



Test hierarchies and organization

This chapter covers

- Running unit tests during automated nightly builds
- Using continuous integration for automated builds
- Organizing tests in a solution
- Exploring test class inheritance patterns

Unit tests are as important to an application as the production source code. As with the regular code, you need to give careful thought to where the tests reside, both physically and logically, in relation to the code under test. If you put unit tests in the wrong place, the tests you've written so carefully may not be run.

Similarly, if you don't devise ways to reuse parts of your tests, create utility methods for testing, or use test hierarchies, you'll end up with test code that's either unmaintainable or hard to understand.

This chapter addresses these issues with patterns and guidelines that will help shape the way your tests look, feel, and run and will affect how well they play with the rest of your code and with other tests.

Where the tests are located depends on where they'll be used and who'll run them. There are two common scenarios: tests run as part of the automated build process and tests run locally by developers on their own machines. The automated build process is very important, and that's what we'll focus on next.

7.1 Automated builds running automated tests

The power of the automated build process shouldn't be ignored. I've been automating my build and delivery process for over a decade, and it's one of the best things you can do to make your team more productive and get feedback faster. If you plan to make your team more agile and equipped to handle requirement changes as they come into your shop, you need to be able to do the following:

- Make a small change to your code.
- Run all the tests to make sure you haven't broken any existing functionality.
- Make sure your code can still integrate well and not break any other projects you depend on.
- Create a deliverable package of your code and deploy it automatically at the push of a button.

You'll likely need several types of build configurations and build scripts to accomplish these tasks. Build scripts are small pieces of script that reside alongside your code in source control and are fully version aware, because they live in source control with your product source code. They get invoked by a continuous integration server's build configuration.

Some of those build scripts will run your tests, especially the ones that will run immediately after you check your code in to source control. Running those tests lets you know whether you've broken any existing or new functionality, for yourself or for anyone else on the project. You're integrating your code with other projects. Your tests will indicate whether you broke the compilation of the code or things that are logically dependent on your code. By doing this automatically upon check-in, you're starting a process commonly known as continuous integration. I'll discuss what that means in section 7.1.2.

If you were to personally integrate your code, it would usually mean the following:

- Getting the latest version of everyone's source code from the source control repository
- Trying to compile it all locally
- Running all tests locally
- Fixing anything that has been broken
- Checking in your source code

You can use tools to automate this work, in the form of automated build scripts and continuous integration servers.

An automated build process combines all these steps under a single logical umbrella that can be thought of as "how we release code here." This build process is a collection

of build scripts, automated triggers, a server, possibly some build agents (which do the work), and a shared team agreement to work this way.

The agreement involves making sure everyone accepts and adheres to the warnings and required steps needed to make all this work, continuously and as automatically as relevantly possible (it might not be relevant to automatically deploy to production without a human watching over the process).

If anything breaks in the process, the build server can notify the relevant parties of a *build break*.

To clarify: a build process is a logical concept, encompassing build scripts, build integration servers, build triggers, and a shared team understanding and acceptance of how code is deployed and integrated.

7.1.1 Anatomy of a build script

I usually end up with several single-purpose build scripts. That kind of setup allows for better maintenance and coherency of the build process, and would include these scripts:

- A continuous integration (CI) build script
- A nightly build script
- A deployment build script

I like to separate them because I treat build scripts like small code functions that can be called with parameters and the current version of source code. The caller of these functions (scripts) is the CI server.

A CI build script will usually, at the very least, compile the current sources in debug mode and run all the unit tests. Potentially it will also run other tests, as long as they're fast. A CI build script is meant to give maximum information in the least amount of time. The quicker it is, the quicker you know you likely didn't break anything and can get back to work.

A nightly build will usually take longer. I like to trigger it just after a CI build, to get even more feedback, but I won't be waiting too eagerly for it and can continue coding while it's running. It takes longer because it's meant to do all the tasks that the CI build considered irrelevant or not important enough to be included in a quick feedback cycle of CI. These tasks can include almost anything but usually include compilation in release mode, running all the slow tests, and possibly deploying to test environments for the next day.

I call them nightly builds, but they can be run many times a day. At the very least, they run once a night. They give more feedback but take more time to give it.

A deployment build script is essentially a delivery mechanism. It's triggered by the CI server and can be as simple as an xcopy to a remote server or as complicated as deploying to hundreds of servers, reinitializing Azure or Amazon Elastic Compute Cloud (EC2) instances, and merging databases.

All builds usually notify the user by email if they break, but the ultimate required destination of notification is the caller of the build scripts: the CI server.

There are many tools that can help you create an automated build system. Some are free or open source, and some are commercial. Following are a few tools you can consider.

For build scripts:

- NAnt (nant.sourceforge.net)
- MSBuild (www.infoq.com/articles/MSBuild-1)
- FinalBuilder (www.FinalBuilder.com)
- Visual Build Pro (www.kinook.com)
- Rake (<http://rake.rubyforge.org/>)

For CI servers:

- CruiseControl.NET (cruisecontrol.sourceforge.net)
- Jenkins (<http://jenkins-ci.org/>)
- Travis CI (<http://about.travis-ci.org/docs/user/getting-started/>)
- TeamCity (JetBrains.com)
- Hudson (<http://hudson-ci.org/>)
- Visual Studio Team Foundation Service (<http://tfs.visualstudio.com/>)
- ThoughtWorks Go (www.thoughtworks-studios.com/go-agile-release-management)
- CircleCI (<https://circleci.com/>) if you work exclusively through github.com
- Bamboo (www.atlassian.com/software/bamboo/overview)

Some CI servers also allow creating build script-related tasks as a built-in feature. I try to stay away from using those features, because I want my build script actions to be version aware (or version controlled), so I can always get back to any version of the source and my build actions will be relevant to that version.

Of these tools, my two favorites are FinalBuilder for build scripts and TeamCity for CI servers. If I weren't able to use FinalBuilder (which is Windows only), I'd use Rake, because I despise the use of XML for build management. It makes the build scripts very hard to maintain. Rake is XML free, whereas MSBuild or NAnt will force so much XML down your throat you'll be dreaming of XML tags in your sleep for a few months. Each tool on these lists excels at doing one thing really well, though TeamCity has been trying to add more and more built-in tasks, which I think drives people to create less-maintainable builds.

7.1.2 Triggering builds and integration

We briefly discussed CI before, but let's do it a bit more officially. The term *continuous integration* is literally about making the automated build and integration process run continuously. You could have a certain build script run every time someone checks in source code to the system, or every 45 minutes, or when another build script has finished running, for example.

A CI server's main jobs are these:

- Trigger a build script based on specific events
- Provide build script context and data such as version, source code, and artifacts from other builds, build script parameters, and so on

- Provide an overview of build history and metrics
- Provide the current status of all the active and inactive builds

First, let's investigate triggers. A trigger can start a build script automatically when certain events occur, such as source control updates, time passing, or another build configuration failing or succeeding. You can configure multiple triggers to start a specific unit of work in the CI server. These units of work are often called build configurations.

A build configuration will have commands that it executes, such as executing a command line, compiling, and so on. I would advise limiting those to an executable, which runs a build script, kept in source control, to maximize action compatibility with the current source version. For example, in TeamCity, when creating a build configuration, you can then add build steps to that configuration. A build step can be of several kinds. Running a DOS command line is one of those types. Another might be to compile a .NET .sln file. I stick with a simple command-line build step, and in that command line I execute a batch file or a build script that's in the checkout source code on the build agent.

A build configuration can have context. This can include many things, but usually it includes a current snapshot of the source code from source control. It might also include setting up environment variables that the build script uses or direct parameters via the command line. A context can also include copying artifacts from previous or different build configurations. Artifacts are the end results of running a build script. They could be binary files, configuration files, or any type of file.

A build configuration can have history. You can see when it ran, how long it took, and the last time it passed. You might also see how many tests were run and which tests failed. The details of the history depend on the CI server.

A CI server will usually have a dashboard showing the current status of the builds. Some servers may even provide custom HTML and JavaScript you can embed on your own company's internal intranet pages to see the status in a customized way. Some CI servers provide integration or custom tools that run on the desktop that continuously monitor build status and notify you if builds you care about have broken.

More info on build automation

There are plenty more good build practices you might want to hear about, but they're not the focus of this book. If you want to read more about continuous delivery, I recommend *Continuous Delivery* by Jez Humble and David Farley (Addison-Wesley Professional, 2010), and *Continuous Integration* by Paul Duvall, Steve Matyas, and Andrew Glover (Addison-Wesley Professional, 2007). You might also be interested in my own book on the subject, called *Beautiful Builds*. *Beautiful Builds* is my attempt to create a pattern language of common build process solutions and problems. It resides at www.BeautifulBuilds.com.

7.2 Mapping out tests based on speed and type

It's easy to run the tests to check their run times and to determine which are integration tests and which are unit tests. Once you do, put them in different places. They don't need to be in separate test projects; a separate folder and namespace should be enough.

Figure 7.1 shows a simple folder structure you can use inside your Visual Studio projects.

Some companies, based on the build software and unit testing framework they use, find it easier to use separate test projects for unit and integration tests. This makes it easier to use command-line tools that accept and run a full test assembly containing only specific kinds of tests. Figure 7.2 shows how you'd set up two separate kinds of test projects under a single solution.

Even if you haven't already implemented an automated build system, separating unit from integration tests is a good idea. Mixing up the two tests can lead to severe consequences, such as people not running your tests, as you'll see next.

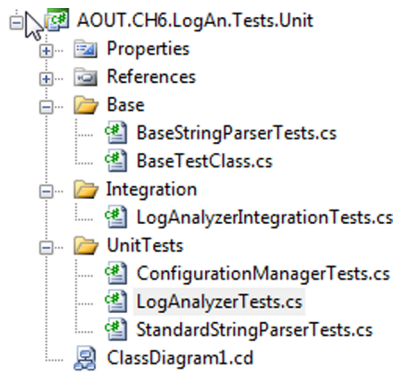


Figure 7.1 Integration tests and unit tests can reside in different folders and namespaces but remain under the same project. Base classes have their own folders.

7.2.1 The human factor when separating unit from integration tests

I recommend separating unit from integration tests. If you don't, there's a big risk people won't run the tests often enough. If the tests exist, why wouldn't people run them as often as needed? One reason is that developers can be lazy or under tremendous time pressure.

If a developer gets the latest version of the source code and finds that some unit tests fail, there are several possible causes:

- There's a bug in the code under test.
- The test has a problem in the way it's written.
- The test is no longer relevant.
- The test requires some configuration to run.

All but the last point are valid reasons for a developer to stop and investigate the code. The last one isn't a development issue; it's a configuration problem, which is often considered less important because it gets in the way of running the tests. If such a test fails, the developer will often ignore the test failure and go on to other things. (They have "more important" things to do.)

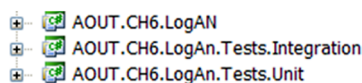


Figure 7.2 The unit testing and integration projects are unique for the LogAn project and have different namespaces.

In many ways, having such hidden integration tests mixed in with unit tests and scattered around your test project with unknown or unexpected configuration requirements (like a database connection) is bad form. These tests are less approachable, they waste time and money on finding problems that aren't there, and they generally discourage the developer from trusting the set of tests again. Like bad apples in a basket, they make all the others look bad. The next time something similar happens, the developer may not even look for a cause for the failure and may simply say, "Oh, that test sometimes fails; it's OK."

To make sure this doesn't happen, you can create a safe green zone.

7.2.2 The safe green zone

Locate your integration and unit tests in separate places. By doing that, you give the developers on your team a safe green test area that contains only unit tests, where they know that they can get the latest code version, they can run all tests in that namespace or folder, and the tests should all be green. If some tests in the safe green zone don't pass, there's a real problem, not a (false positive) configuration problem in the test.

This doesn't mean that the integration tests shouldn't all pass. But because integration tests inherently take longer to execute, it's more likely that developers will run the unit tests more times a day and run the integration tests less often but at least during the nightly build. Developers can focus on being productive and getting at least a partial sense of confidence when all their unit tests are passing. The nightly build should have all the automated tasks of getting everything to work to make the integration tests pass.

In addition, creating a separate integration zone (the opposite of a safe green zone) for the integration tests gives you not only a place to quarantine tests that may run slowly but also a place to put documents detailing what configuration needs to take place to make all these tests work.

An automated build system will do all the configuration work for you. But if you want to run locally, you should have in your solution or project an integration zone that has all the information you need to make things run but that you can also skip if you want to just run the quick tests (in the safe green zone).

But none of this matters if you don't have your tests inside the source control tree, as you'll see next.

7.3 Ensuring tests are part of source control

Tests must be part of source control. The test code that you write needs to reside in a source control repository, just like your real production code. In fact, you should treat your test code as thoughtfully as you treat your production code. It should be part of the branch for each version of the product, and it should be part of the code that developers receive automatically when they get the latest version.

Because unit tests are so connected to the code and API, they should always stay attached to the version of the code they're testing. Obtaining version 1.0.1 of your

product means also getting version 1.0.1 of the tests for your product; version 1.0.2 of your product and its tests will be different.

Also, having your tests as part of the source control tree is what allows your automated build processes to consistently run the correct version of the tests against your software.

So now that tests are part of source control, where should they reside?

7.4 **Mapping test classes to code under test**

When you create test classes, the way they're structured and placed should allow you to easily do the following:

- Look at a project and find all the tests that relate to it
- Look at a class and find all the tests that relate to it
- Look at a method and find all the tests that relate to it

There are several patterns that can help you do this. We'll examine these goals one by one.

7.4.1 **Mapping tests to projects**

I like to create a project to contain the tests and give it the same name as the project under test, adding `.UnitTests` to the end of the name. For example, if I had a project named `Osherove.MyLibrary`, I would also have a test project named `Osherove.MyLibrary.UnitTests` as well as `Osherove.MyLibrary.IntegrationTests`, or some variation on this idea. (See figure 7.2 for an example.) This may sound crude, but it's intuitive, and it allows a developer to find all the tests for a specific project.

You may also want to use Visual Studio's ability to create folders under the solution and group this threesome into its own folder, but that's a matter of preference.

7.4.2 **Mapping tests to classes**

There are several ways to go about mapping the tests for a class you're testing. We'll look at two main scenarios: having one test class for each class under test and having separate test classes for complex methods being tested.

TIP These are the two test class patterns I use most, but others exist. I suggest you look at Gerard Meszaros's *xUnit Test Patterns: Refactoring Test Code* for more.

ONE TEST CLASS PER CLASS OR UNIT OF WORK UNDER TEST

You want to be able to quickly locate all tests for a specific class, and the solution is much like the previous pattern for projects: take the name of the class you want to write tests for and, in the test project, create a test class with the same name postfixed with `UnitTests`. For a class called `LogAnalyzer`, you'd create a test class in your test project named `LogAnalyzer.UnitTests`.

Note the plural; this is a class that holds multiple tests for the class under test, not just one test. It's important to be accurate. Readability and language matter a lot when

it comes to test code, and once you start cutting corners in one place, you'll be doing so in others, which can lead to problems.

The one-test-class-per-class pattern (also mentioned in Meszaros's *xUnit Test Patterns: Refactoring Test Code*) is the simplest and most common pattern for organizing tests. You put all the tests for all methods of the class under test in one big test class. When you're using this pattern, some methods in the class under test may have so many tests that the test class becomes difficult to read or browse. Sometimes the tests for one method drown out the other tests for other methods. That in itself could indicate that maybe the method test is doing too much.

TIP Test readability is important. You're writing tests as much for the person who will read them as for the computer that will run them. I cover readability aspects in the next chapter.

If the person reading the test has to spend more time browsing the test code than understanding it, the test will cause maintenance headaches as the code gets bigger and bigger. That's why you might think about doing it differently.

ONE TEST CLASS PER FEATURE

An alternative is creating a separate test class for a particular feature (which could be as small as a method). The one-test-class-per-feature pattern is also mentioned in Meszaros's book. If you seem to have lots of test methods that make your test class difficult to read, find the method or group of methods whose tests are drowning out the other tests for that class, and create a separate test class for it, with the name relating to the feature.

Suppose a class named `LoginManager` has a `ChangePassword` method you'd like to test, but it has so many test cases that you want to put it in a separate test class. You might end up with two test classes: `LoginManagerTests`, which contains all the other tests, and `LoginManagerTestsChangePassword`, which contains only the tests for the `ChangePassword` method.

7.4.3 Mapping tests to specific unit of work method entry points

Beyond making test names readable and understandable, your main goal is to be able to easily find all test methods for a specific unit of work under test, so you should give your test methods meaningful names. You can use the starting public method name as part of the test name.

You could name a test `ChangePassword_scenario_expectedbehavior`. This naming convention is discussed in chapter 2 (section 2.3.2). There are times in your production code you won't want to use the injection techniques specified in the previous chapters, such as extracting interfaces or overriding virtual methods. That happens when you're dealing with cross-cutting concerns.

7.5 Cross-cutting concerns injection

When you're dealing with cross-cutting concerns such as time management, or exceptions, or logging, you might end up with code that's less readable and maintainable when using these techniques.

The problem with cross-cutting concerns like `DateTime` is that when they exist in your app, they're used in so many places that architecting them as injectable pieces of Lego can end up making your code very testable but also very hard to read and follow.

Let's say that your application needs the current time for scheduling or for logging, and you'd also like to test that your application is using the current time in its logs.

You might have this type of code in your system:

```
public static class TimeLogger
{
    public static string CreateMessage(string info)
    {
        return DateTime.Now.ToShortDateString() + " " + info;
    }
}
```

If you were to make it more testable by making an `ITimeProvider` interface, you'd then have to use this interface everywhere `DateTime` is used. This is very time consuming, when in fact you can have more straightforward approaches.

The approach I like to use for time-based systems is to create a custom class, named `SystemTime`, and make sure all my production code uses that class instead of the standard built-in `DateTime`.

That class and the revised production code that uses it might look like the following listing.

Listing 7.1 Using the `SystemTime` class

```
public static class TimeLogger
{
    public static string CreateMessage(string info)
    {
        return SystemTime.Now.ToShortDateString() + " " + info;
    }
}

public class SystemTime
{
    private static DateTime _date;

    public static void Set(DateTime custom)
    {
        _date = custom;
    }

    public static void Reset()
    {
        _date = DateTime.MinValue;
    }

    public static DateTime Now
    {
        get
    }
}
```

Production code that uses `SystemTime`

`SystemTime` allows changing the current time...

...and resetting the current time

`SystemTime` returns real time or fake one if it was set

```

    {
        if (_date != DateTime.MinValue)
        {
            return _date;
        }
        return DateTime.Now;
    }
}

```

The simple trick here is that there are special functions on the `SystemTime` class that allow you to alter the current time throughout the system. That is, everyone who uses this `SystemTime` class will see whatever date and time you choose.

This gives you a perfect way to test that the current time is used in your production code through a simple test like the one in the next listing.


Listing 7.2 A test using `SystemTime`


```

[TestFixture]
public class TimeLoggerTests
{
    [Test]
    public void SettingSystemTime_Always_ChangesTime()
    {
        SystemTime.Set(new DateTime(2000,1,1));
        string output = TimeLogger.CreateMessage("a");
        StringAssert.Contains("01.01.2000", output);
    }

    [TearDown]
    public void afterEachTest()
    {
        SystemTime.Reset();
    }
}

```


Set fake date


Reset date at end of each test

As a bonus, you don't need to inject a million interfaces into your app. The price you pay is a simple `[TearDown]` method in your test class that makes sure any test doesn't change the time for other tests.

But you need to take into account that the system's current culture (en-US versus en-GB, for example) can change the output string. In that case, you can also include a `CultureInfoAttribute`, in NUnit, on the test to force the test to run under a specific culture.

This type of external abstraction of a cross-cutting concern allows you to create a fake focal point in your production code instead of many small ones. But it only makes sense for things that are used throughout the system. If you use this for everything, you end up with a system that might be just as hard to read as what you're trying to avoid.

A question many developers ask me when I point out this example is, “How do we make sure everyone uses this class?” My answer is that I do code reviews, and in them I make sure nobody uses `DateTime` directly. I try not to rely on tools too much, because I believe true learning happens when two people (or more) are sitting close enough to hear and see each other and can work together and take turns working with the same keyboard to talk about code. But if this is an existing project that we’re converting to use `SystemTime`, I simply do a “find in files” for code that uses `DateTime`, and if possible, I simply do a “replace” on all the things I find. `SystemTime` is named so that it’s easy to find and replace.

Next, we’ll discuss building a test API for your application.

7.6 *Building a test API for your application*

Sooner or later, as you start writing tests for your applications, you’re bound to refactor them and create utility methods, utility classes, and many other constructs (either in the test projects or in the code under test) solely for the purpose of testability or test readability and maintenance.

Here are some things you may want to do:

- Use inheritance in your test classes for code reuse, guidance, and more.
- Create test utility classes and methods.
- Make your API known to developers.

Let’s look at these in turn.

7.6.1 *Using test class inheritance patterns*

One of the most powerful arguments for object-oriented code is that you can reuse existing functionality instead of recreating it over and over again in other classes—what Andy Hunt and Dave Thomas called the DRY (“don’t repeat yourself”) principle in *The Pragmatic Programmer* (Addison-Wesley Professional, 1999). Because the unit tests you write in .NET and most object-oriented languages are in an object-oriented paradigm, it’s not a crime to use inheritance in the test classes themselves. In fact, I urge you to do this if you have a good reason to. Implementing a base class can help alleviate standard problems in test code in the following ways:

- Reusing utility and factory methods
- Running the same set of tests over different classes (we’ll look at this one in more detail)
- Using common setup or teardown code (also useful for integration testing)
- Creating testing guidance for programmers who will derive from the base class

I’ll introduce you to three patterns based on test class inheritance, each one building on the previous pattern. I’ll also explain when you might want to use each pattern and what the pros and cons are for each.

These are the basic three patterns:

- Abstract test infrastructure class
- Template test class
- Abstract test driver class

We'll also take a look at the following refactoring techniques that you can apply when using the preceding patterns:

- Refactoring into a class hierarchy
- Using generics

ABSTRACT TEST INFRASTRUCTURE CLASS PATTERN

The *abstract test infrastructure class pattern* creates an abstract test class that contains essential common infrastructure for test classes deriving from it. Scenarios where you'd want to create such a base class can range from having common setup and teardown code to having special custom asserts that are used throughout multiple test classes.

We'll look at an example that will allow you to reuse a setup method in two test classes. Here's the scenario: all tests need to override the default logger implementation in the application so that logging is done in memory instead of in a file. (That is, all tests need to break the logger dependency in order to run correctly.)

Listing 7.3 shows these classes:

- *The LogAnalyzer class and method*—The class and method you'd like to test
- *The LoggingFacility class*—The class that holds the logger implementation you'd like to override in your tests
- *The ConfigurationManager class*—Another user of LoggingFacility, which you'll test later
- *The LogAnalyzerTests class and method*—The initial test class and method you'll write
- *The ConfigurationManagerTests class*—A class that holds tests for Configuration Manager

Listing 7.3 An example of not following the DRY principle in test classes

```
//This class uses the LoggingFacility Internally
public class LogAnalyzer
{
    public void Analyze(string fileName)
    {
        if (fileName.Length < 8)
        {
            LoggingFacility.Log("Filename too short:" + fileName);
        }
        //rest of the method here
    }
}

//another class that uses the LoggingFacility internally
public class ConfigurationManager
```

```

    {
        public bool IsConfigured(string configName)
        {
            LoggingFacility.Log("checking " + configName);
            return result;
        }
    }
}

public static class LoggingFacility
{
    public static void Log(string text)
    {
        logger.Log(text);
    }
    private static ILogger logger;
    public static ILogger Logger
    {
        get { return logger; }
        set { logger = value; }
    }
}

[TestFixture]
public class LogAnalyzerTests
{
    [Test]
    public void Analyze_EmptyFile_ThrowsException()
    {
        LogAnalyzer la = new LogAnalyzer();
        la.Analyze("myemptyfile.txt");
        //rest of test
    }

    [TearDown]
    public void teardown()
    {
        // need to reset a static resource between tests
        LoggingFacility.Logger = null;
    }
}

[TestFixture]
public class ConfigurationManagerTests
{
    [Test]
    public void Analyze_EmptyFile_ThrowsException()
    {
        ConfigurationManager cm = new ConfigurationManager();
        bool configured = cm.IsConfigured("something");
        //rest of test
    }

    [TearDown]
    public void teardown()
    {

```

```

        // need to reset a static resource between tests
        LoggingFacility.Logger = null;
    }
}

```

The `LoggingFacility` class is probably going to be used by many classes. It's designed so that the code using it is testable by allowing the implementation of the logger to be replaced using the property setter (which is static).

There are two classes that use the `LoggingFacility` class internally, the `LogAnalyzer` and `ConfigurationManager` classes, and you'd like to test both of them.

One possible way to refactor this code into a better state is to extract and reuse a new utility method to remove some repetition in both test classes. They both fake the default logger implementation. You could create a base test class that contains the utility method and then call the method from each test in the derived classes.

You won't use a common base `[SetUp]` method, because that would hurt readability of the derived classes. Instead you'll use a utility method called `FakeTheLogger()`. The full code for the test classes is shown here.

Listing 7.4 A refactored solution

```

[TestFixture]
public class BaseTestsClass
{
    public ILogger FakeTheLogger()
    {
        LoggingFacility.Logger =
            Substitute.For<ILogger>();
        return LoggingFacility.Logger;
    }

    [TearDown]
    public void teardown()
    {
        // need to reset a static resource between tests
        LoggingFacility.Logger = null;
    }
}

[TestFixture]
public class ConfigurationManagerTests:BaseTestsClass
{
    [Test]
    public void Analyze_EmptyFile_ThrowsException()
    {
        FakeTheLogger();
        ConfigurationManager cm =
new ConfigurationManager();
        bool configured = cm.IsConfigured("something");
        //rest of test
    }
}

```

Refactors into a common readable utility method to be used by derived classes

Automatic cleanup for derived classes

Call base class helper method

```

[TestFixture]
public class LogAnalyzerTests : BaseTestsClass
{
    [Test]
    public void Analyze_EmptyFile_ThrowsException()
    {
        FakeTheLogger();

        LogAnalyzer la = new LogAnalyzer();
        la.Analyze("myemptyfile.txt");
        //rest of test
    }
}

```

← Call base class
helper method

If you had used a `Setup` attributed method in the base class, it would have now automatically run before each test in either of the derived classes. The main problem this would introduce in the derived test classes is that anyone reading the code would no longer easily understand what happens when `setup` is called. They would have to look up the `setup` method in the base class to see what the derived classes get by default. This leads to less-readable tests, so instead you use a utility method that's more explicit.

This also hurts readability in a way, because developers who use your base class have little documentation or idea what API to use from your base class. That's why I recommend using this technique as little as you can but no less. More specifically, I've never had a good enough reason to use multiple base classes. I always made it more readable with a single base class, although a bit less maintainable. Also, do *not* have more than a single level of inheritance in your tests. That mess becomes unreadable faster than you can say, "Why is my build failing?"

Let's look at a more interesting use of inheritance to solve a common problem.

TEMPLATE TEST CLASS PATTERN

Let's say you want to make sure people who test specific kinds of classes in the code never forget to go through a certain set of unit tests for them as they develop the classes; for example, network code with packets, security code, database-related code, or just plain-old parsing code. The point is, you know that when they work on this kind of class in code, some tests must exist because that kind of class has to provide a known set of services with its API.

The template test class pattern is an abstract class that contains abstract test methods that derived classes must implement. The driving force behind this pattern is the need to be able to dictate to deriving classes which tests they should always implement.

If you have classes with interfaces in your system, they might be good candidates for this pattern. I find I use it when I have a hierarchy of classes that expands, and each new member of a derived class implements roughly the same ideas.

Think of an interface as a behavior contract, where the same end behavior is expected from all derived classes, but they can achieve the end result in different ways. An example of such a behavior contract could be a set of parsers all implementing `parse` methods that act the same way but on different input types.

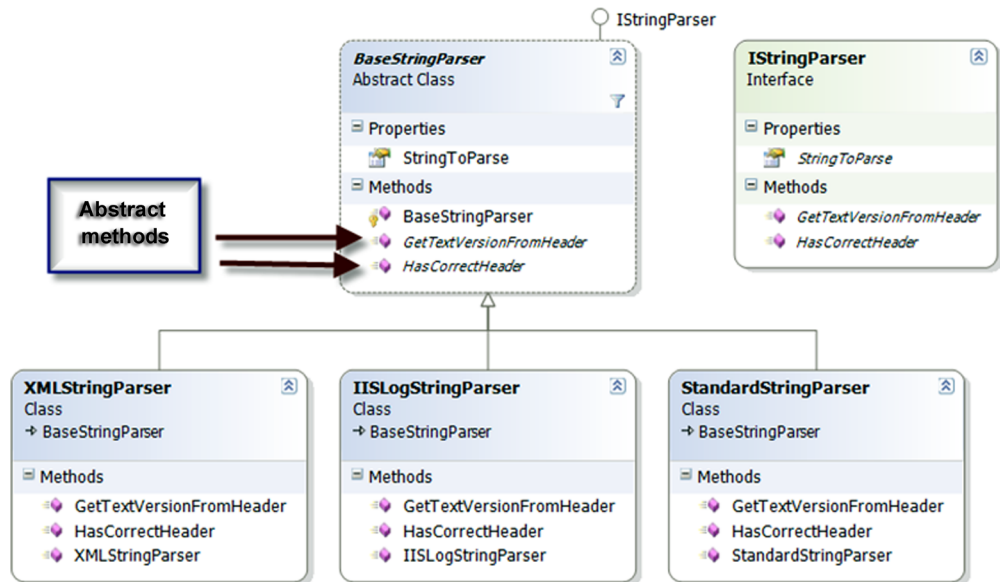


Figure 7.3 A typical inheritance hierarchy that you'd like to test includes an abstract class and classes that derive from it.

Developers often neglect or forget to write all the required tests for a specific case. Having a base class for each set of identically interfaced classes can help create a basic test contract that all developers must implement in derived test classes.

So here's a real scenario. Suppose you have the object model shown in figure 7.3 to test. The `BaseStringParser` is an abstract class that other classes derive from to implement some functionality over different string content types. From each string type (XML strings, IIS log strings, standard strings), you can get some sort of versioning info (metadata on the string that was put there earlier). You can get the version info from a custom header (the first few lines of the string) and check whether that header is valid for the purposes of your application. The `XMLStringParser`, `IISLogStringParser`, and `StandardStringParser` classes derive from this base class and implement the methods with logic for their specific string types.

The first step in testing such a hierarchy is to write a set of tests for one of the derived classes (assuming the abstract class has no logic to test in it). Then you'd have to write the same kinds of tests for the other classes that have the same functionality.

The next listing shows tests for the `StandardStringParser` that you might start out with before you refactor your test classes to use the template base test class pattern.

Listing 7.5 An outline of a test class for `StandardStringParser`

```
[TestFixture]
public class StandardStringParserTests
{
```

```

private StandardStringParser GetParser(string input)
{
    return new StandardStringParser(input);
}

[Test]
public void GetStringVersionFromHeader_SingleDigit_Found()
{
    string input = "header;version=1;\n";
    StandardStringParser parser = GetParser(input);

    string versionFromHeader = parser.GetStringVersionFromHeader();
    Assert.AreEqual("1", versionFromHeader);
}

[Test]
public void GetStringVersionFromHeader_WithMinorVersion_Found()
{
    string input = "header;version=1.1;\n";
    StandardStringParser parser = GetParser(input);

    //rest of the test
}

[Test]
public void GetStringVersionFromHeader_WithRevision_Found()
{
    string input = "header;version=1.1.1;\n";
    StandardStringParser parser = GetParser(input);
    //rest of the test
}
}

```

1 Defines the parser factory method

2 Uses factory method

Note how you use the `GetParser()` helper method **1** to refactor away **2** the creation of the parser object, which you use in all the tests. You use the helper method, and not a setup method, because the constructor takes the input string to parse, so each test needs to be able to create a version of the parser to test with its own specific inputs.

When you start writing tests for the other classes in the hierarchy, you'll want to repeat the same tests that are in this specific parser class. All the other parsers should have the same outward behavior: getting the header version and validating that the header is valid. How they do this differs, but the behavior semantics are the same. This means that for each class that derives from `BaseStringParser`, you'd write the same basic tests, and only the type of class under test would change.

First things first: let's see how you can easily dictate to derived test classes what tests are crucial to run. The following listing shows a simple example of this (you can find `IStringParser` in the book code on GitHub).

Listing 7.6 A template test class for testing string parsers

```

[TestFixture]
public abstract class TemplateStringParserTests
{
    public abstract
        void TestGetStringVersionFromHeader_SingleDigit_Found();
}

```

The test template class

```

public abstract
    void TestGetStringVersionFromHeader_WithMinorVersion_Found();

public abstract
    void TestGetStringVersionFromHeader_WithRevision_Found();
}

[TestFixture]
public class XmlStringParserTests : TemplateStringParserTests
{
    protected IStringParser GetParser(string input)
    {
        return new XMLStringParser(input);
    }

    [Test]
    public override
        void TestGetStringVersionFromHeader_SingleDigit_Found()
    {
        IStringParser parser = GetParser("<Header>1</Header>");

        string versionFromHeader = parser.GetStringVersionFromHeader();
        Assert.AreEqual("1",versionFromHeader);
    }

    [Test]
    public override
        void TestGetStringVersionFromHeader_WithMinorVersion_Found()
    {
        IStringParser parser = GetParser("<Header>1.1</Header>");

        string versionFromHeader = parser.GetStringVersionFromHeader();
        Assert.AreEqual("1.1",versionFromHeader);
    }

    [Test]
    public override
        void TestGetStringVersionFromHeader_WithRevision_Found()
    {
        IStringParser parser = GetParser("<Header>1.1.1</Header>");

        string versionFromHeader = parser.GetStringVersionFromHeader();
        Assert.AreEqual("1.1.1",versionFromHeader);
    }
}

```

← The derived class

Figure 7.4 shows the visualization of this code, if you have two derived classes. Note that `GetParser()` is just a standard method, and it can be named anything in the derived classes.

I've found this technique useful in many situations, not only as a developer but also as an architect. As an architect, I was able to supply a list of essential test classes for developers to derive from and to provide guidance on what kinds of tests they'd want to write next. It's essential in this situation that the test names are understandable. I use the word `Test` to prefix the abstract methods in the base class, so that people who override them in derived classes have an easier time finding what's important to override.

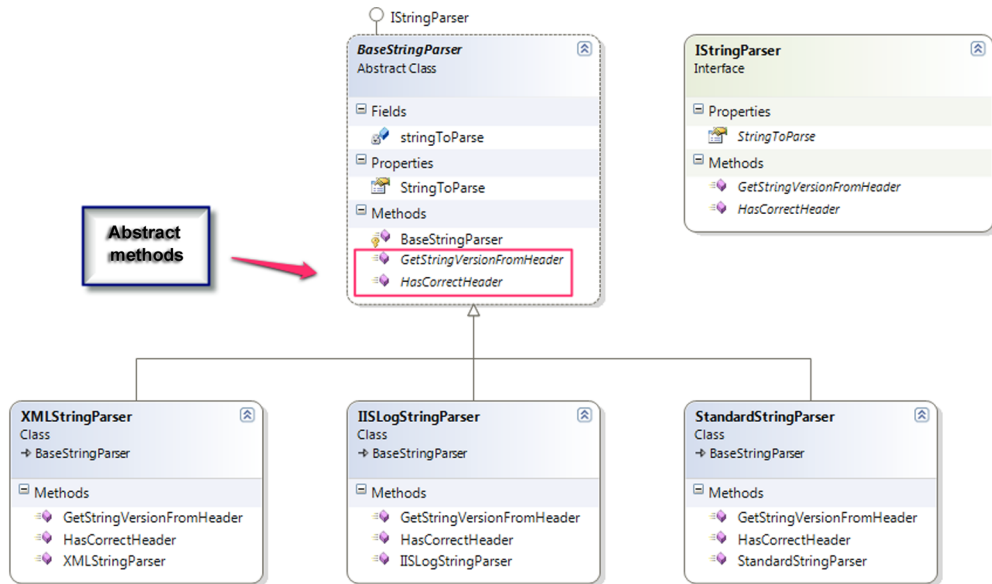


Figure 7.4 A template test pattern ensures that developers don't forget important tests. The base class contains abstract tests that derived classes must implement.

But what if you could make the base class do even more?

ABSTRACT "FILL IN THE BLANKS" TEST DRIVER CLASS PATTERN

The abstract test driver class pattern (I like to call it "fill in the blanks") takes the previous idea further, by implementing the tests in the base class itself and providing abstract method hooks that derived classes will have to implement.

It's essential that your tests don't explicitly test one class type but instead test against an interface or base class in your production code under test.

Here's an example of this base class.

Listing 7.7 A "fill in the blanks" base test class

```

public abstract class FillInTheBlanksStringParserTests
{
    protected abstract IStringParser GetParser(string input);
    protected abstract string HeaderVersion_SingleDigit { get; }
    protected abstract string HeaderVersion_WithMinorVersion { get; }
    protected abstract string HeaderVersion_WithRevision { get; }
    public const string EXPECTED_SINGLE_DIGIT = "1";
    public const string EXPECTED_WITH_REVISION = "1.1.1";
    public const string EXPECTED_WITH_MINORVERSION = "1.1";

    [Test]
    public void GetStringVersionFromHeader_SingleDigit_Found()
    {
        string input = HeaderVersion_SingleDigit;
        IStringParser parser = GetParser(input);
    }
}
  
```

Abstract input methods to provide data in a specific format for derived classes

Abstract factory method that requires a returned interface

Predefined expected output for derived classes if needed

```

    string versionFromHeader = parser.GetStringVersionFromHeader();
    Assert.AreEqual(EXPECTED_SINGLE_DIGIT, versionFromHeader);
}

[Test]
public void GetStringVersionFromHeader_WithMinorVersion_Found()
{
    string input = HeaderVersion_WithMinorVersion;
    IStringParser parser = GetParser(input);

    string versionFromHeader = parser.GetStringVersionFromHeader();
    Assert.AreEqual(EXPECTED_WITH_MINORVERSION, versionFromHeader);
}

[Test]
public void GetStringVersionFromHeader_WithRevision_Found()
{
    string input = HeaderVersion_WithRevision;
    IStringParser parser = GetParser(input);

    string versionFromHeader = parser.GetStringVersionFromHeader();
    Assert.AreEqual(EXPECTED_WITH_REVISION, versionFromHeader);
}
}

[TestFixture]
public class StandardStringParserTests : FillInTheBlanksStringParserTests
{
    protected override string HeaderVersion_SingleDigit
    {
        get {
            return string.Format("header\tversion={0}\t\n",
                EXPECTED_SINGLE_DIGIT);
        }
    }

    protected override string HeaderVersion_WithMinorVersion
    {
        get {
            return string.Format("header\tversion={0}\t\n",
                EXPECTED_WITH_MINORVERSION);
        }
    }

    protected override string HeaderVersion_WithRevision
    {
        get {
            return string.Format("header\tversion={0}\t\n",
                EXPECTED_WITH_REVISION);
        }
    }

    protected override IStringParser GetParser(string input)
    {
        return new StandardStringParser(input);
    }
}

```

Predefined test logic using derived inputs

Derived class that fills in the blanks

Filling in the right format for this requirement

Filling in the right type of class under test

In the listing, you don't have any tests in the derived class. They're all inherited. You could add extra tests in the derived class if that makes sense. Figure 7.5 shows the inheritance chain that you've just created.

How do you modify existing code to use this pattern? That's our next topic.

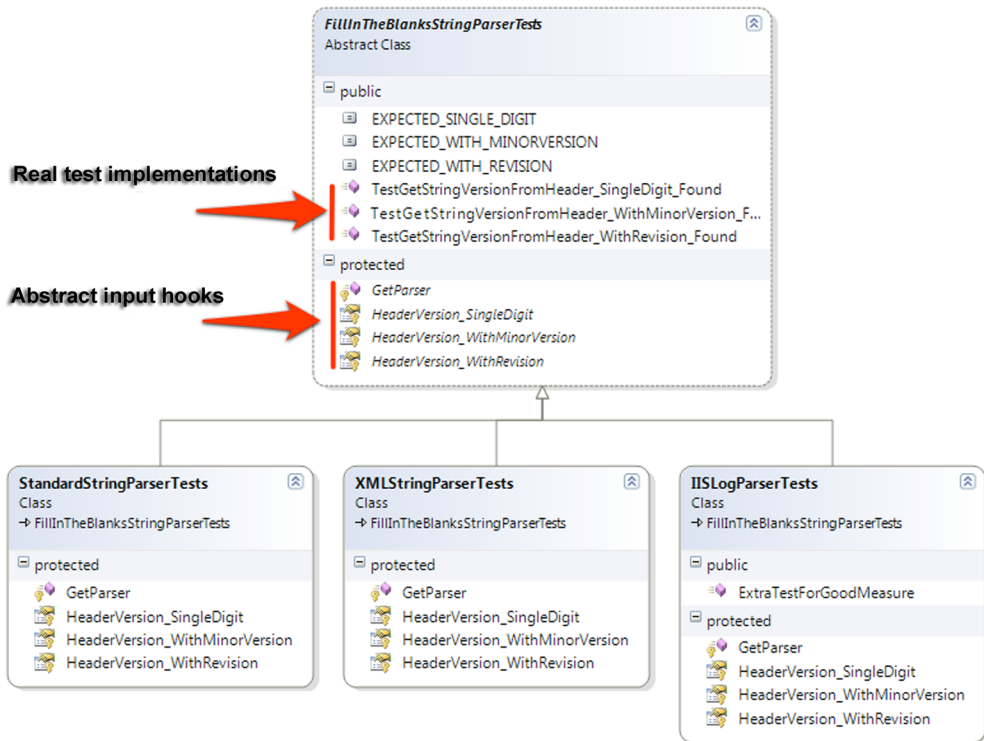


Figure 7.5 A standard test class hierarchy implementation. Most of the tests are in the base class, but derived classes can add their own specific tests.

REFACTORING YOUR TEST CLASS INTO A TEST CLASS HIERARCHY

Most developers don't start writing their tests with these inheritance patterns in mind. Instead, they write the tests normally, as shown in listing 7.7. The steps to convert your tests into a base class are fairly easy, particularly if you have IDE refactoring tools available, like the ones in Eclipse, IntelliJ IDEA, or Visual Studio (JetBrains' ReSharper, Telerik's JustCode, or Refactor! from DevExpress).

Here's a list of possible steps for refactoring your test class:

- 1 Refactor: extract the superclass.
 - Create a base class (BaseXXXTTests).
 - Move the factory methods (like GetParser) into the base class.
 - Move all the tests to the base class.
 - Extract the expected outputs into public fields in the base class.
 - Extract the test inputs into abstract methods or properties that the derived classes will create.
- 2 Refactor: make factory methods abstract, and return interfaces.
- 3 Refactor: find all the places in the test methods where explicit class types are used, and change them to use the interfaces of those types instead.

- 4 In the derived class, implement the abstract factory methods and return the explicit types.

You can also use .NET generics to create the inheritance patterns.

A VARIATION USING .NET GENERICS TO IMPLEMENT TEST HIERARCHY

You can use generics as part of the base test class. This way, you don't need to override any methods in derived classes; just declare the type you're testing against. The next listing shows both the generic version of the test base class and a class derived from it.

Listing 7.8 Implementing test case inheritance with .NET generics

```
//An example of the same idea using Generics
public abstract class GenericParserTests<T>
    where T:IStringParser
{
    protected abstract string GetInputHeaderSingleDigit();
    protected T GetParser(string input)
    {
        return (T) Activator.CreateInstance(typeof (T), input);
    }
    [Test]
    public void GetStringVersionFromHeader_SingleDigit_Found()
    {
        string input = GetInputHeaderSingleDigit();
        T parser = GetParser(input);

        bool result = parser.HasCorrectHeader();
        Assert.IsFalse(result);
    }

    //more tests
    //...
}
//An example of a test inheriting from a Generic Base Class
[TestFixture]
public class StandardParserGenericTests
    :GenericParserTests<StandardStringParser>
{
    protected override string GetInputHeaderSingleDigit()
    {
        return "Header;1";
    }
}
```

1 Defines generic constraint on parameter

2 Gets generic type variable instead of an interface

3 Returns generic type

4 Inherits from generic base class

5 Returns custom input for the current type under test

Several things change in the generic implementation of the hierarchy:

- The `GetParser` factory method ② no longer needs to be overridden. Create the object using `Activator.CreateInstance` (which allows creating objects without knowing their type) and send the input string arguments to the constructor as type `T` ③.
- The tests themselves don't use the `IStringParser` interface but instead use the `T` generic type ④.

- The generic class declaration contains the where clause that specifies that the T type of the class must implement the IStringParser interface ❶.
- The derived class returns a custom input into the base test ❺.

Overall, I don't find more benefit in using generic base classes. Any performance gain that would result is insignificant to these tests, but I leave it to you to see what makes sense for your projects. It's more a matter of preference than anything else.

Let's move on to something completely different: infrastructure API in your test projects.

7.6.2 Creating test utility classes and methods

As you write your tests, you'll create many simple utility methods that may or may not end up inside your test classes. These utility classes become a big part of your test API, and they may turn out to be a simple object model you could use as you develop your tests.

You might end up with the following types of utility methods:

- Factory methods for objects that are complex to create or that routinely get created by your tests.
- System initialization methods (such as methods for setting up the system state before testing, or changing logging facilities to use stub loggers).
- Object configuration methods (for example, methods that set the internal state of an object, such as setting a customer to be invalid for a transaction).
- Methods that set up or read from external resources such as databases, configuration files, and test input files (for example, a method that loads a text file with all the permutations you'd like to use when sending in inputs for a specific method and the expected results). This is more commonly used in integration or system testing.
- Special assert utility methods, which may assert something that's complex or that's repeatedly tested inside the system's state. (If something was written to the system log, the method might assert that X, Y, and Z are true, but not G.)

You may end up refactoring your utility methods into these types of utility classes:

- Special assert utility classes that contain all the custom assert methods
- Special factory classes that hold the factory methods
- Special configuration classes or database configuration classes that hold integration-style actions

There are a few helpful utility frameworks in the open source world of .NET that provide good examples of how to make something beautiful. One example is the *Fluent Assertions framework* that can be found at <https://github.com/dennisdoomen/FluentAssertions>.

Having those utility methods around doesn't guarantee anyone will use them. I've been to plenty of projects where developers kept reinventing the wheel, recreating utility methods they didn't know already existed.

Next, you'll find out how to make your API known.

7.6.3 Making your API known to developers

It's imperative that the people who write tests know about the various APIs that have been developed while writing the application and its tests. There are several ways to make sure your APIs are used:

- Have teams of two people write tests together (at least once in a while), where one is familiar with the existing APIs and can teach the other, as they write new tests, about the existing benefits and code that could be used.
- Have a short document (no more than a couple of pages) or a cheat sheet that details the types of APIs out there and where to find them. You can create short documents for specific parts of your testing framework (APIs specific to the data layer, for example) or a global one for the whole application. If it's not short, no one will maintain it. One possible way to make sure it's up to date is by automating the generation process:
 - Have a known set of prefixes or postfixes on the API helpers' names (helper [something], for example).
 - Have a special tool that parses out the API names and their locations and generates a document that lists them and where to find them, or have some simple directives that the special tool can parse from comments you put on them.
 - Automate the generation of this document as part of the automated build process.
- Discuss changes to the APIs during team meetings—one or two sentences outlining the main changes and where to look for the significant parts. That way the team knows that this is important and it's always a consideration.
- Go over this document with new employees during their orientation.
- Perform test reviews (in addition to code reviews) that make sure tests are up to standards of readability, maintainability, and correctness, and ensure that the right APIs are used when needed. For more on that practice, see <http://5whys.com/blog/step-4-start-doing-code-reviews-seriously.html> on my blog for software leaders.

Following one or more of these recommendations can help keep your team productive and will create a shared language the team can use when writing their tests.

7.7 Summary

Let's look back and see what you can draw from the chapter you've been through.

- Whatever testing you do—however you do it—automate it, use an automated build process to run it as many times as possible during the day or night, and continuously deliver the product as much as possible.
- Separate the integration tests from the unit tests (the slow tests from the fast ones) so that your team can have a safe green zone where all the tests must pass.

- Map out tests by project and by type (unit versus integration tests, slow versus fast tests), and separate them into different directories, folders, or namespaces (or all of these). I usually use all three types of separation.
- Use a test class hierarchy to apply the same set of tests to multiple related types under test in a hierarchy or to types that share a common interface or base class.
- Use helper classes and utility classes instead of hierarchies if the test class hierarchy makes tests less readable, especially if there's a shared setup method in the base class. Different people have different opinions on when to use which, but readability is usually the key reason for not using hierarchies.
- Make your API known to your team. If you don't, you'll lose time and money as team members unknowingly reinvent APIs over and over again.