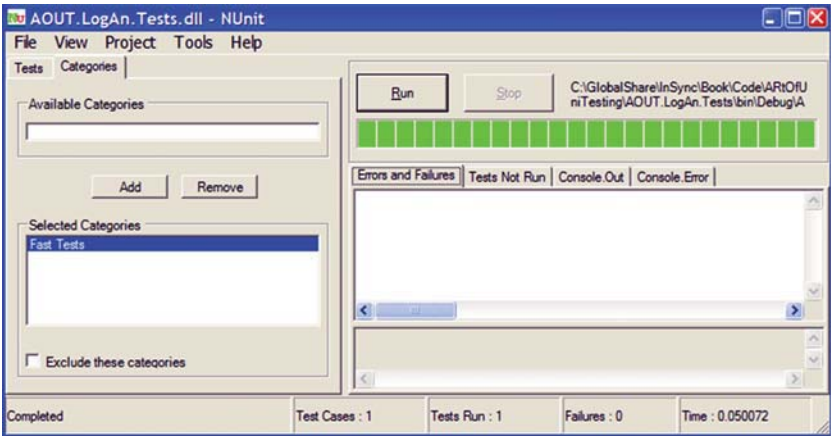

Рис. 2.7. Мы можем распределить тесты по категориям, а затем в пользовательском интерфейсе NUnit указать, из какой категории прогонять тесты

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.