2 A first unit test

This chapter covers

- Exploring unit testing frameworks in .NET
- Writing your first test with NUnit
- Working with the NUnit attributes
- Understanding the three output types of a unit of work

When I first started writing unit tests with a real unit testing framework, there was little documentation, and the frameworks I worked with didn't have proper examples. (I was mostly coding in VB 5 and 6 at the time.) It was a challenge learning to work with them, and I started out writing rather poor tests. Fortunately, times have changed.

This chapter will get you started writing tests even if you have no idea where to start. It will get you well on your way to writing real-world unit tests with a framework called NUnit—a .NET unit testing framework. It's my favorite framework in .NET for unit testing because it's easy to use, easy to remember, and has lots of great features.

There are other frameworks in .NET, including some with more features, but NUnit is where I always start. If the need arises, I sometimes then expand to a different framework. We'll look at how NUnit works, its syntax, and how to run it and get feedback when the test fails or passes. To accomplish this, I'll introduce a small

software project that we'll use throughout the book to explore testing techniques and best practices.

You may feel like NUnit is forced on you in this book. Why not use the built-in MSTest framework in Visual Studio? The answer consists of two parts:

- NUnit contains better features than MSTest relating to writing unit tests and test attributes that help write more maintainable, readable tests.
- In Visual Studio 2012, the built-in test runner allows running tests written in other frameworks, including NUnit. To allow this, simply install the NUnit test adapter for Visual Studio via NuGet. (NuGet is explained later in this chapter.)

This makes the choice of which framework to use pretty easy for me.

First, we need to look at what a unit testing framework is and at what it enables you to do that you couldn't and wouldn't do without it.

2.1 Frameworks for unit testing

Manual tests suck. You write your code, you run it in the debugger, you hit all the right keys in your app to get things just right, and then you repeat all this the next time you write new code. And you have to remember to check all that other code that might have been affected by the new code. More manual work. Great.

Doing tests and regression testing completely manually, repeating the same actions again and again like a monkey, is error prone and time consuming, and people seem to hate doing that as much as anything can be hated in software development. These problems are alleviated by tooling. Unit testing frameworks help developers write tests more quickly with a set of known APIs, execute those tests automatically, and review the results of those tests easily. And they never forget! Let's dig deeper into what they offer.

2.1.1 What unit testing frameworks offer

Up to now, for many of you reading this, the tests you've done were limited:

- *They weren't structured.* You had to reinvent the wheel every time you wanted to test a feature. One test might have looked like a console application, another used a UI form, and another used a web form. You didn't have time to spend on testing, and the tests failed the "easy to implement" requirement.
- *They weren't repeatable.* Neither you nor your team members could run the tests you'd written in the past. That breaks the "repeatedly" requirement and prevents you from finding regression bugs. With a framework, you can more easily and automatically write tests that are repeatable.
- *They didn't cover all the important parts of the code.* The tests didn't test all the code that matters. That means all the code with logic in it, because each and every one of those could contain a potential bug. (Property getters and setters don't count as logic but will eventually get used as part of some unit of work.) If it were easier to write the tests, you'd be more inclined to write more of them and get better coverage.

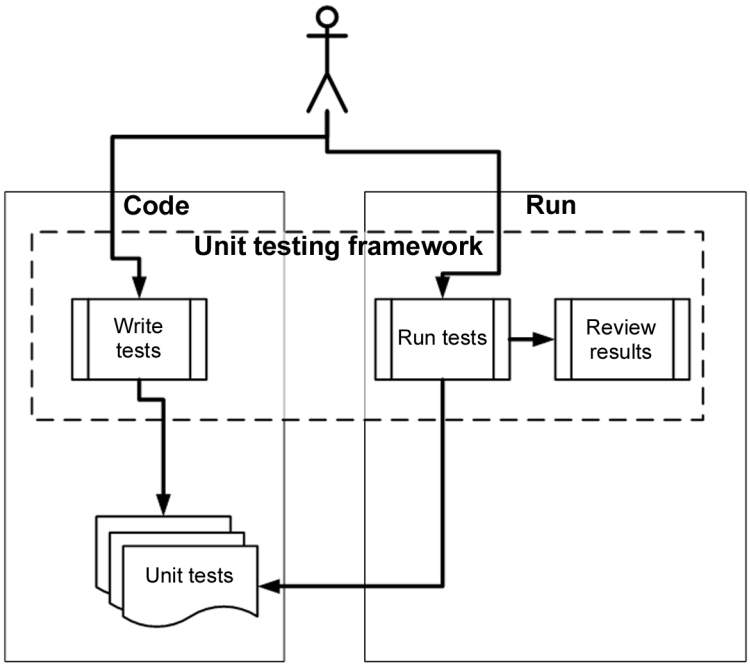

Figure 2.1 Unit tests are written as code, using libraries from the unit testing framework. Then the tests are run from a separate unit testing tool or inside the IDE, and the results are reviewed (either as output text, the IDE, or the unit testing framework application UI) by the developer or an automated build process.

In short, what you've been missing is a *framework* for writing, running, and reviewing unit tests and their results. Figure 2.1 shows the areas in software development where a unit testing framework has influence.

Unit testing frameworks are code libraries and modules that help developers unit test their code, as outlined in table 2.1. They also have another side—running the tests as part of an automated build, which I cover in later chapters.

Table 2.1 How unit testing frameworks help developers write and execute tests and review results

Unit testing practice	How the framework helps
Write tests easily and in a structured manner.	Framework supplies the developer with a class library that contains ▪ Base classes or interfaces to inherit ▪ Attributes to place in your code to note which of your methods are tests ▪ Assertion classes that have special assertion methods you invoke to verify your code
Execute one or all of the unit tests.	Framework provides a test runner (a console or GUI tool) that ▪ Identifies tests in your code ▪ Runs tests automatically ▪ Indicates status while running ▪ Can be automated by the command line

Table 2.1 How unit testing frameworks help developers write and execute tests and review results *(continued)*

Unit testing practice	How the framework helps
Review the results of the test runs.	The test runners will usually provide information such as ▪ How many tests ran ▪ How many tests didn't run ▪ How many tests failed ▪ Which tests failed ▪ The reason tests failed ▪ The `ASSERT` message you wrote ▪ The code location that failed ▪ Possibly a full stack trace of any exceptions that caused the test to fail, and will let you go to the various method calls inside the call stack

At the time of this writing, there are more than 150 unit testing frameworks out there—practically one for every programming language in public use. You can find a good list at http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks. Consider that .NET alone has at least 3 different active unit testing frameworks: MSTest (from Microsoft), xUnit.net, and NUnit. Among these, NUnit was once the de facto standard. These days I feel it's quite a battle between MSTest and NUnit, simply because MSTest is built into Visual Studio. But, when given a choice, I'd choose NUnit for some of the features you'll see later in this chapter and also in the appendix about tools and frameworks.

> **NOTE** Using a unit testing framework doesn't ensure that the tests you write are *readable*, *maintainable*, or *trustworthy* or that they cover all the logic you'd like to test. We'll look at how to ensure that your unit tests have these properties in chapter 7 and in various other places throughout this book.

2.1.2 The xUnit frameworks

Collectively, these unit testing frameworks are called the *xUnit frameworks* because their names usually start with the first letters of the language for which they were built. You might have CppUnit for C++, JUnit for Java, NUnit for .NET, and HUnit for the Haskell programming language. Not all of them follow these naming guidelines, but most do.

In this book, we'll be using NUnit, a .NET unit testing framework that makes it easy to write tests, run them, and get the results. NUnit started out as a direct port of the ubiquitous JUnit for Java and has since made tremendous strides in its design and usability, setting it apart from its parent and breathing new life into an ecosystem of test frameworks that's changing more and more. The concepts we'll be looking at will be understandable to Java and C++ developers alike.

2.2 Introducing the LogAn project

The project that we'll use for testing in this book will be simple at first and will contain only one class. As the book moves along, we'll extend that project with new classes and features. We'll call it the LogAn project (short for "log and notification").

Here's the scenario. Your company has many internal products it uses to monitor its applications at customer sites. All these products write log files and place them in a special directory. The log files are written in a proprietary format that your company has come up with that can't be parsed by any existing third-party tools. You're tasked with building a product, LogAn, that can analyze these log files and find special cases and events in them. When it finds these cases and events, it should alert the appropriate parties.

In this book, I'll teach you to write tests that verify LogAn's parsing, event-recognition, and notification abilities. Before we get started testing our project, though, we'll look at how to write a unit test with NUnit. The first step is installing it.

2.3 First steps with NUnit

As with any new tool, you'll need to install it first. Because NUnit is open source and freely downloadable, this task will be rather simple. Then you'll see how to start writing a test with NUnit, use the built-in attributes that NUnit ships with, and run your test and get some real results.

2.3.1 Installing NUnit

The best and easiest way to install NUnit is by using NuGet—a free extension to Visual Studio that allows you to search, download, and install references to popular libraries from within Visual Studio with a few clicks or a simple command text.

I highly suggest you install NuGet by going to the Tools > Extension Manager menu in Visual Studio, clicking Online Gallery, and installing the top-ranked NuGet Package Manager. After installation don't forget to restart Visual Studio, and voilà—you have a very powerful and easy tool to add and manage references to your projects. (If you come from the Ruby world, you'll notice NuGet resembles Ruby Gems and the GemFile idea, although it's still very new in terms of features related to versioning and deployment to production.)

Now that you have NuGet installed, you can open the following menu: Tools > Library Package Manager > Package Manager Console, and type `Install-Package NUnit` in the text window that appears. (You can also use the Tab key to autocomplete possible commands and library package names.)

Once all is said and done, you should see a nice message, "NUnit Installed Successfully." NuGet will have locally downloaded a zip file containing NUnit files, added a reference to the default project that is set in the Package Manager Console window's combo box, and finished by telling you it did all these things. You should now see a reference to NUnit.Framework.dll in your project.

A note about the NUnit GUI—this is the basic UI runner that NUnit has. I cover this UI later in this chapter, but I usually don't use it. Consider it more of a learning tool so you can understand how NUnit runs as a bare-bones tool with no add-ons to Visual Studio. It also doesn't come bundled with NuGet's version of NUnit. NuGet installs only required DLLs but not the UI (this makes some sense, because you can have

multiple projects using NUnit, but you don't need multiple versions of its UI to run them). To get the NUnit UI, which I also show a bit later in this chapter, you can install NUnit.Runners from NuGet, or you can go to NUnit.com and install the full version from there. This full version also comes bundled with the NUnit Console Runner, which you use when running tests on a build server.

If you don't have access to NUnit, you can download it from www.NUnit.com and add a reference to nunit.framework.dll manually.

As a bonus, NUnit is an open source product, so you can get the source code for NUnit, compile it yourself, and use the source freely within the limits of the open source license. (See the license.txt file in the program directory for license details.)

NOTE At the time of writing, the latest version of NUnit is 2.6.0. The examples in this book should be compatible with most future versions of the framework.

If you chose the manual route to install NUnit, run the setup program you downloaded. The installer will place a shortcut to the GUI part of the NUnit runner on your desktop, but the main program files should reside in a directory named something like C:\Program Files\NUnit-Net-2.6.0. If you double-click the NUnit desktop icon, you'll see the unit test runner shown in figure 2.2.

NOTE The C# Express Edition of Visual Studio (or above) is fine for use with this book.

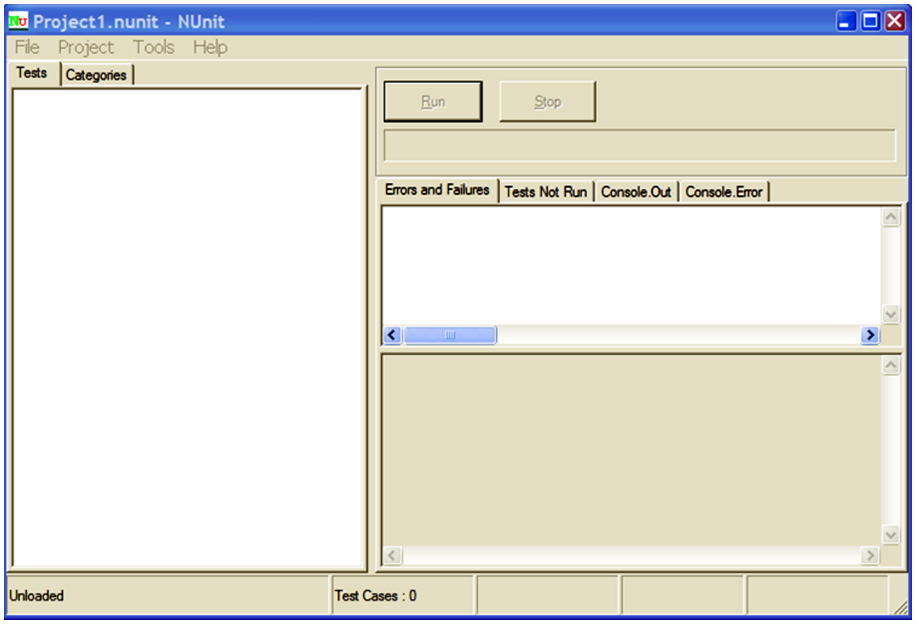

Figure 2.2 The NUnit GUI is divided into three main parts: the tree listing the tests on the left, messages and errors at the top right, and stack trace information at the bottom right.

2.3.2 Loading up the solution

If you have the book's code on your machine, load up the ArtOfUnitTesting2ndEd .Samples.sln solution from the Code folder inside Visual Studio 2010 or later.

We'll begin by testing the following simple class with one method (the unit you're testing) inside it:

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

        return true;
    }
}
```

Please note that I've purposely left out an `!` before the `if`, so that this method has a bug—it returns `false` instead of `true` when the filename ends with .SLF. This is so you can see what it looks like in a test runner when a test fails.

This method may not seem complicated, but we'll test it to make sure it works, mostly to follow through the testing routine. In the real world, you'll want to test any method that contains logic, even if it seems simple. Logic can fail, and you want to know when it does. In the following chapters, we'll test more complicated scenarios and logic.

The method looks at the file extension to determine whether or not a file is a valid log file. The first test will be to send in a valid filename and make sure the method returns `true`.

Here are the first steps for writing an automated test for the `IsValidLogFileName` method:

1. Add a new class library project to the solution, which will contain your test classes. Name it LogAn.UnitTests (assuming the original project name is LogAn.csproj).
2. To that library, add a new class that will hold your test methods. Name it `LogAnalyzerTests` (assuming that your class under test is named `LogAnalyzer.cs`).
3. Add a new method to the preceding test case named `IsValidLogFileName_BadExtension_ReturnsFalse()`.

We'll touch more on test-naming and arrangement standards later in the book, but the basic rules are listed in table 2.2.

Table 2.2 Basic rules for placing and naming tests

Object to be tested	Object to create on the testing side
Project	Create a test project named [ProjectUnderTest].UnitTests.
Class	For a class located in `ProjectUnderTest`, create a class with the name `[ClassName]Tests`.

Table 2.2 Basic rules for placing and naming tests *(continued)*

Object to be tested	Object to create on the testing side
Unit of work (a method, or a logical grouping of several methods, or several classes)	For each unit of work, create a test method with the following name: `[UnitOfWorkName]_[ScenarioUnderTest]_[ExpectedBehavior]`. The unit of work name could be as simple as a method name (if that's the whole unit of work) or more abstract if it's a use case that encompasses multiple methods or classes such as `UserLogin` or `RemoveUser` or `Startup`. You might feel more comfortable starting with method names and moving to more abstract names later. Just make sure that if these are method names, those methods are public, or they don't really represent the start of a unit of work.

The name for our LogAn test project is LogAn.UnitTests. The name for the `LogAnalyzer` test class is `LogAnalyzerTests`.

Here are the three parts of the test method name:

- `UnitOfWorkName`—The name of the method or group of methods or classes you're testing.
- `Scenario`—The conditions under which the unit is tested, such as "bad login" or "invalid user" or "good password." You could describe the parameters being sent to the public method or the initial state of the system when the unit of work is invoked such as "system out of memory" or "no users exist" or "user already exists."
- `ExpectedBehavior`—What you expect the tested method to do under the specified conditions. This could be one of three possibilities: return a value as a result (a real value, or an exception), change the state of the system as a result (like adding a new user to the system, so the system behaves differently on the next login), or call a third-party system as a result (like an external web service).

In our test of the `IsValidLogFileName` method, the scenario is that you're sending the method a valid filename, and the expected behavior is that the method will return a `true` value. The test method name might be `IsValidFileName_BadExtension_ReturnsFalse()`.

Should you write the tests in the production code project? Or maybe separate them into a different test-related project? I usually prefer to separate them, because it makes all the rest of the test-related work easier. Also, lots of people aren't happy including tests in their production code, which leads to ugly conditional compilation schemes and other bad ideas that make code less readable.

I'm not religious about this. I also like the idea of having tests next to your running production app so you can test its health after deployment. That requires some careful thought, but it does *not* require you to have the tests and production code in the same project. You *can* actually have your cake and eat it too.

You haven't used the NUnit test framework yet, but you're close. You still need to add a reference to the project under test for the new testing project. Do this by right-clicking the test project and selecting Add Reference. Then select the Projects tab and select the LogAn project.

The next thing to learn is how to mark the test method to be loaded and run by NUnit automatically. First, make sure you've added the NUnit reference either by using NuGet or manually, as explained in section 2.3.1.

2.3.3 Using the NUnit attributes in your code

NUnit uses an attribute scheme to recognize and load tests. Just like bookmarks in a book, these attributes help the framework identify the important parts in the assembly that it loads and which parts are tests that need to be invoked.

NUnit provides an assembly that contains these special attributes. You need only to add a reference in your test project (not in your production code!) to the NUnit.Framework assembly. You can find it under the .NET tab in the Add Reference dialog box (you don't need to do this if you've used NuGet to install NUnit). Type `NUnit` and you'll see several assemblies starting with that name.

Add nunit.framework.dll as a reference to your test project (if you've installed it manually and not through NuGet).

The NUnit runner needs at least two attributes to know what to run:

- The `[TestFixture]` attribute that denotes a class that holds automated NUnit tests. (If you replace the word `Fixture` with `Class`, it makes much more sense, but only as a mental exercise. It won't compile if you literally change the code that way.) Put this attribute on top of your new `LogAnalyzerTests` class.
- The `[Test]` attribute that can be put on a method to denote it as an automated test to be invoked. Put this attribute on your new test method.

When you've finished, your test code should look like this:

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

     }
 }
```

TIP NUnit requires test methods to be public, to be void, and to accept no parameters at the most basic configuration, but you'll see that sometimes these tests can also take parameters!

At this point, you've marked your class and a method to be run. Now whatever code you put inside your test method will be invoked by NUnit whenever you want.

2.4 Writing your first test

How do you test your code? A unit test usually comprises three main actions:

1. *Arrange* objects, creating and setting them up as necessary.
2. *Act* on an object.
3. *Assert* that something is as expected.

Here's a simple piece of code that does all three, with the assert part performed by the NUnit framework's `Assert` class:

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

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

     Assert.False(result);
 }
```

Before we go on, you'll need to know a little more about the `Assert` class, because it's an important part of writing unit tests.

2.4.1 The Assert class

The `Assert` class has static methods and is located in the NUnit.Framework namespace. It's the bridge between your code and the NUnit framework, and its purpose is to declare that a specific assumption is supposed to exist. If the arguments that are passed into the `Assert` class turn out to be different than what you're asserting, NUnit will realize the test has failed and will alert you. You can optionally tell the `Assert` class what message to alert you with if the assertion fails.

The `Assert` class has many methods, with the main one being `Assert.True (some Boolean expression)`, which verifies a Boolean condition. But there are many other methods, which you can view as syntactical sugar that make asserting various things cleaner (such as `Assert.False` that we use).

Here's one that verifies that an expected object or value is the same as the actual one:

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

Here's an example:

```
Assert.AreEqual(2, 1+1, "Math is broken");
```

This one verifies that the two arguments reference the same object:

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

Here's an example:

```
Assert.AreSame(int.Parse("1"),int.Parse("1"),
"this test should fail").
```

`Assert` is simple to learn, use, and remember.

Also note that all the assert methods take a last parameter of type "string," which gets displayed in addition to the framework output, in case of a test failure. Please, never, *ever*, use this parameter (it's always optional to use). Just make sure your test name explains what's supposed to happen. Often, people write the trivially obvious things like "test failed" or "expected x instead of y," which the framework already

provides. Much like comments in code, if you have to use this parameter, your method name should be clearer.

Now that we've covered the basics of the API, let's run a test.

2.4.2 Running your first test with NUnit

It's time to run your first test and see if it passes.

There are at least four ways you can run this test:

- Using the NUnit GUI
- Using Visual Studio 2012 Test Runner with an NUnit Runner Extension, called the NUnit Test Adapter in the NUget Gallery
- Using the ReSharper test runner (a well-known commercial plug-in for VS)
- Using the TestDriven.NET test runner (another well-known commercial plug-in for VS)

Although this book covers only the NUnit GUI, I personally use NCrunch, which is fast and runs automatically, but also costs money. (This tool and others are covered in the appendix.) It provides simple, quick feedback inside the Visual Studio Editor window. I find that this runner makes a seamless companion to test-driven development in the real world. You can find out more about it at www.ncrunch.net/.

To run the test with the NUnit GUI, you need to have a build assembly (a .dll file in this case) that you can give to NUnit to inspect. After you build the project, locate the path to the assembly file that was built.

Then, load up the NUnit GUI. (If you installed NUnit manually, find the icon on your desktop. If you installed NUnit.Runners via Nugget, you'll find the NUnit GUI EXE file in the Packages folder under your solution's root directory.) Select File > Open. Enter the name of your test's assembly. You'll see your single test and the class and namespace hierarchy of your project on the left, as shown in figure 2.3. Click the Run button to run your tests. The tests are automatically grouped by namespace (assembly, type name), so you can pick and choose to run only by specific types or namespaces. (You'll usually want to run all of the tests to get better feedback on failures.)

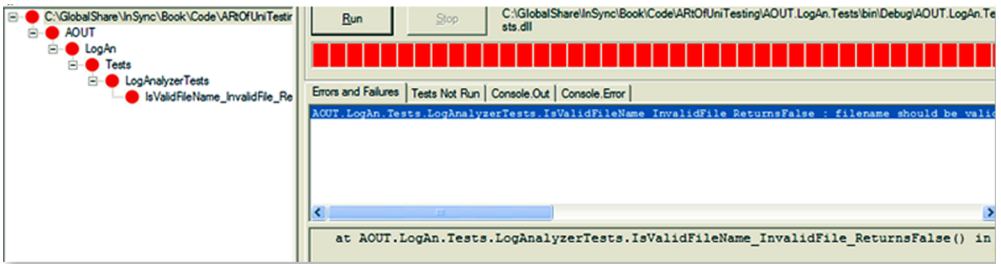

Figure 2.3 NUnit test failures are shown in three places: the test hierarchy on the left becomes red, the progress bar at the top becomes red, and any errors are shown on the right.

You have a failing test, which might suggest that there's a bug in the code. It's time to fix the code and see the test pass. Change the code to add the missing `!` in the `if` clause so that it looks like this:

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

2.4.3 Adding some positive tests

You've seen that bad extensions are flagged as such, but who's to say that good ones do get approved by this little method? If you were doing this in a test-driven way, a missing test here would have been obvious, but because you're writing the tests after the code, you have to come up with good test ideas that will cover all the paths. The following listing adds a couple more tests to see what happens when you send in a file with a good extension. One of them will have uppercase extensions, and another will have lowercase.

Listing 2.1 The `LogAnalyzer` filename-validation logic to test

```
[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);
}
```

If you rebuild the solution now, you'll find that NUnit's GUI can detect that the assembly has changed, and it will automatically reload the assembly in the GUI. If you rerun the tests, you'll see that the test with lowercase extensions fails. You need to fix the production code to use case-insensitive string matching for this test to pass:

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

```
  return true;
}
```

If you run the tests again, they should all pass, and you'll have a nice green bar again in the NUnit GUI.

2.4.4 From red to green: passing the tests

NUnit's GUI is built with a simple idea in mind: all the tests should pass in order to get the green light to go ahead. If even one of the tests fails, you'll see a red light on the top progress bar to let you know that something isn't right with the system (or your tests).

The red/green concept is prevalent throughout the unit testing world and especially in test-driven development. Its mantra is "Red-Green-Refactor," meaning that you start with a failing test, then pass it, and then make your code readable and more maintainable.

Tests can also fail if an unexpected exception suddenly gets thrown. A test that stops because of an unexpected exception will be considered a failed test for most test frameworks, if not all. It's part of the point—sometimes you have bugs in the form of an exception you didn't expect.

Speaking of exceptions, you'll also see later in this chapter a form of test that expects an exception to be thrown from some code, as a specific result or behavior. Those tests will fail if an exception is not thrown.

2.4.5 Test code styling

Notice that the tests I'm writing have several characteristics in terms of styling and readability that look different from "standard" code. The test name can be very long, but the underscores help make sure you don't forget to include all the important pieces of information. Also, notice that there's an empty line between the arrange, act, and assert stages in each test. This helps me read tests much faster and find problems with tests faster.

I also try to separate the assert from the act as much as possible. I'd rather assert on a value than directly against a call to a function. It makes the code much more readable.

Readability is one of the most important aspects when writing a test. As far as possible, it has to read effortlessly, even to someone who's never seen the test before, without needing to ask too many questions—or any questions at all. More on that in chapter 8. Now let's see if you can make these tests less repetitive and a bit more concise, but still readable.

2.5 Refactoring to parameterized tests

All the tests you've written so far suffer from some maintainability problems. Imagine that now you want to add a parameter to the constructor of the `LogAnalyzer` class. Now you'd have three noncompiling tests. Going in and fixing 3 tests might not sound so bad, but it could easily be 30 or 100. When it comes to the real world, developers feel they have better things to do than to start chasing the compiler for what they

thought should be a simple change. If your tests break your sprint, you might not want to run them or even might want to delete annoying tests.

Let's refactor them so that you never come across this problem.

NUnit has a cool feature that can help a lot here. It's called *parameterized tests*. To use them simply take one of the existing test methods that look exactly the same as the others, and do the following:

1. Replace the `[Test]` attribute with the `[TestCase]` attribute.
2. Extract all the hardcoded values the test is using into parameters for the test method.
3. Move the values you had before into the braces of the `[TestCase(param1, param2,..)]` attribute.
4. Rename this test method to a more generic name.
5. Add a `[TestCase(..)]` attribute on this same test method for each of the tests you want to merge into this test method, using the other test's values.
6. Remove the other tests so you're left with just one test method that has multiple `[TestCase]` attributes.

Let's do this step by step. The last test will look like this after step 4:

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

            bool result = analyzer.IsValidLogFileName(file);

            Assert.True(result);
        }
```

TestCase attribute sends a parameter to the method in the next line

The parameter that tests case attributes can attach a value to

The parameter is being used generically.

The parameter sent into the `TestCase` attribute is mapped by the test runner to the first parameter of the test method itself at runtime. You can add as many parameters as you want to the test method and to the `TestCase` attribute.

Now, here's the kicker: you can have *multiple* `TestCase` attributes on the same test method. So after step 6, the test will look like this:

```
[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);
}
```

Another attribute means another test with a different value attached to the method parameter.

And now you can *delete* the previous test method that used a good, lowercase extension because it's encompassed as a test case attribute in the current test method. If you run

the tests again, you'll see you still have the same number of tests, but the code is more maintainable and more readable.

You can take this one step further and include the negative test (the asserts that expect a false value as an end result) into the current test method. I'll show here how to do it, but I'll warn that doing this will likely create a less-readable test method because the name will have to become even *more* generic. Consider this a demo of the syntax, and know that this is possibly taking this technique too far in the right direction, because it makes the tests less understandable without going deeply through the code.

Here's how you can refactor all the tests in the class—by adding another parameter to the test case and test method and by changing the assert to `Assert.AreEqual`:

```
[TestCase("filewithgoodextension.SLF",true)]
[TestCase("filewithgoodextension.slf",true)]
[TestCase("filewithbadextension.foo",false)]
public void
IsValidLogFileName_VariousExtensions_ChecksThem(string file,
                                bool expected)
{
    LogAnalyzer analyzer = new LogAnalyzer();

    bool result = analyzer.IsValidLogFileName(file);

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

Adding another parameter to the test case

Receiving the second test case parameter

Using the second parameter value

With this one test method, you can get rid of all the other test methods in this class, but notice how the name of the test has become so generic that it's hard to figure out what makes the difference between valid and invalid. That information has to be easily self-evident in the parameter values you send in, so you have to keep them as simple as possible and as obvious as possible that these are the simplest values that prove your point. More on that readability objective, again, in chapter 8.

In terms of maintainability, notice how you have only one call to the constructor now. It's better, but it's not good enough, because you can't have just one, big, parameterized test method become all of your tests. More techniques for maintainability later on. (Yes, in chapter 8. You're psychic.)

Another refactoring you can do at this point is to change how the conditional `if` in the production code looks. You can minimize it to a single return statement. If you like that sort of thing, now is a good time to refactor that. I don't. I like a bit of verbosity and not making the reader think too hard about the code. I like code that isn't too smart for its own good, and return statements that contain conditionals rub me the wrong way. But this isn't a book about design, remember? Do what you like. I will refer you to the book's "clean code" by Robert Martin (Uncle Bob) first.

2.6 More NUnit attributes

Now that you've seen how easy it is to create unit tests that run automatically, we'll look at how to set up the initial state for each test and how to remove any garbage that's left by your test.

A unit test has specific points in its lifecycle that you'll want to have control over. Running the test is only one of them, and there are special setup methods that run before each test runs, as you'll see in the next section.

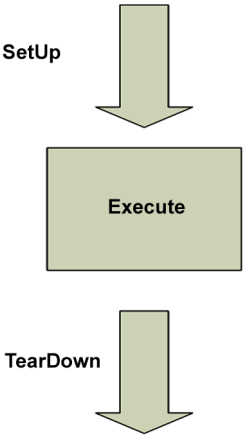

Figure 2.4 NUnit performs setup and teardown actions before and after (respectively) every test method.

2.6.1 Setup and teardown

For unit tests, it's important that any leftover data or instances from previous tests are destroyed and that the state for the new test is recreated as if no tests have been run before. If you have leftover state from a previous test, you might find that your test fails, but only if it's run after a different test and it passes other times. Locating that kind of dependency bug between tests is difficult and time consuming, and I don't recommend it to anyone. Having tests that are totally independent is one of the best practices I'll cover in part 2 of this book.

In NUnit, there are special attributes that allow easier control of setting up and clearing out state before and after tests. These are the `[SetUp]` and `[TearDown]` action attributes. Figure 2.4 shows the process of running a test with setup and teardown actions.

For now, make sure that each test you write uses a new instance of the class under test, so that no leftover state will mess up your tests.

You can take control of what happens in the setup and teardown steps by using two NUnit attributes:

- `[SetUp]`—This attribute can be put on a method, just like a `[Test]` attribute, and it causes NUnit to run that setup method each time it runs any of the tests in your class.
- `[TearDown]`—This attribute denotes a method to be executed once after each test in your class has executed.

Listing 2.2 shows how you can use the `[SetUp]` and `[TearDown]` attributes to make sure that each test receives a new instance of `LogAnalyzer`, while also saving some repetitive typing.

But know that the more you use `[SetUp]`, the less readable your tests will be, because people will have to keep reading test code in two places in the file to understand how the test gets its instances and what type of each object the test is using. I tell my students, "Imagine that the readers of your test have never met you and never will. They arrive and read your tests two years after you've left the company. Every little thing you do to help them understand the code without needing to ask any questions is a big help. They probably have nobody around who can answer those questions, so you're their only hope." Making their eyes jump constantly between two regions of code to understand your test isn't a good idea.

Listing 2.2 Using [SetUp] and [TearDown] attributes

```
using NUnit.Framework;

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

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

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

             Assert.IsTrue(result, "filename should be valid!");
         }

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

            Assert.IsTrue(result, "filename should be valid!");
        }

        [TearDown]                                       <-- A teardown attribute
         public void TearDown()
         {
           //the line below is included to show an anti pattern.
           //This isn't really needed. Don't do it in real life.
            m_analyzer = null;                           <-- A common antipattern—you don't need to do this
         }
   }
```

Think of the setup and teardown methods as constructors and destructors for the tests in your class. You can have only one of each in any test class, and each one will be performed once for each test in your class. In listing 2.2 you have two unit tests, so the execution path for NUnit will be something like that shown in figure 2.5.

In real life I do *not* use setup methods to initialize my instances. I show it here for you to know that it exists and to avoid it. It may seem like a good idea, but soon it makes the tests below the setup method harder to read. Instead, I use factory methods to initialize my instances under test. Read about that in chapter 7.

NUnit contains several other attributes to help with setup and cleanup of state. For example, [TestFixtureSetUp] and [TestFixtureTearDown] allow setting up state once before all the tests in a specific *class* run and once after all the tests have been run (once per test fixture). This is useful when setting up or cleaning up takes a long time, and you want to do it only once per fixture. You'll need to be cautious about

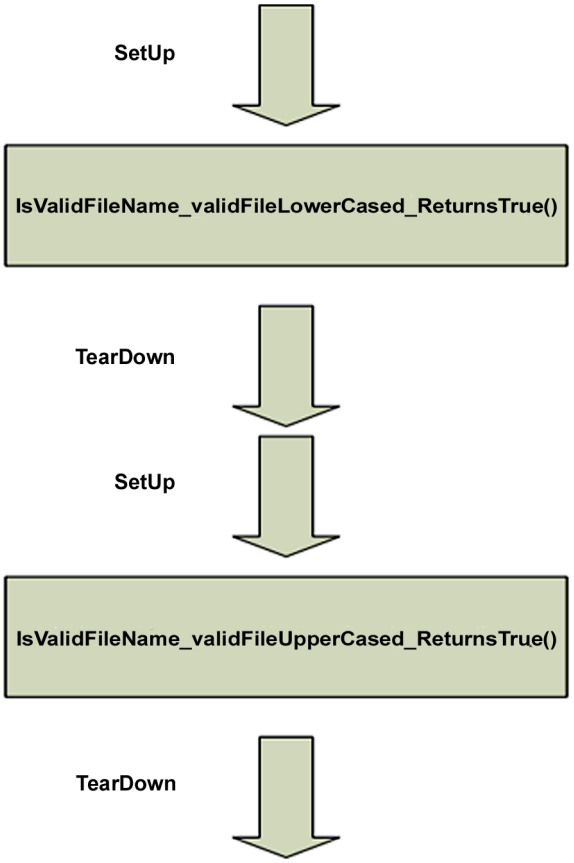

Figure 2.5 How NUnit calls `SetUp` and `TearDown` with multiple unit tests in the same class: each test is preceded by running `SetUp` and followed by a `TearDown` method run.

using these attributes. You may find that you're sharing state between tests if you're not careful.

You almost *never, ever* use `TearDown` or `TestFixture` methods in unit test projects. If you do, you're very likely writing an integration test, where you're touching the filesystem or a database, and you need to clean up the disk or the DB after the tests. The only time it makes sense to use a `TearDown` method in unit tests, I've found, is when you need to "reset" the state of a static variable or singleton in memory between tests. Any other time, you're likely doing integration tests. That's not a bad thing to be doing, but you should be doing it in a separate project that's dedicated to integration tests.

Next, we'll look at how you can test that an exception is thrown by your code when it should be.

2.6.2 Checking for expected exceptions

One common testing scenario is making sure that the correct exception is thrown from the tested method when it should be.

Let's assume that your method should throw an `ArgumentException` when you send in an empty filename. If your code doesn't throw an exception, it means your test should fail. We'll test the method logic in the following listing.

Listing 2.3 The `LogAnalyzer` filename-validation logic to test

```
public class LogAnalyzer
    {
        public bool IsValidLogFileName(string fileName)
        {
            …
            if (string.IsNullOrEmpty(fileName))
            {
                throw new ArgumentException(
                    "filename has to be provided");
            }
            …
        }
    }
```

There are two ways to check for this. Let's start with the one you shouldn't use, because it's very prevalent, and because it used to be the only API to achieve this. There's a special attribute in NUnit that helps you test exceptions: the `[ExpectedException]` attribute. Here's what a test that checks for the appearance of an exception might look like:

```
[Test]
[ExpectedException(typeof(ArgumentException),
  ExpectedMessage ="filename has to be provided")]
public void IsValidFileName_EmptyFileName_ThrowsException()
{
    m_analyzer.IsValidLogFileName(string.Empty);
}

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

There are several important things to note here:

- The expected exception message is provided as a parameter to the `[ExpectedException]` attribute.
- There's no `Assert` call in the test itself. The `[ExpectedException]` attribute contains the assert within it.
- There's no point getting the value of the Boolean result from the method because the method call is supposed to trigger an exception.

Not related to this example, I've gone ahead and extracted the code that creates the instance of `LogAnalyzer` into a factory method. I use this factory method in all my tests, so that maintainability of the constructor is easier without needing to fix many tests.

Given the method in listing 2.3 and the test for it, this test should pass. Had your method *not* thrown an `ArgumentException`, or had the exception's message been different than the one expected, the test would have failed—saying either that an exception was not thrown or that the message was different than expected.

So why did I mention that you shouldn't use this method? Because this attribute basically tells the test runner to wrap the execution of this whole method in a big `try-catch` block and fail the test if nothing was "catch"-ed. The big problem with this is that you don't know *which* line threw the exception. In fact, you could have a bug in the constructor that throws an exception, and your test will pass, even though the constructor should never have thrown this exception! The test could be lying to you when you use this attribute, so try not to use it.

Instead, there's a newer API in NUnit: `Assert.Catch<T>(delegate)`. Here's the test rewritten to use `Assert.Catch` instead:

```
[Test]                                                    <-- No ExpectedException attribute needed
public void IsValidFileName_EmptyFileName_Throws()
{
            LogAnalyzer la = MakeAnalyzer();                          <-- Using Assert.Catch
   var ex =
            Assert.Catch<Exception>(() => la.IsValidLogFileName(""));

   StringAssert.Contains("filename has to be provided",
                                          ex.Message);    <-- Using the Exception object returned by Assert.Catch
}
```

There are a lot of changes here:

- You no longer have the `[ExpectedException]` attribute.
- You use `Assert.Catch` and use a lambda expression that takes no arguments, whose body is the call to `la.IsValidLogFileName("")`.
- If that code inside the lambda throws an exception, the test will pass. If any other line of code not in the lambda throws an exception, the test will fail.
- `Assert.Catch` is a function that returns the instance of the exception object that was thrown inside the lambda. This allows you to later assert on the message of that exception object.
- You're using `StringAssert`—a class that's part of NUnit framework you haven't been introduced to yet. It's contains helpers that make testing with strings simpler and more readable.
- You're not asserting full string equality with `Assert.AreEqual` but use `StringAssert.Contains`. The string message *contains* a string you're looking for. This makes the test more maintainable, because strings are notorious about changing over time, when new features are added. Strings are a form of UI, really, so they can have extra line breaks, extra information you don't care about, and so on. If you asserted on the whole string being equal to a specific string you expect, you'd have to fix this test every time you added a new feature to the

beginning or end of the message that you don't care about in this test (like extra lines or something that benefits user formatting).

This test is less likely to "lie" to you, and I recommend using `Assert.Catch` over `[ExpectedException]`.

There are other ways to use NUnit's fluent syntax to check the exception message. I don't like them much, but it's more a matter of style. Look up NUnit's fluent syntax at NUnit.com to learn other ways.

2.6.3 Ignoring tests

Sometimes you'll have tests that are broken, and you still need to check in your code to the main source tree. In those rare cases (and they should be rare!), you can put an `[Ignore]` attribute on tests that are broken because of a problem in the test, not in the code.

It can look like this:

```
[Test]
[Ignore("there is a problem with this test")]
 public void IsValidFileName_ValidFile_ReturnsTrue()
 {
 /// ...
 }
```

Running this test in the NUnit GUI will produce a result like that shown in figure 2.6.

What happens when you want to have tests running not by a namespace but by some other type of grouping? That's where test categories come in. I explain them in section 2.6.5.

2.6.4 NUnit's fluent syntax

NUnit also has an alternative, more fluent syntax that you can use instead of calling simple `Assert.*` methods. The fluent syntax always starts with `Assert.That(..)`.

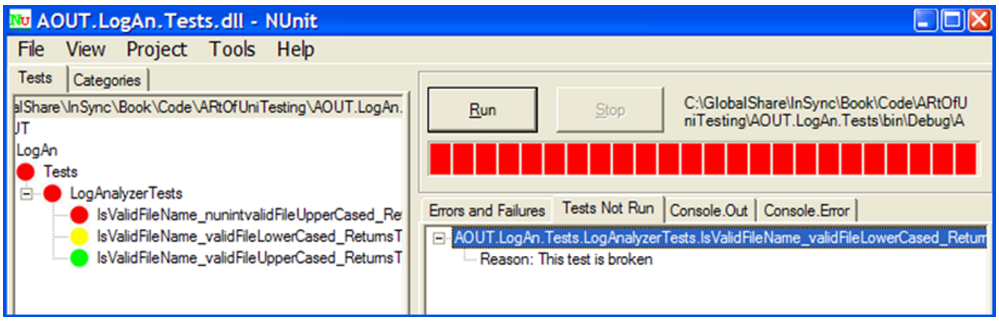

Figure 2.6 In NUnit, an ignored test is marked in yellow (the middle test), and the reason for not running the test is listed under the Tests Not Run tab on the right.

Here's the last test rewritten with NUnit fluent syntax:

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

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

     Assert.That(ex.Message,
            Is.StringContaining("filename has to be provided"));
 }
```

Personally I like the terser, simpler, and shorter syntax of `Assert.something()` than `Assert.That`. Although fluent syntax seems friendlier at first, it takes longer to understand what you're testing for (all the way at the end of the line). Choose as you like, but make sure you're consistent with your choice across the test project, because lack of consistency leads to many readability issues.

2.6.5 Setting test categories

You can set up your tests to run under specific test categories, such as slow tests and fast tests. You do this by using NUnit's `[Category]` attribute:

```
[Test]
[Category("Fast Tests")]
 public void IsValidFileName_ValidFile_ReturnsTrue()
 {
 /// ...
 }
```

When you load your test assembly again in NUnit, you can see it organized by categories instead of namespaces. Switch to the Categories tab in NUnit, and double-click the category you'd like to run so that it moves into the lower Selected Categories pane. Then click the Run button. Figure 2.7 shows what the screen might look like after you select the Categories tab.

So far, you've run simple tests against methods that return some value as a result. What if your method doesn't return a value but changes some state in the object?

2.7 Testing results that are system state changes instead of return values

Up until this section, you've seen how to test for the first, simplest kind of result a unit of work can have: return values (explained in chapter 1). Here and in the next chapter we'll also discuss the second type of result: system state change—checking that the system's behavior is different after performing an action on the system under test.

> **DEFINITION** *State-based testing* (also called *state verification*) determines whether the exercised method worked correctly by examining the changed behavior of the system under test and its collaborators (dependencies) after the method is exercised.

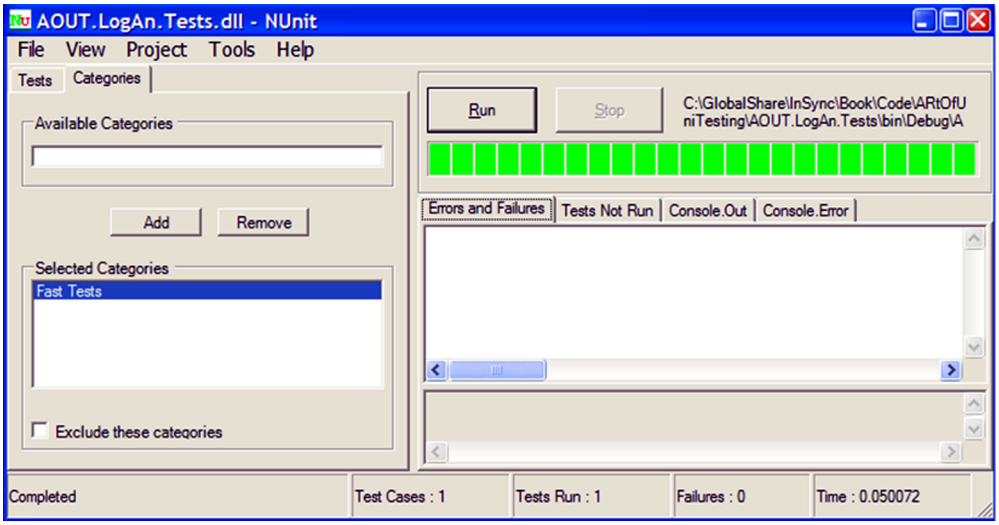

Figure 2.7 You can set up categories of tests in the code base, and then choose a particular category to be run from the NUnit GUI.

If the system acts exactly the same as it did before, then you didn't really change its state, or there's a bug.

If you've read other definitions of state-based testing elsewhere, you'll notice that I define it differently. That is because I view this in a slightly different light—that of test maintainability. Simply testing direct state (sometimes externalizing it to make it testable) is something I wouldn't usually endorse, because it leads to less-maintainable and less-readable code.

Let's consider a simple state-based testing example using the `LogAnalyzer` class, which you can't test simply by calling one method in your test. Listing 2.4 shows the code for this class. In this case, you introduce a new property, `WasLastFileNameValid`, that should keep the last success state of the `IsValidLogFileName` method. Remember, I'm showing the code first, because I'm not trying to teach you TDD here, but how to write good tests. Tests *could* become better by TDD, but that's a step you take when you know how to write tests *after* the code.

Listing 2.4 Testing the property value by calling `IsValidLogFileName`

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

    public bool IsValidLogFileName(string fileName)
    {
        WasLastFileNameValid = false;

        if (string.IsNullOrEmpty(fileName))
        {
            throw new ArgumentException("filename has to be provided");
```

Changes the state of the system

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

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

Changes the state of the system

As you can see in this code, `LogAnalyzer` remembers the last outcome of a validation check. Because `WasLastFileNameValid` depends on having another method invoked first, you can't simply test this functionality by writing a test that gets a return value from a method; you have to use alternative means to see if the logic works.

First, you have to identify the unit of work you're testing. Is it in the new property called `WasLastFileNameValid`? Partly. It's also in the `IsValidLogFileName` method, so your test should start with the name of that method because that's the unit of work you invoke publicly to change the state of the system. The following listing shows a simple test to see if the outcome is remembered.

Listing 2.5 Testing a class by calling a method and checking the value of a property

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

            la.IsValidLogFileName("badname.foo");

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

Asserts on state of the system

Notice that you're testing the functionality of the `IsValidLogFileName` method by asserting against code in a different location than the piece of code under test.

Here's a refactored example that adds another test for the opposite expectation of the system state:

```
[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);
}
```

The next listing shows another example. This one looks into the functionality of a built-in memory calculator.

Listing 2.6 The `Add()` and `Sum()` methods

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

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

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

The `MemCalculator` class works a lot like the pocket calculator you know and love. You can click a number, then click Add, then click another number, then click Add, and so on. When you've finished, you can click Equals and you'll get the total so far.

Where do you start testing the `Sum()` function? You should always consider the simplest test to begin with, such as testing that `Sum()` returns `0` by default. This is shown in the following listing.

Listing 2.7 The simplest test for a calculator's `Sum()`

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

       int lastSum = calc.Sum();

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

Asserts on default return value

Also note the importance of the name of the method here. You can read it like a sentence.

Here's a simple list of naming conventions of scenarios I like to use in such cases:

- `ByDefault` can be used when there's an expected return value with no prior action, as shown in the previous example.
- `WhenCalled` or `Always` can be used in the second or third kind of unit of work results (change state or call a third party) when the state change is done with no prior configuration or when the third-party call is done with no prior configuration; for example, `Sum_WhenCalled_CallsTheLogger` or `Sum_Always_CallsTheLogger`.

You can't write any other test without first invoking the `Add()` method, so the next test will have to call `Add()` and assert against the number returned from `Sum()`. The next listing shows the test class with this new test.

Listing 2.8 Two tests, with the second one calling the `Add()` method

```
[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);
}

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

The system's behavior and state change if sum returns a different number in this test

Notice that this time you use a factory method to initialize `MemCalculator`. This is a good idea, because it saves time writing the tests, makes the code inside each test smaller and a little more readable, and makes sure `MemCalculator` is always initialized the same way. It's also better for test maintainability, because if the constructor for `MemCalculator` changes, you only need to change the initialization in one place instead of going through each test and changing the `new` call.

So far, so good. But what happens when the method you're testing depends on an external resource, such as the filesystem, a database, a web service, or anything else that's hard for you to control? And how do you test the third type of result for a unit of work—a call to a third party? That's when you start creating test stubs, fake objects, and mock objects, which are discussed in the next few chapters.

2.8 Summary

In this chapter, we looked at using NUnit to write simple tests against simple code. You used the `[TestCase]`,`[SetUp]`, and `[TearDown]` attributes to make sure your tests always use new and untouched state. You used factory methods to make this more maintainable. You used `[Ignore]` to skip tests that need to be fixed. Test categories can help you group tests in a logical way rather than by class and namespace, and `Assert.Catch()` helps you make sure your code throws exceptions when it should. We also looked at

what happens when you aren't facing a simple method with a return value, and you need to test the end state of an object.

This isn't enough, though. Most test code has to deal with far more difficult coding issues. The next couple of chapters will give you additional basic tools for writing unit tests. You'll need to pick and choose from these tools when you write tests for various difficult scenarios you'll come across.

Finally, keep the following points in mind:

- It's common practice to have one test class per tested class, one unit test project per tested project (aside from an integration tests project for that tested project), and at least one test method per unit of work (which can be as small as a method or as large as multiple classes).
- Name your tests clearly using the following model: `[UnitOfWork]_[Scenario]_[ExpectedBehavior]`.
- Use factory methods to reuse code in your tests, such as code for creating and initializing objects all your tests use.
- Don't use `[SetUp]` and `[TearDown]` if you can avoid them. They make tests less understandable.

In the next chapter, we'll look at more real-world scenarios, where the code to be tested is a little more realistic than what you've seen so far. It has dependencies and testability problems, and we'll start discussing the notion fakes, mocks, and stubs, and how you can use them to write tests against such code.

Part 2

Core techniques

Having covered the basics in previous chapters, I'll now introduce the core testing and refactoring techniques necessary for writing tests in the real world.

In chapter 3, we'll begin by examining stubs and how they help break dependencies. We'll go over refactoring techniques that make code more testable, and you'll learn about seams in the process.

In chapter 4, we'll move on to mock objects and interaction testing and look at how mock objects differ from stubs, and we'll explore the concept of fakes.

In chapter 5, we'll look at isolation frameworks (also known as mocking frameworks) and how they solve some of the repetitive coding involved in handwritten mocks and stubs. Chapter 6 also compares the leading isolation frameworks in .NET and uses FakeItEasy for examples, showing its API in common use cases.