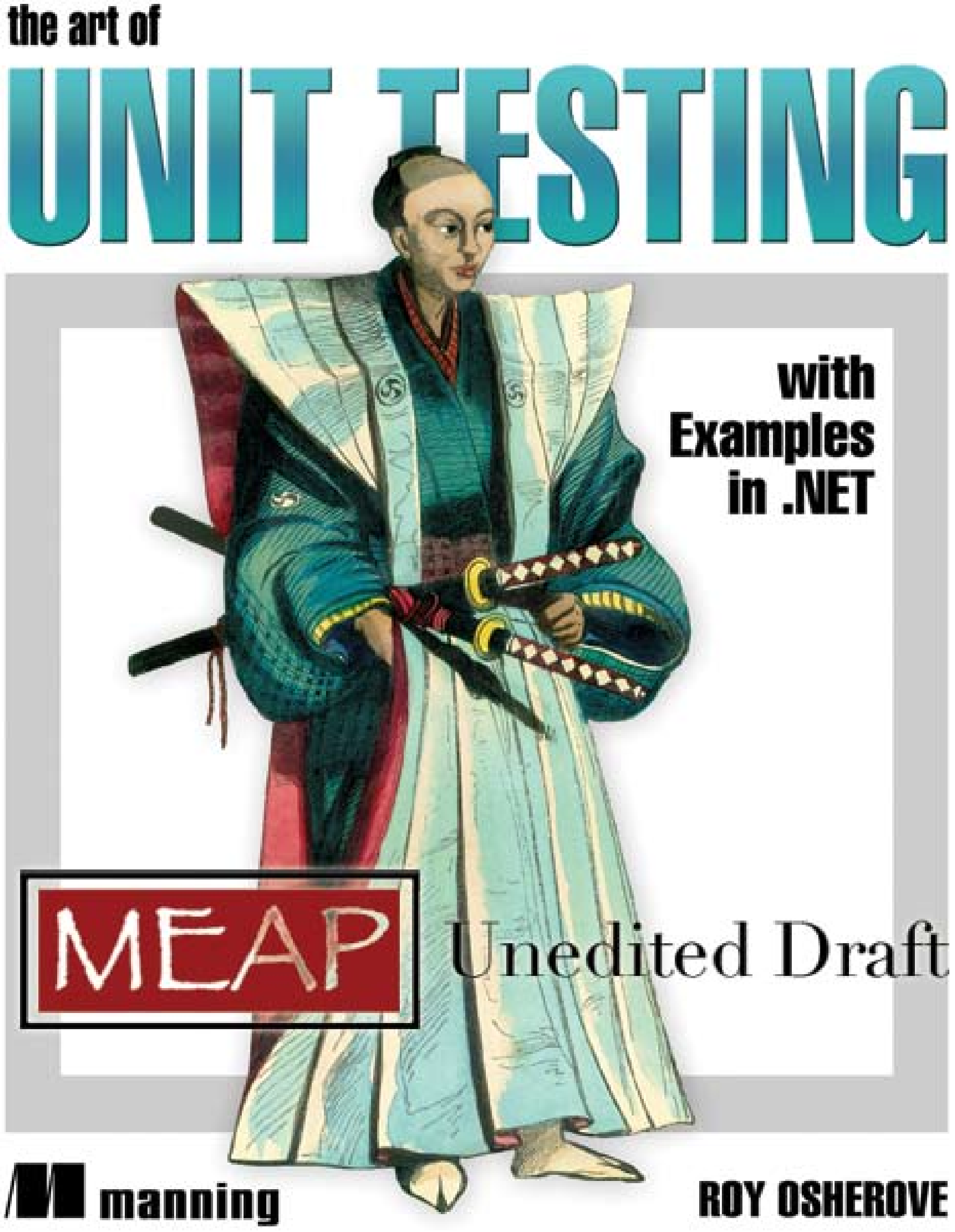


the art of

UNIT TESTING

with Examples in .NET

MEAP Unedited Draft

manning

ROY OSHEROVE

MEAP Edition
Manning Early Access Program

Table of Contents

Part 1 Getting started

Chapter 1 The basics of unit testing

Chapter 2 The first unit test

Part 2 Core techniques

Chapter 3 Using Stubs to break dependencies

Chapter 4 Interaction testing using Mock Objects

Chapter 5 Mock Object frameworks

Part 3 The test code

Chapter 6 Test hierarchies and organization

Chapter 7 The pillars of good tests

Part 4 Design and process

Chapter 8 Integrating unit testing into the organization

Chapter 9 Working with legacy code

Appendices

Appendix A Design and testability

Appendix B Extra tools and frameworks

1 The basics of unit testing

One of the biggest failed projects I worked on had unit tests. Or so I thought. I was leading a group of programmers to create a billing application, and we were doing it in a fully test-driven manner – writing the test, then writing the code, seeing the test fail, making the test pass, refactor, rinse, repeat.

The first few months of the project were great; things were looking up, and we had tests that proved that our code worked. As time went by, requirements changed, and we were forced to change our code to fit those new requirements. Whenever we changed the code, tests broke and we had to fix them – the code was still working, but the tests we wrote were so brittle that any little change in our code broke them, even though the code was working just fine. It became a daunting task to change our code in a class or a method for fear of changing all the unit tests involved with that unit being tested.

Worse yet, some tests became unusable because the people who wrote them had left the project and no one knew how to maintain the tests, or what they were testing. The names we gave our unit test methods were not clear enough. We had tests relying on other tests. We ended up throwing away most of the tests less than 6 months into the project.

It was a miserable failure because we let the tests we wrote do more harm than good – they were taking too much time to maintain and understand than they were saving us in the long run. So we stopped using them. I moved on to other projects, where we did a better job writing our unit tests, and even had some great successes using them, saving huge amounts of debugging and integration time.

Ever since that first project that failed, I've been compiling best practices for unit tests and using them on the next project. Every time I get involved in a new project, I find a few more best practices. A solid naming guideline is just one of those. Understanding how to write unit tests, and making them maintainable, readable and trust-worthy is what this book is about– no matter what language or Integrated Development Environment (IDE) you work with.

This book will cover the basics of writing a unit test, then move on to the basics of Interaction testing, and from then we'll move to best practices for writing, managing and maintaining unit tests in the real world.

1.1 Unit testing - classic definition

Unit testing in software development is not a new concept. It's been floating around since the early days of the programming language Smalltalk in the 1970s and proves itself time and time again as one of the best ways a developer can improve the quality of code while gaining a deeper understanding of the functional requirements of a class or method.

Kent Beck is the person who introduced the concept of unit testing in Smalltalk. The concept he created has carried on into many programming languages today, which has made unit testing an extremely useful practice in software programming. Before we get too far, we need to define unit testing better. I will start with the classic definition most of us have heard a million times.

UNIT TEST – THE CLASSIC DEFINITION

A unit test is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed. A "unit" is a method or function.

The piece of code being tested is often called *SUT* (System Under Test). The piece of code that tests the SUT will usually reside in a *Test Method*. This classic definition, while technically correct, is hardly enough to get us started down a path where we can better ourselves as developers. Chances are you already know this and are getting bored even reading this definition again, as it appears practically in any web site or book that discusses unit testing.

Don't worry; in this book, I'll take you beyond the classic definition of unit testing by addressing issues not mentioned in it: Maintainability, Readability, Correctness and more. However, precisely *because* you probably might already be familiar with the classic definition, it gives us a shared knowledge base from which to extend the idea of a unit test into something with more value than is currently defined.

No matter what programming language you are using, one of the hardest aspects of defining a unit test is defining what is meant by a "good" one.

1.1.1 Defining a "good" unit test

I firmly believe that there's no point in writing a bad unit test. If you're going to write a unit test badly, you may as well not write it all and save yourself the trouble it will cause down the road with maintainability and time schedules. Defining what a good unit test is, is the first step we can do to make sure we don't start off with the wrong notion of what we're trying to write.

Most people who try to unit test their code either give up at some point or don't actually perform unit tests. Instead, they either rely on system and integration tests to be performed much later in the lifecycle of the product they are developing or resort to manually testing the code via custom test applications or actually using the end product they are developing to invoke their code.

To succeed, it is essential that you not only have a *technical* definition of a unit test, but that you describe *the properties of a good unit test*. To understand what a good unit test is, we'll need look at what we do today to understand what developers have been doing so far in a software project.

If you haven't been doing unit tests, how *did* you make sure that the code works?

1.1.2 We've all written unit tests

You may be surprised to learn this, but you've already implemented some types of unit testing on your own. Have you ever met a developer who has not tested the code they wrote before letting it out from under their hands? Well, neither have I.

It might have been a console application that called the various methods of a class or component, some specially created Winform or Webform UI that checks the functionality of that class or component, or even manual tests that were run by performing various actions within the real application's UI to test the end functionality. The end result is that the developer is certain, to a degree, that the code works well enough to give it away to someone else. Figure 1.1 shows how most developers test their code.

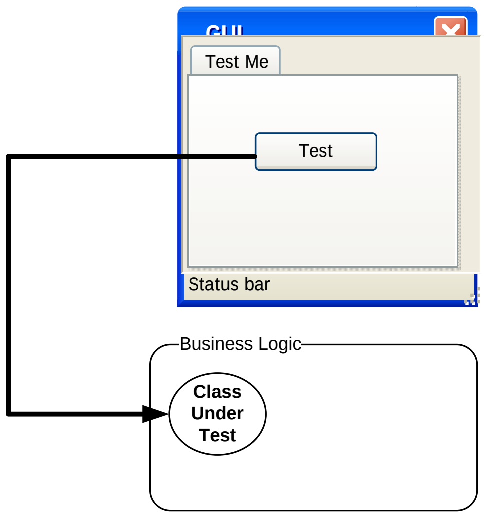


Figure 1.1 For classic testing, developers use a GUI (windows or web) and trigger an action on the class they want to test. Then they check the results somewhere (UI or other places).

Of course, the UI may change, but the pattern is usually the same: use a manual external tool to check something repeatedly.

While these may have been very useful, and, technically, they may be close to the “classic” definition of a unit test (although sometimes not even that, especially when they are done manually), they are far from the definition of a *unit test* as used in this book. That brings us to the first and most important question a developer has to face when attempting to define the qualities of a good unit test: what a unit test is and is not.

1.2 Good unit tests

We can map out what properties a unit test *should* have. After we write these out, we’ll try to define what a unit test is:

- It is automated and repeatable.
- It is easy to implement.
- Once it’s written, it stays on for the future.
- Anyone can run it.
- It runs at the push of a button.
- It runs quickly.

Many people confuse the act of simply testing their software with the concept of a unit test.

To start off, ask yourself the following questions about the tests you’ve written up to now:

- Can I run and get results of a unit test I wrote two weeks/months/years ago?
- Can any member of my team run and get the results from unit tests I wrote two months ago?
- Can it take me no more than a few minutes to run all the unit tests I’ve written so far?
- Can I run all the unit tests I’ve written at the push of a button?
- Can I write a basic unit test in no more than a few minutes?

If you’ve answered any of these questions with a “no,” there’s a high probability that what you’re actually implementing is not really a unit test. It’s *some* kind of test, absolutely, and it’s *just* as important as a unit test, but it has enough drawbacks to consider writing tests that answer “yes” to all of these questions.

“So what *was* I doing until now?” you might ask. You’ve done *integration testing.* We’ll touch on what that means in the next section.

1.3 Integration tests

What happens when your car breaks down? How do you know what the problem is, let alone how to fix it? Perhaps all you've heard is a weird rumbling sound before the car suddenly stopped moving, or some odd-colored exhaust fumes were being emitted – but where *is* the problem? An engine is made of many parts working together in partnership—each relying on the other to function properly and produce the final result: a moving car. If the car stops moving, the fault could be on any one of these parts, or more than one of them all at once. In effect, it is the integration of those parts that makes the car move – you could think of the car's eventual movement as the ultimate test of the integration of these parts.

If the test fails – all the parts fail together, and if it succeeds – they all succeed as well.

The same thing happens in software: The way most developers test their functionality is in the eventual final functionality through some sort of user interface. Clicking some button somewhere triggers a series of events – various classes and components working together, relying on each other to produce the final result – a working set of functionality. If suddenly the test fails – all of these software components fail as a team – it could get really hard finding out the true culprit of the failure of the final operation. See figure 1.2 for an example.

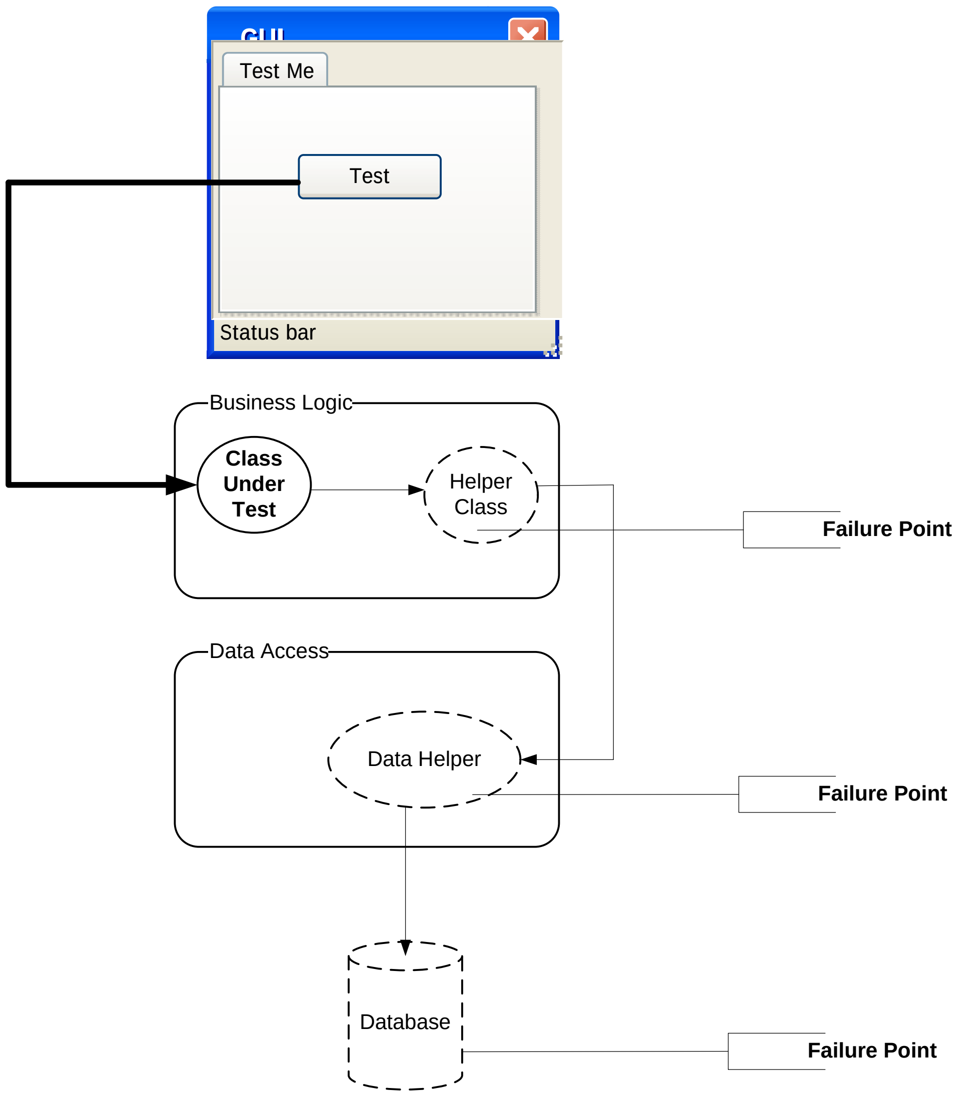


Figure 1.2 – You can have many failure points in an integration test because all the units have to work together, and each of them could malfunction, making it harder to find the source of the bug. With unit tests, like the hero says in the movie *Highlander*, "There

can be only one" (culprit).

With previous description of an integration test in mind, let's look at the classic definition of Integration Tests. According to the book *The Complete Guide to Software Testing* integration testing is "An orderly progression of testing in which software and/or hardware elements *are combined and tested* until the entire system has been integrated."

That definition of integration testing falls a bit short of what many people do all the time, not as part of a system integration test, but as part of development and unit tests (in parallel).

BETTER DEFINITION: INTEGRATION TESTING

Testing two or more dependent software modules as a group.

An integration test would exercise many units of code that work together to evaluate one or more results, while a unit test would usually exercise and test only a single unit in isolation.

The *questions* at the beginning of Section 1.2 can help you realize some of the drawbacks with integration testing and are easily identifiable. We'll cover them next and try to define the good qualities we are looking for in a unit test based on the explanations to these questions.

1.3.1 Drawbacks of integration tests

Let's apply the questions in the previous section to integrsation style tests. This time I'll list some things that we will want to achieve when implementing real world unit tests and why these questions are important for finding out whether they are achieved or not.

Can You Run the Same Test in the Future?

Question	Can I run and get results of a unit test I wrote two weeks/months/years ago?
Problem	If you can't do that, how would you know whether you broke a feature that you created two weeks ago (also called a "Regression")? Code changes all the time during the life of an application. When you can't (or won't) run the tests for all the previous working features after changing your code, you just might break it without knowing. I call it "accidental bugging." This "accidental bugging" seems to occur a lot near the end of a software project, when under time pressures, developers are fixing bugs and introducing new bugs inadvertently as they solve the old ones. Some places fondly call that stage in the project's lifetime "hell month." Wouldn't it be great to know that you broke something within three minutes of breaking it? We'll see how that can be done later in this book
Test Rule	Tests should be easily executed in their original form, not manually.

Can Other Team Members Run the Same Test?

Question	Can any member of my team run and get the results from unit tests I wrote two months ago?
Problem	This goes with the last point, but takes it up a notch. You want to make sure that you don't break someone else's code when you fix/change something. Many developers fear changing *legacy code* in older systems for fear of not knowing what dependencies other code has on what they are changing. In essence, they are changing the system into an unknown state of stability. Few things are scarier than not knowing if the application still works, especially when you didn't write that code. But if you had the ability to make sure nothing broke, you'd be much less afraid of taking on code with which you are less familiar just because you had that safety net of unit tests to tell you whether you broke something anywhere in the system.
Test Rule	Anyone can get and run the tests

We just introduced a new term in the last question, let's establish what *Legacy Code* means.

LEGACY CODE

Legacy code is defined by *Wikipedia* as “source code that relates to a no-longer supported or manufactured operating system or other computer system,” but many shops refer to any older version of the application currently under maintenance as “legacy code.” It often refers to code that is hard to work with, hard to test, and usually even hard to read.

A client of mine once defined legacy code in a very down-to-earth way, “Code that works.” In many ways, although that does bring a smile to your face, you can’t help but nod and say to yourself “yes, I know what he’s talking about. . ..”. Many people like to define Legacy code as “code that has not tests”, which is also a reasonable enough definition to be considered while reading this book.

Can You Run All Unit Tests in Minutes?

Question	Does it take no more than a few minutes to run all the unit tests I’ve written so far?
Problem	If you can’t run your tests quickly (seconds is better than minutes), you’ll run them less often (daily, or even weekly or monthly in some places). The problem is that when you change code, you want to get feedback as early as possible to see if you broke something. The longer you take between running the tests, the more changes you make to the system, and when you do find that you broke something, you’ll have many more places to look for to find out where the bug resides.
Test Rule	Tests should run *quickly.*

Can You Run All Unit Tests at the Push of a Button?

Question	Can I run all the unit tests I’ve written at the push of a button?
Problem	If you can’t, that probably means that you have to configure the machine on which the tests will run so that they run correctly (setting connection strings to the database, as an example), or that your unit tests are not *fully automated.* If you can’t fully automate your unit tests, bigger chance that you’ll avoid running them repeatedly, or anyone else on your team for that matter No one likes to get bogged down with little configuration details to run tests when all they are trying to do is make sure that the system still works. As developers, we have more important things to do, like write more features into the system.
Test Rule	Tests should be easily executed in their original form, not manually.

Can You Write a Basic Test in a Few Minutes?

Question	Can I write a basic unit test in no more than a few minutes?
Problem	One of the easiest ways to spot an integration test is that it takes time to prepare correctly and to implement, not only to execute. It takes time to figure out how to write it because of all the internal and sometimes external dependencies (a database may be considered an external dependency). If you're not automating the test, that is less of a problem, but that just means you're losing all the benefits of an automated test. The harder it is to write a test, the less likely you are to write more tests, or focus on anything else than just the "big" stuff that you're worried about. One of the strengths of unit tests is that they tend to test every little thing that might break, not just the big stuff – people are often surprised at just how many bugs they can find in code they considered to be totally simple and bug free. When you concentrate only on the big tests, the *coverage* that your tests have is smaller – many parts of the core logic in the code are not tested, and you may find many bugs that you hadn't considered.
Test Rule	Unit Tests against the system should be easy and quick to write

From the lessons we've learned so far about what a unit test is not and about the various features that need to be present for testing to be useful, we can now start to answer the primary question this chapter poses: what is a unit test?

1.4 Unit test - final definition

Now that we've covered the important properties that a unit test should have, let's define a unit test once and for all.

DEFINITION: A GOOD UNIT TEST

A unit test is an automated piece of code that invokes a different method and then checks some assumptions about the *logical* behavior of that method or class under test.

A unit test is written using a *unit testing framework*. It can be written easily and runs quickly. It can be executed, repeatedly, by anyone on the development team.

That definition sounds more like something I'd like to see you implement after writing this book. But it sure looks like a tall order, especially based on how you've probably implemented unit tests so far. It makes us take a hard look at the way we, as developers, have implemented testing up until now, compared to how we'd like to implement it.

Here's what "Logical Code" means in the definition of a unit test:

DEFINITION: LOGICAL CODE

Logical code means any piece of code that has some sort of logic in it, small as it may be. It's logical code if the piece of code has one or more of the following:

- An IF statement
- A loop
- Swith, case
- Calculations
- Any other type of decision making code

Properties (getters/setters in java) are a good example of code that usually does *not* contain any logic, and so does not require testing. But watch out, once you want to add any check inside the property, you'll want to make sure that logic is being tested for correctness.

In the next section we'll take a look at a simple example of a unit test done entirely in code, without using any Unit Test Framework (which you'll learn about in chapter 2)

1.5 A simple unit test example

Assume we have a `SimpleParser` class in our project that we'd like to test as shown in listing 1.1. It takes in a string of 0 or more numbers with a comma between them. If there are no numbers it returns zero. For a single number it returns that number as an int. For multiple numbers it sums them all up and returns the sum (right now it can only handle zero or one number though):

Listing 1.1: A simple parser class we'd like to test

```
public class SimpleParser
    {
        public int ParseAndSum(string numbers)
        {
            if(numbers.Length==0)
            {
                return 0;
            }
            if(!numbers.Contains(","))
            {
                return int.Parse(numbers);
            }
            else
            {
                throw new InvalidOperationException("I can only handle 0 or 1 numbers for 
now!");
            }
        }
    }
```

We can add a simple console application project that has a reference to the assembly containing this class, and write a method like this in a class called `SimpleParserTests`, as shown in listing 1.2.

The test is simply a method, which invokes the production class (production: the actual product you're building and would like to test) and then checks the returned value. If it's not what is expected to be, it writes to the console. It also catches any exception and writes it to the console.

Listing 1.2:A simple coded method that tests our SimpleParser class.

```
class SimpleParserTests
    {
        public static void TestReturnsZeroWhenEmptyString()
        {
            try
            {
                SimpleParser p = new SimpleParser();
                int result = p.ParseAndSum(string.Empty);
                if(result!=0)
                {
                    Console.WriteLine(@"***
SimpleParserTests.TestReturnsZeroWhenEmptyString:
-------
Parse and sum should have returned 0 on an empty string");
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e);
            }
        }
    }
```

Next, we can simply invoke the tests we've written using a simple `Main` method run inside a console application in this project, as seen in listing 1.3. The main method is used here as a simple test runner, which invokes the tests one by one, letting them write out to the console for any problem. Since it's an executable, this can be run without human intervention (assuming no test pops up any interactive user dialogs).

Listing 1.3: Running our coded tests via a simple console application

```
public static void Main(string[] args)
        {
            try
            {
                SimpleParserTests.TestReturnsZeroWhenEmptyString();
            }
            catch (Exception e)
            {
                Console.WriteLine(e);
            }
        }
```

The test also catches any exception that might occur and writes it to the console output.

It's the test method's responsibility to catch any exceptions that occur and write them to the console, so that they don't interfere with the running of other methods after them. We can then add more method calls into the main method as we add more and more tests into the test project. Each test is responsible writing the problem output (if there is a problem) to the console screen).

Obviously, this is a very simplistic way of doing it. You might want to have a generic "`ShowProblem`" method that all unit tests can use, which will format the errors the same way for everyone, and you could add special helper methods that help ease the checks on various things like null objects, empty strings, and so on, so that you don't write the same long lines of code in many tests.

Listing 1.4 shows what this test would look like with a little more generic `ShowProblem` method:

Listing 1.4: Using a more generic implementation of the ShowProblem method

```
public class TestUtil
    {
        public static void ShowProblem(string test,string message )
        {
            string msg = string.Format(@"
---{0}---
       {1}
--------------------
", test, message);
            Console.WriteLine(msg);
        }
    }


public static void TestReturnsZeroWhenEmptyString()
        {
            //use .NET's reflection API to get the current method's name
// it's possible to just hard code this, but it's a useful technique to know
            string testName = MethodBase.GetCurrentMethod().Name;
            try
            {
                SimpleParser p = new SimpleParser();
                int result = p.ParseAndSum(string.Empty);
                if(result!=0)
                {
                   //Calling the helper method
                    TestUtil.ShowProblem(testName, "Parse and sum should have returned 0 on
an empty string");
                }
            }
            catch (Exception e)
```

```
                {
                    TestUtil.ShowProblem(testName, e.ToString());
                }
            }
```

Making helper methods more generic to show error reports, so tests are written more easily are just one of the things that unit test frameworks, which we'll talk about in chapter 2, help out with.
Before we get there, I'd like to discuss one important matter, regarding not just *how* you write a unit test, but *when* you would want to write it during the development process – that's where Test Driven Development comes into play.

1.6 Test-driven development

Even if we know how to write structured, maintainable, and solid tests with a unit test framework, we're left with the question of when we should write the tests.

Many people feel that the best time to write unit tests for software (if any at all) is after the software has been written. Still, a growing number of people prefer a very different approach to writing software – writing a unit test *before* the actual production code is even written. That approach is called test driven development or TDD for short.

Figures 1.3 and 1.4 show the differences between traditional coding and Test Driven Development.

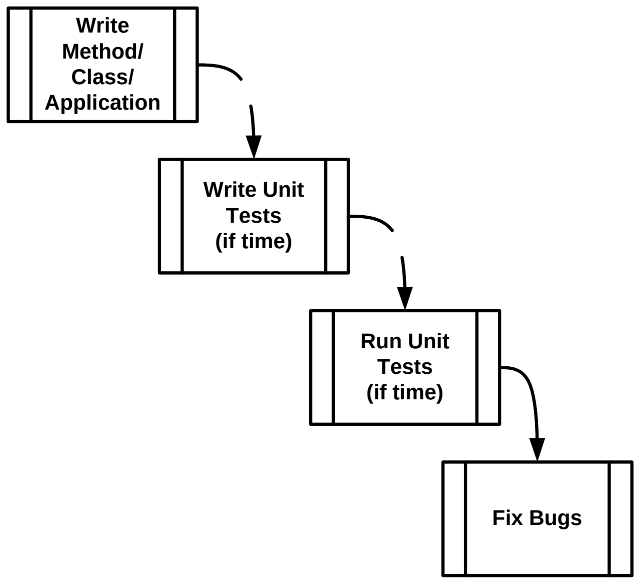


Figure 1.3 The "Traditional" way of writing unit tests. Dotted lines represented actions people treat as optional during the process.

Test Driven Development is vastly different than "classic" development as figure 1.5 shows. You actually begin by writing a test that fails, then move on to creating the actual production code, and see the test pass, and continue on to either refactor your code or to create another failing test.

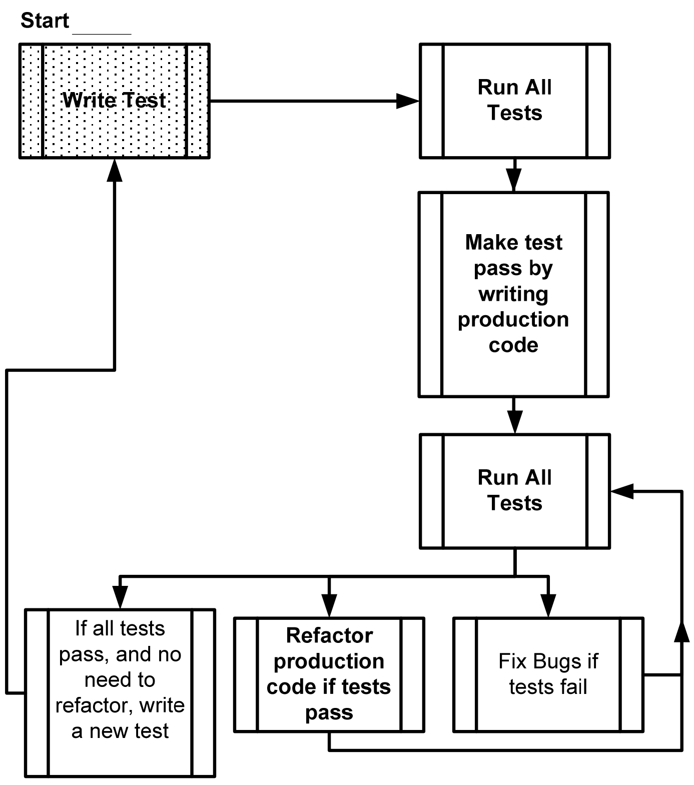

Start
Write Test
Run All Tests
Make test pass by writing production code
Run All Tests
If all tests pass, and no need to refactor, write a new test
Refactor production code if tests pass
Fix Bugs if tests fail

Figure 1.4 Test Driven Development - a bird's eye view. Notice the spiral nature of the process: write test, write code, refactor – write next test. It shows clearly the incremental nature of TDD: small steps lead to a quality end result.

You can read a more in-depth look at TDD in chapter 12 – "Test Driven Development", which is dedicated to this technique. That chapter also references other books you might want to read on this subject.. This book focuses on the technique of writing a good unit test, rather than test driven development, but the subject is so important it cannot go unwritten in any book that talks about unit testing.

I'm a big fan of doing test-driven development. I've written several major applications and frameworks using this technique, have managed teams that utilize this technique, and have taught more than a hundred courses and workshops on test-driven development and unit testing techniques. Throughout my career I've found TDD to be helpful in creating quality code, quality tests and a better design for the code I was writing. I am now convinced more than ever that it can work for your benefit, but not without a price. It's worth the admission price, though. Big time.

It is important to realize TDD does not ensure project success or tests that are robust or maintainable. It is quite easy to get caught up in the technique of TDD and not pay attention to the way the unit test is written: its naming, how maintainable or readable it is, or whether it really does test the right thing or might have a bug. That's why I'm writing this book. But let's turn our focus for a moment to *test driven development.*

The technique itself is quite simple.

1. Write a failing test to prove code or functionality is missing from the end product

The test is written *as if* the production code is already working, so the test failing means there is a bug in the production code. So, if I wanted to add a new feature to a calculator class that remembers the "LastSum" value, I would write a test that verifies that LastSum is indeed some number as if it was working

	already. It will fail because we have not implemented that functionality yet.
2. Make the test pass	By writing production code that fits the reality that your test is expecting. It should be written as simply as possible.
3. Refactor your code	When the test passes, you are free to move on to the next unit test or *refactor* your code to make it more readable, remove code duplication, and more.

Refactoring can be done after writing several tests, or after each test. In any case, it is a very important practice, because it makes sure your code gets easier to read and maintain, while still passing all of the previously written tests.

REFACTORING DEFINITION

Refactoring is the act of changing a piece of code without changing its functionality. If you've ever renamed a method, you've done refactoring. If you've ever split a large method into multiple smaller method calls, you've refactored your code. It still does the same thing; it's just easier to maintain, read, debug, and change.

These outlined steps sound very technical, but there is a lot of wisdom behind them. In chapter 21, I'll be looking closely at those reasons, and will be explaining all the benefits. For now, I can tell you the following without spoiling it for that chapter: done correctly, TDD can make your code quality soar, decrease the amount of bugs, raise your confidence in the code, shorten the time to find bugs, improve your code's design, and keep your manager happier. If TDD is done incorrectly, it can make your project schedule slip, waste your time, lower your motivation, and decrease/worsen? Your code quality. It's a double-edged sword, which many people only find out the hard way. In the rest of this book, we'll see how to make sure only the first part of this paragraph is true for your projects.

1.7 Summary

In this chapter we discussed the origins of unit tests. We then defined a good unit test as an automated piece of code that invokes a different method and then checks some assumptions on the logical behavior of that method or class, is written using a *unit testing framework*, can be written easily, runs quickly, and can be executed repeatedly by anyone on the development team.

To understand what a unit is, we had to figure out the sort of testing we've done up until now. We identified that type of testing as integration testing and defined it as testing a set of units that depend on each other as one unit.

I cannot stress enough the importance of realizing the difference between unit tests and integration tests. You will be using that knowledge in your day to day life as developers when deciding when to place your tests, what kind of test to write when, and which option is better for specific problems you will be dealing with. It will also help in identifying how to fix problems with tests that are already causing you some headaches. I will be talking later in the book even more about integration testing and best practices for those kinds of test.

We also talked about the cons of doing just integration testing without a framework behind it: hard to write, slow to run, needs configuration, hard to automate, and more. While you do want to have integration tests in a project, Unit Tests can provide a lot of value beforehand, when bugs are smaller, easier to find, and less code is there to skim through.

Lastly, we talked about test driven development, how it's different than traditional coding, and the basic benefits of TDD. I've found TDD helps out in making sure that your code coverage (how much of the code your tests actually exercise) in your test code is very high (close to 100% of *logical* code) and it helps make sure that your test can be trusted with a little more confidence by making sure that it indeed fails when the production code is not there, and passes when the production code actually works.

By writing tests *after* writing the code, you assume the test is OK just because it's passing. Trust me – finding bugs in your *tests* is one of the most frustrating things you can imagine. It's important you don't let your tests get to that state, and TDD is one of the best ways I know to make sure that possibility is close to zero.

In the next chapter we'll dive in and start writing our first unit tests using NUnit – the *de facto* unit test framework for .NET developers.

2 The first unit test

When I first started writing unit tests with a real unit test framework, there was little documentation out there for me to learn how to use them. The frameworks I worked with (I was mostly coding in VB 5 and 6 at the time) did not have proper examples. It was a tough challenge to learn to work with them and I started out doing a rather poor job of writing tests.
This chapter should be useful to you if you were in the same situation: You want to start writing tests, but you have no idea where to start.
While in the previous chapter I discussed the main ideas in unit testing, this chapter should get you well on your way to write real world unit tests with a framework called NUnit- a .NET Unit testing framework. It is my favorite framework in .NET for unit testing because it's easy to use and easy to remember, and has lots of great features.

There are other frameworks out there in .NET, some with even more features, but NUnit is where I always start, and perhaps expand to a different framework when the need arises. We'll see exactly how NUnit works, its syntax, and how to run it and get the 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 demonstrate the techniques and best practices outlined in each chapter.

First, let's understand what a unit test framework is, and what it enables us to do that we couldn't and wouldn't do before using it.

2.1 Frameworks for unit testing

If you think of what advantages a software Integrated Development Environment (IDE) gives you as a developer, it would be easy to see how those advantages apply to Unit Test Frameworks.
When using a modern day IDE like Visual Studio .NET or Eclipse for Java, you do all your coding tasks within that environment, in a structured manner: You write the code, you compile it, you build any resources like graphics and text into it,(with C++ you may link the binary output to the referenced libraries in your code) and you create the final binary – all that in no more than a couple of keystrokes on the keyboard.

It wasn't always that easy to build your code.
To this day, in many IDEs that deal with other environments(Unix comes to mind), the steps to getting a final binary output from your code are not as simple, and may require manually calling other external tools to do parts of this big task. Even in *those* cases, some sort of command or batch file could be used to invoke all those different compilers and linkers in the correct order to build the final output. doing things completely manually, even in those days would be error-prone, time consuming, and people would defer doing it as much as possible.

Today's developers are miles ahead in terms of automation. IDEs give a structured environment for performing the task, writing the code, and producing the output automatically. In the same way, Unit testing frameworks help developers write tests faster with a set of known APIs, execute those tests automatically, and review the results of those tests easily.

Let’s find out what that means exactly.The tests you’ve done up until now were:

- **Not structured** —You had to reinvent the wheel every time you wanted to test the feature. One test looks like a console application; the other is a UI form; and another is a web form. You don’t have that time to spend, and it fails the “easy to implement” requirement.
- **Not repeatable**—Neither you nor your team can run tests you’ve written in the past. That breaks the “repeatedly” requirement and prevents you from finding regression bugs.
- **Not on *all* your code**—All the code that matters, anyway. That means all the code that has pieces of logic in it, since each and every one of those could contain a potential bug. (Property getters and setters don’t count as logic, unless you actually have some sort of logic inside them.)

In short, what you’re missing is a *framework* for writing, running, and reviewing unit tests and their results. Figure 2.1 shows the areas in software development where the unit test framework of your choice has influence.

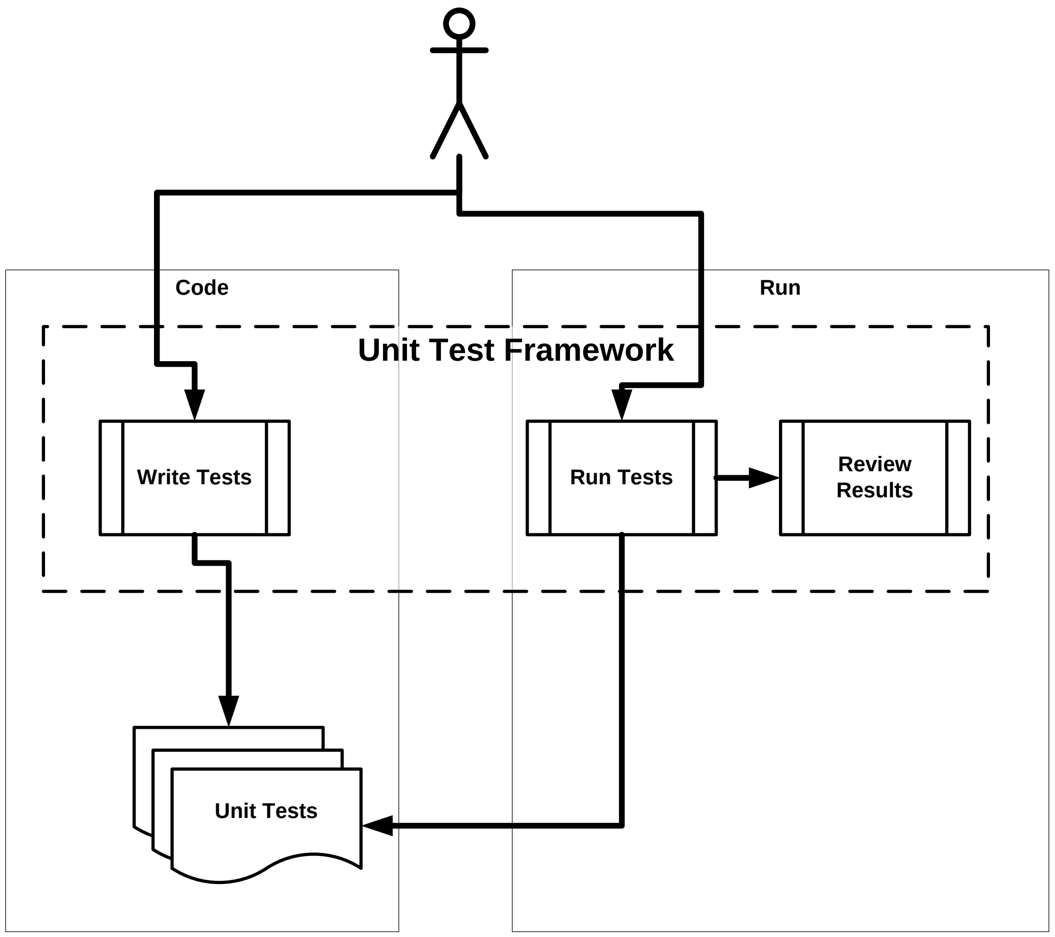


Figure 2.1 Unit tests are written as code using libraries of the unit test framework. Then the tests are run from a separate unit test tool and results are reviewed in the UI or as text results by the developer.

Unit test frameworks are code libraries and modules that help developers who would like to unit test their code do the practices shown in table 2.1.

Table 2.1 Using unit test frameworks helps developers write tests, execute them, and review failures easily

Practice	How
Write the tests easily and in a structured manner.	Framework supplies the developer with a class library that holds: ▪ Base classes or interfaces to inherit ▪ Attributes to place in your code to note your tests to run ▪ Assert classes that have special assert method you invoke to verify your code
Execute one or all of the unit tests.	Framework provides a Test Runner - console or GUI tool that ▪ identifies tests in your code ▪ runs them automatically ▪ gives you status while running ▪ can be automated by command line
Review the results of the test runs.	The test runners will usually let you know ▪ How many tests ran ▪ How many didn't run ▪ How many failed ▪ Which tests failed ▪ The reason they failed ▪ The ASSERT message you wrote in ▪ The code location that failed ▪ Possibly full stack trace of any exceptions that have caused the test to fail and let you go to the various method calls inside the call stack.

There are many unit test frameworks out there. A good list can be found at [http://www.xprogramming.com], there is practically one out there for every programming language in some form of public use. At the time of this writing, there are more than 150 of them, ranging in target language from C, C++, and .NET to Tcl, Ada, and AppleScript. .NET alone has at least nine different unit testing frameworks, among these, NUnit is the *de-facto* standard for writing unit tests in .NET.

2.1.1 The XUnit Frameworks

Collectively, these unit testing frameworks are called "The X-Unit Framework," since their names usually start with the first letters of the language for which they were built. Hence, you might have "CppUnit" for C++, JUnit for Java, "NUnit" for .NET, and "HUnit" for the Haskell programming language (though most of us are not in the profession long enough to have ever used it). Not all of them follow these naming guidelines, but most of them do.

In this book, I'll be using *NUnit*, the .NET flavor of a unit test framework. 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 a ecosystem of test frameworks that today is changing more and more.

Although many frameworks exist in .NET for unit testing, NUnit is considered the *de-facto* standard for use. Even today. The reason I chose .NET (and C#) for the examples and framework in this book is simply because it is the framework I'm most comfortable with, have used it the longest. Also, the concepts in it can be easily understandable to java and C++ developers alike. This chapter deals with writing your first test with NUnit and explains the NUnit behavior and usage.

Using a unit test framework, however, does *not* ensure that the tests we are writing are *readable, maintainable, or trust-worthy*, or that they actually cover all the logic that we would like to test. We will be talking about many ways that together join to provide exactly these properties for our unit tests in the chapter 7 and throughout this book.

One of those techniques deals with a common question: "*When* should I write a unit test?" – the answer brings up a technique known as "Test Driven Development," which is growing quickly in the developer community.

2.2 Introducing the LogAn project

The project in question will be a simple one at first, and will only contain one class. As the book moves along, we'll extend that project to new classes and features. Let's call this project the "LogAn Project".

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 and cannot be parsed by any of the existing 3rd party tools available today.

You are tasked with building a product that can analyze a log file and can find various special cases and events in the log file. When it finds them, it should alert various interested parties that these events have occurred.

In this book we'll be writing tests that verify parsing abilities, event recognition abilities and notification abilities by this product, "LogAn" (as in "**Log A**nd **N**otification").

2.2.1 Our first mission - Write a simple unit test with NUnit

Before we get started testing out our project, we'll find out how to write a unit test with NUnit.

As stated in the previous chapter, NUnit is one of the XUnit frameworks and allows doing 3 things: Write tests easily, run them easily and get the test results easily.

We'll look at each of these in detail. First, we'll have to install 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 should be rather simple.

2.3.1 Installing NUnit

You can download Nunit from `www.NUnit.org` or `www.NUnit.com` . Nunit is free to use and is an open source product, so you can actually get the source code for NUnit, compile it yourself on your own machine and use the source freely (as long as it stands in the terms of the open source license – see the `license.rtf` file that ins installed in the program directory for details.

NUNIT VERSION USED IN THIE BOOK

At the time of writing this book, the latest version for NUnit is 2.2.8. However, the examples in this book should be compatible with most future versions of the framework.

To install, simply run the setup program you downloaded. The installer will place a shortcut to the GUI part of the NUnit runner on your desktop, but he main program files should reside in a directory similar to "c:\Program Files\NUnit-Net-2.0 2.2.8".

If you double click on the desktop Icon for NUnit, here's the Unit Test Runner you'll see:

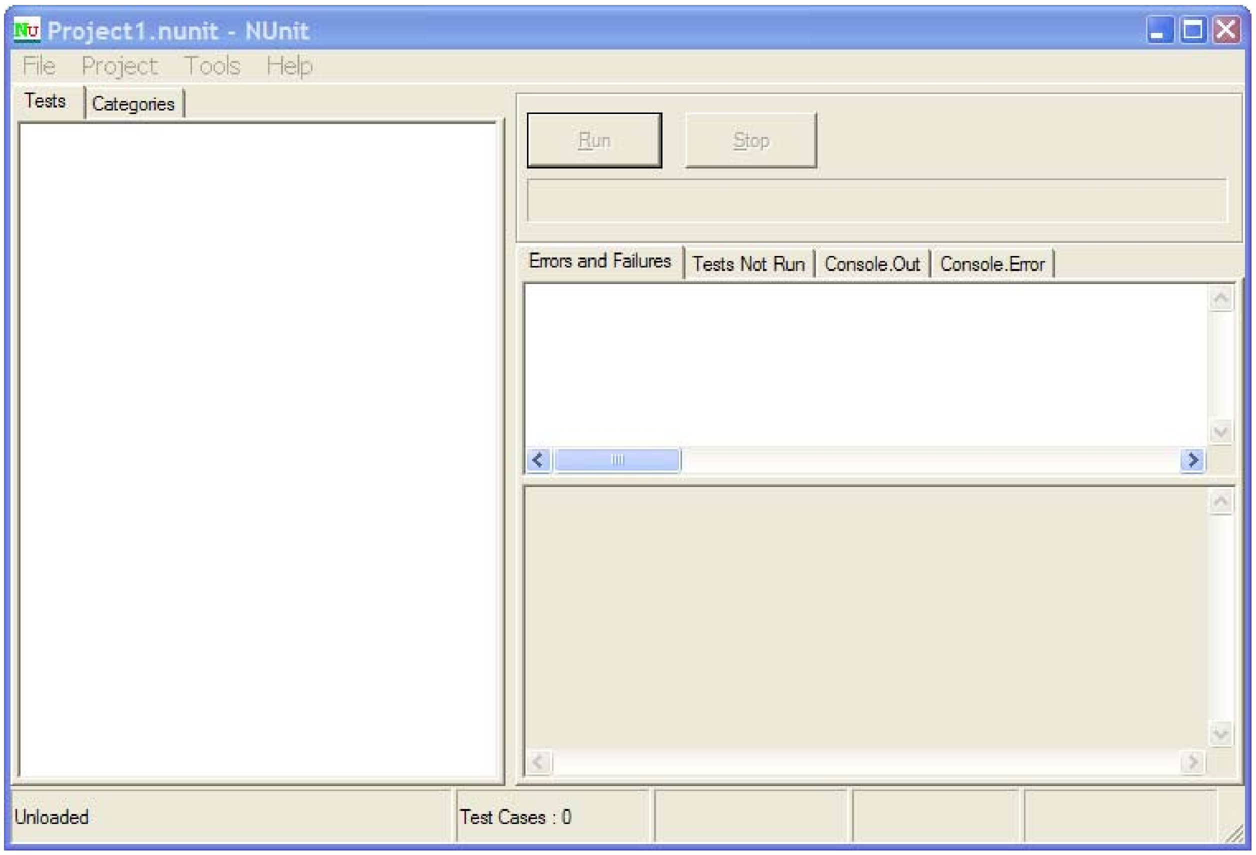


Figure 2.2 The NUnit Gui is divided into three main parts – the test tree on the left, messages and errors on the top right, and stack trace information on the bottom right.

The UI is divided into three main parts. On the left the list of tests and classes to be run automatically, and on the right are the results of any test failures, with the location of the test failure in code written at the bottom. We'll be using this UI to run our tests shortly.

2.3.2 Loading up the solution

If you have the book's code on your machine, load up the following solution inside Visual Studio 2005 (C# Express Edition should be fine, as well as Professional and any other version of Visual Studio 2005 with C# Installed). The solution to load is under the Code folder "CH2".

Let's begin by testing the following simple class with one single method (our unit to test) inside it:

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

The method may not seem complicated but we'd like to test it to make sure it actually works. In the real world you'd want to test any method that has any logic, even if it seems very simple. Logic can fail and we want to make sure we know when it does. In the following chapters we'll see more complicated scenarios and logic that we'll be running tests against. Consider this your easiest test for the book.

The method just looks at the file extension sent in to tell you if it's a valid log file or not. Let's consider a simple first test. Our first test will be to send in a non valid file name, and make sure that it returns "false".

The first steps you need to make in order to start writing an automated test for the IsValidLogFileName method are:

- Add a new class library project to the solution, which will contain your tests.
- To that library add a new class which will hold your tests
- Add a new method to test the IsValidLogFileName method.

We'll touch more on standards later in the book, but for now the basic rules are detailed in table 2.2:

Table 2.2: Basic rules for managing tests

Object you'd like to test	**Object to create on the testing side**	**Example**
For each project that contains production code we'd like to test	Have a counter "Test" project names `[ProjectUnderTest].Tests`	In this case the name for the new test project will be "`AOUT.Logan.Tests`".
For each class you will be testing	For each class you will be testing there should be at least one counter class with the name "[ClassName]Tests "	In our case the name for that test class would be "`LogAnalyzerTests`"
For each method we will be testing	have at least one test method with the following naming: "[MethodName]_[StateUnderTest]_[ExpectedBehavior]"	See detailed explanation below

Here's a small explanation on each part of the test name:

MethodName	So you know what method you are testing
StateUnderTest	What conditions are used to produce the expected behavior?
ExpectedBehavior	What do you expect the tested method to do under specific conditions?

In our case the state or condition is that we are sending the method a valid file name, and we expect the behavior to be returning a true value. So the test method name might look like this:

```
public void IsValidFileName_validFile_ReturnsTrue()
{
}
```

We haven't used the NUnit test framework yet, but we're close. You still need to add a reference to the project under test for your new testing project. The next thing to learn is how to mark your test method to be loaded and run by NUnit automatically.

2.3.3 Using the NUnit Attributes in your code

NUnit uses an attribute scheme to recognize and load tests that you've written in your code. Just like bookmarks in a book, these attributes help the framework know the important parts in the assembly that it will be loading and which parts are tests that need to be invoked.

NUnit provides an assembly that holds these special attributes. You will need to add a reference in your test project (not in your production code!) to the assembly named "`NUnit.Framework`". You can easily find it located under the .NET tab in the `Add Reference` dialog. Just type `Nunit` and you'll see several assemblies starting with that name. You only need to add the one titled `NUnit.Framework`.

NUnit needs at least these two attributes to know what to run:

- `[TestFixture]` - The `TestFixtureAttribute` can be put on a class to denote it as a class that holds Automated NUnit tests. Simply put this attribute on your new `LogAnalyzerTests` class. If you replace the word "Fixture" with "Class" it makes much more sense.
- `[Test]` – The TestAttribute can be put on a method to demote it as an automated test to be invoked. Simply put this attribute on your new test method.

Also, NUnit requires any test methods to be void and accept no parameters.

When you're done your test code should look like this:

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

    }
}
```

We still have not written the test, so let's do that.

2.4 Writing the first test

You can think of it like a "fill in the blanks" game. You've marked your class, and a method to be run. Now, whatever code you put inside your new test method will be invoked automatically by NUnit whenever you want.

But how do we test our code? A unit test usually comprises 3 main actions:

- *Arrange*: Create and setup objects
- *Act* on an object
- *Assert* that something is as expected.

Here's a simple piece of code that does all three, with the "Assert" part using the NUnit Framework's Assert class:

```
           [Test]
            public void IsValidFileName_validFile_ReturnsTrue()
            {
                       //arrange
LogAnalyzer analyzer = new LogAnalyzer();

//act
bool result = analyzer.IsValidLogFileName("whatever.slf");

//assert
Assert.IsTrue(result, “filename should be valid!”);
            }
```

Before we go on, you'll need to know a little more about the Assert class, as it is such an important and integral part of writing your unit tests.

2.4.1 The Assert Class

The Assert class is a class with static methods located in the Nunit.Framework namespace. It's the bridge between your code and the NUnit framework. The purpose is to declare that a specific assumption is supposed to exist. If the argument that are passed into the Assert class turn out to be different than what we are asserting, then NUnit will realize the our test has actually failed, and will alert us to this fact. We can even tell it what message to alert us with if the assertion fails. This is optional.

The Assert class has many Assert methods, with the main one being Assert.IsTrue (some Boolean expression). If all else fails, we can always use this method to verify some Boolean condition.

But there are many others. Here are some of the most important method signatures with a short explanation:

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

Verifies that some object or value is the same as the actual one.

Example:

```
    Assert.AreEqual(2, 1+1, “Math is broken”)
```

```
Assert.AreSame(Obj, Obj)
```

Verifies that the two arguments are actually a reference to the same object.

Example:

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

It’s really simple to use, learn and remember.

Now that we have the basics of the API, let’s get right to it.

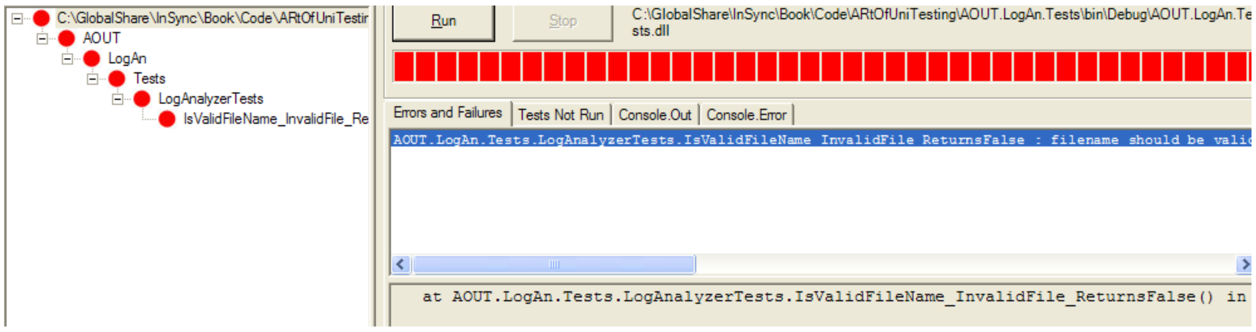
2.4.2 Running our first test with NUnit

It’s time to run our first test and see if it passes or not. To do that, we need to have a built assembly (dll file in this case) that we can give to NUnit to inspect. After you build the project, locate the path to the assembly file that was built

Load up the NUnit GUI and click File-Open. Put in the name of your tests assembly. You’ll see your single test and the class and namespace hierarchy of your project on the left.

Press the “Run” button to run your tests.

In the case of the code above, you may find that the test has actually failed!


C:\GlobalShare\InSync\Book\Code\ARtOfUniTestir
AOUT
LogAn
Tests
LogAnalyzerTests
IsValidFileName_InvalidFile_Re
Run
Stop
C:\GlobalShare\InSync\Book\Code\ARtOfUniTesting\AOUT.LogAn.Tests\bin\Debug\AOUT.LogAn.Tests.dll
Errors and Failures
Tests Not Run
Console.Out
Console.Error

```
AOUT.LogAn.Tests.LogAnalyzerTests.IsValidFileName InvalidFile ReturnsFalse : filename should be vali
```

```
   at AOUT.LogAn.Tests.LogAnalyzerTests.IsValidFileName_InvalidFile_ReturnsFalse() in
```

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 show on the right.

We 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.

2.4.3 Fixing our code and passing the test

A quick browse through the code reveals that we are testing for an uppercase extension to the file name, and in our test we’re sending in a lowercase file extension, which makes our code return false instead of true. Our test could have also failed if for any reason our code were to throw an exception of any kind – An unhandled exception in your code is considered a failure, unless your code is *supposed* to throw an exception under specific circumstances (we’ll see how to test for deliberate exceptions in section 2.5.2).

If we “fix” the production code to look like this:

```
            if(!fileName.ToLower().EndsWith(".slf"))
```

we can make the test pass. But in fact, this is a sign that perhaps the name of our test needs changing as well, and that we need another test, to make sure that sending in an Upper case’s extension works as well (we know that it works now, but who’s to say that in the future some programmer working on this feature won’t break it without realizing it?).

A better name for our current test might be:

```
public void IsValidFileName_validFileLowerCased_ReturnsTrue()
```

If you rebuild your solution now, you'll find that NUnit's GUI can detect that the assembly has changed, and will automatically reload the assembly in the GUI. If you re-run the tests now you'll see that the test passes with flying (green) colors.

2.5.4 From Red to Green

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

The "Red-Green" concept is prevalent throughout the unit testing world, and especially with the notion of "Test Driven Development". The mantra of TDD is "Red-Green-Refactor", meaning "Start with a failing test, then pass it, then make your code readable and more maintainable, now that you have tests for it. We'll touch on TDD in chapter 12 later in this book.

2.5 More NUnit Attributes

Now that you've seen how easy it is to create unit tests that can be run automatically, we'll talk a little more about setting up the initial state for each test, and removing any leftover "garbage" that was left by your test.

The life cycle of a unit test run in your code has specific points in its life that you'd want to have control over. Running the test is just one of them, and not even the first. Special `SetUp` methods run before that, as we'll see in the next section.

2.5.1 Setup and TearDown

In a way, a unit test is like going to a restaurant. You want to make sure that any leftovers from the previous guest's dinner are gone from the table you are sitting at, as if you're the first person to eat at that table that day.

For unit tests, it's very important that any leftover data or instances from the previous test will be destroyed and the state for the new test will be recreated as if no test has been run before it.

If you have left over state from the previous test you may find that your test fails, but only if it is run after a different test, and passes other times. Finding that kind of "dependency" bug between tests is hard and time consuming, and I don't recommend it to anyone. Having gone through this myself many times, having tests that are totally independent of each other is one of the best practices I will be going through in part II of the book.

In NUnit there are special attributes that allow an easier control of setting up and clearing out state before and after tests. these are called the "`Setup`" and "`TearDown`" actions. Figure 2.4 shows the process of running a test with setup and teardown actions.

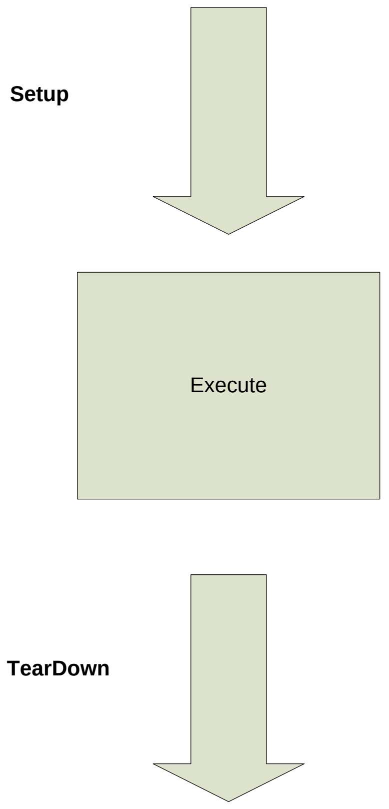


Figure 2.4 Nunit performs setup and teardown actions before each and every test method.

For now, make sure that each test you write is using a new instance of the class under test, so that no left over state messes up our tests.

We can take control of what happens in these steps by using two attributes from NUnit:

`[Setup]` – The `SetupAttribute` can be put on a method, just like a Test Method, and indicates to NUnit to run that Setup method each time it runs any of the tests in your class.

`[TearDown]` – the `TearDown` attribute denotes a method to be executed once after each test in your class has executed.

Listing 2.1 shows how we can use the Setup and Teardown class to make sure that each test in our class received a new instance of LogAnalyzer, while saving some repetitive typing at the same time.

Listing 2.1 : Using setup and teardown to setup state and clear it up after a test.

```
using NUnit.Framework;

namespace AOUT.LogAn.Tests
{
    [TestFixture]
    public class LogAnalyzerTests
    {
        private LogAnalyzer m_analyzer=null;

        [SetUp]
        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]
            public void TearDown()
            {
                m_analyzer = null;
            }

        }
    }
```

Another good way to think about the Setup and teardown methods is like constructors and destructors for the tests in your class. You can only have one of each in any test class, and each one will be performed once for each test in your class. In listing 2.1 we have two unit tests, so the execution path for NUnit will be something like figure 2.5:

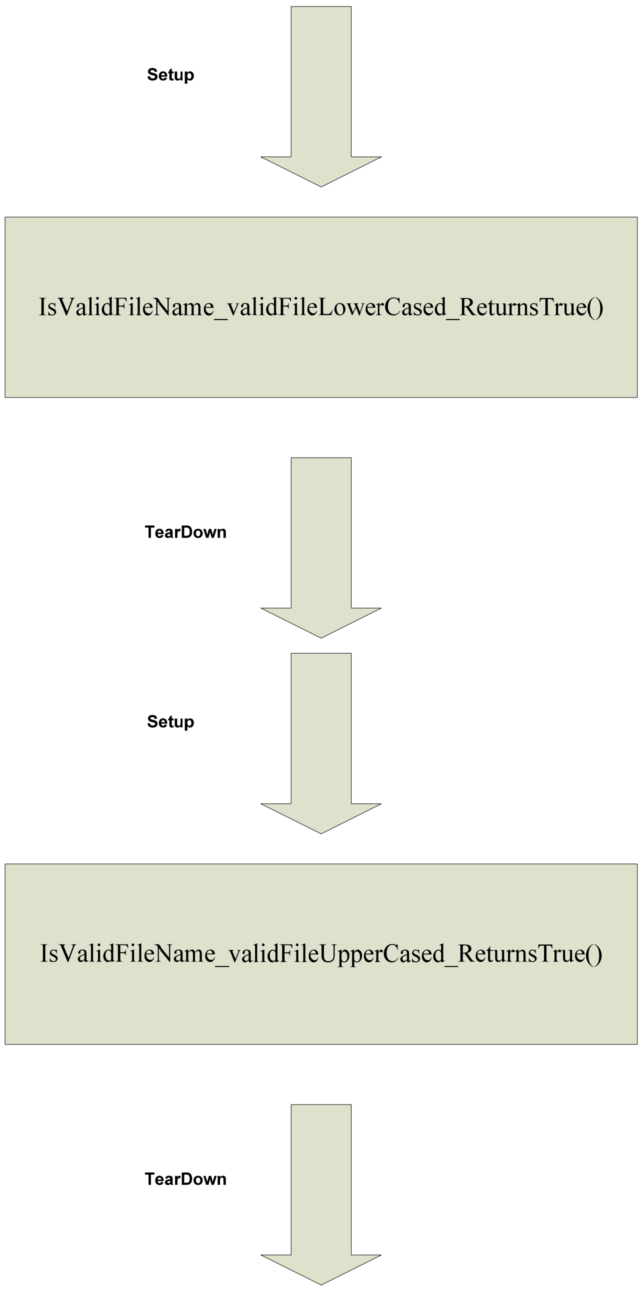

Setup
IsValidFileName_validFileLowerCased_ReturnsTrue()
TearDown
Setup
IsValidFileName_validFileUpperCased_ReturnsTrue()
TearDown

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.

NUnit contains several other attributes to help with cleanup and setup of state:

`[TestFixtureSetup]` and `[TestFixtureTearDown]` allow setting up state once before all the tests in the specific class run, and once after all the tests run (once per test fixture). Good for when setting up takes a long time, or cleanup takes a long time, and you want to only do it once per fixture, though you'll need to be cautious about using it. You may find that you are sharing state between tests if you're not careful!

`[AssemblySetup]` and `[AssemblyTearDown]` allow setting up state once before and after all the tests in the specific **assembly** run. This is used much less often.

Next, we'll take a look at how we can test that an exception is thrown from our logical code when it should.

2.5.2 Checking for expected exceptions

One of the common testing scenarios is to make sure that a correct exception is thrown from the tested method when it should. In our case, let's assume that our method should throw an `ArgumentException` when we send in an empty file name.

The interesting thing to note in this case is that if our code does **not** throw an exception, it means our test should fail.

So, if we're going to test this method logic in listing 2.2:

listing 2.2: Logic we'd like to test inside LogAnalyzer – File name validation.

```
public class LogAnalyzer
    {
        public bool IsValidLogFileName(string fileName)
        {
            if(!fileName.IsNullOrEmpty(filename))
            {
                Throw new ArgumentException(“No filename provided!”);
            }
            if(!fileName.EndsWith(".SLF"))
            {
                return false;
            }
            return true;
        }
    }
```

There is a special attribute in NUnit that helps us deal with these kinds of situations: the `[ExpectedException]` Attribute. Here's how a test that checks for the appearance of an exception might look like:

```
  [Test]
  [ExcpectedException(typeof(ArgumentException),”No filename provided!”)]
        public void IsValidFileName_EmptyFileName_ThrowsException()
        {
            m_analyzer.IsValidLogFileName(string.Empty);
        }
```

There are several important things to note here:

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

Given the method above and the test for it, this test should pass. Had our method *not* thrown an `ArgumentException` or the exception's message would be different than the one expected, our test would have failed saying that either an exception was not thrown or the message is different than excepted.

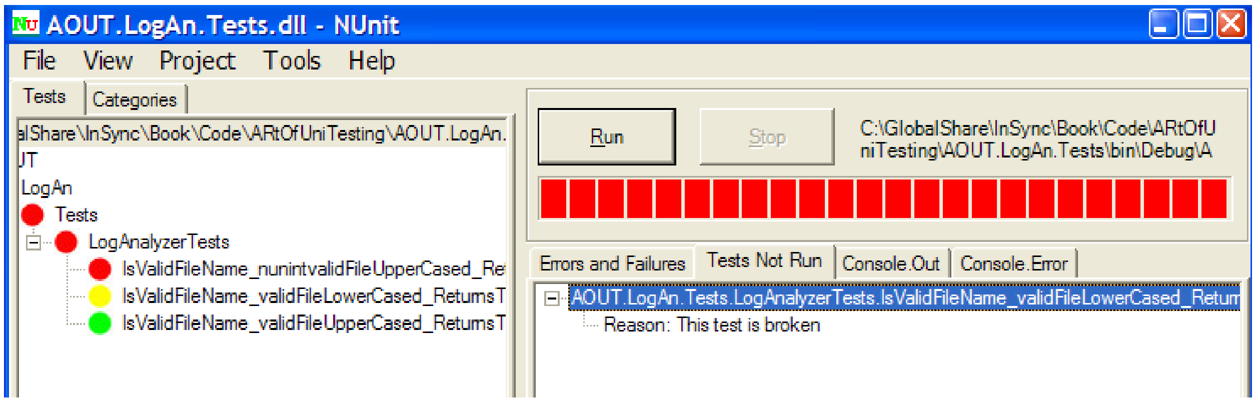
2.5.3 Ignoring tests

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 I do mean they should be very rare!) you can put a `[Ignore]` attribute on those 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 might provide us with a result similar to figure 2.6


AOUT.LogAn.Tests.dll - NUnit
File View Project Tools Help
Tests Categories
Run Stop
Errors and Failures Tests Not Run Console.Out Console.Error

```
alShare\InSync\Book\Code\ARtOfUniTesting\AOUT.LogAn.
JT
LogAn
Tests
LogAnalyzerTests
IsValidFileName_nunintvalidFileUpperCased_Re
IsValidFileName_validFileLowerCased_ReturnsT
IsValidFileName_validFileUpperCased_ReturnsT
```

```
C:\GlobalShare\InSync\Book\Code\ARtOfU
niTesting\AOUT.LogAn.Tests\bin\Debug\A
```

```
AOUT.LogAn.Tests.LogAnalyzerTests.IsValidFileName_validFileLowerCased_Return
  Reason: This test is broken
```

Figure 2.6 - Ignoring Tests in NUnit. The ignored test is marked in yellow(middle test), and the reason for not running the test is found under the “Tests Not Run” tab on the right.

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 as we’ll see next.

2.5.4 Setting test categories

You can set up your tests to run under specific test categories. For example “Slow tests” and “Fast Tests”. You do this by using the [Category] attribute from NUnit:

```
[Test]
[Category(“Fast tests”)]
        public void IsValidFileName_ValidFile_ReturnsTrue()
        {
///…
        }
```

When you load those tests in NUnit, you can see the tests based on categories instead of namespaces. Switch to the “Categories” tab in NUnit, then double click the Category you’d like to run so that it moves into the lower “Selected Categories” pane. Then click run. Figure 2.7 shows what the screen might look like when you click the categories tab.

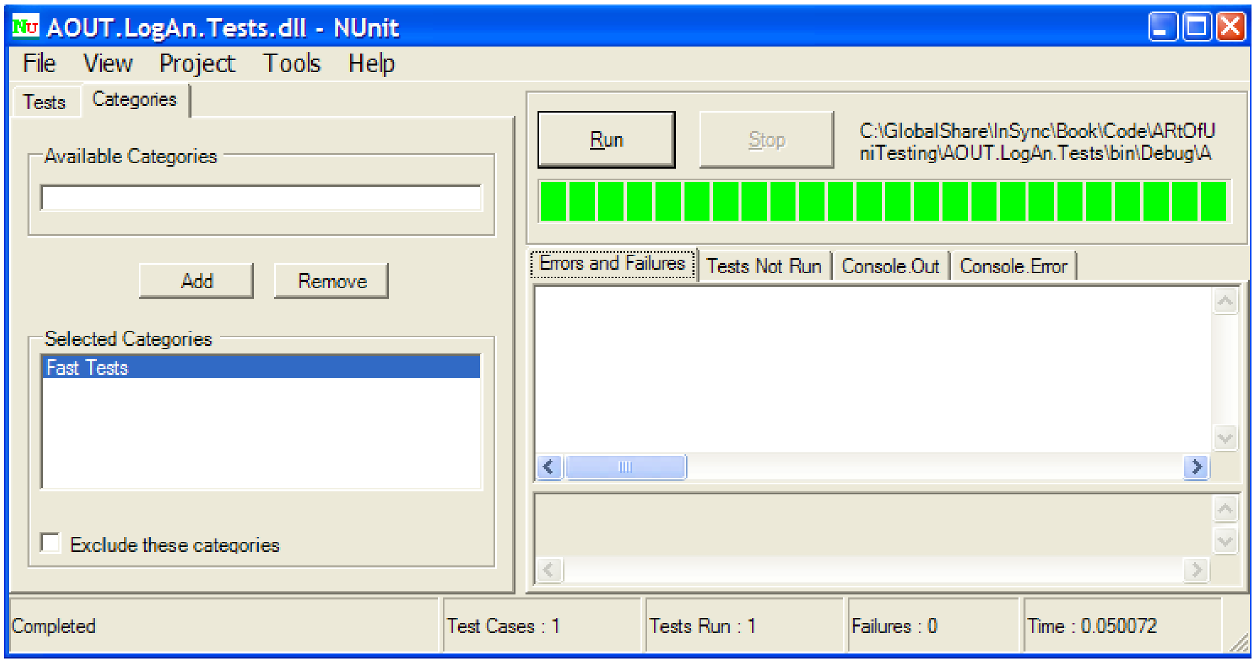

AOUT.LogAn.Tests.dll - NUnit
File View Project Tools Help
Tests Categories
Available Categories
Add Remove
Selected Categories
Fast Tests
Exclude these categories
Run Stop
C:\GlobalShare\InSync\Book\Code\ARtOfUniTesting\AOUT.LogAn.Tests\bin\Debug\A
Errors and Failures Tests Not Run Console.Out Console.Error
Completed Test Cases : 1 Tests Run : 1 Failures : 0 Time : 0.050072

Figure 2.7 Running categorized tests in NUnit can be done by setting up categories in the code base, and then choosing them to be run from the NUnit GUI.

So far we've ran very simple tests against methods that return some value as a result. What if our method does not return a value, but changes some state in the object?

2.6 Indirect testing of state

Let's consider a simple example using the `LogAnalyzer` class which we cannot test simply by calling one method in our test. Listing 2.3 shows the code for this class.

Listing 2.3: The property value has to be tested by calling the IsValidLogFileName method – an indirect testing technique.

```
public class LogAnalyzer3
    {
        private bool wasLastFileNameValid;

        public bool WasLastFileNameValid
        {
            get { return wasLastFileNameValid; }
            set { wasLastFileNameValid = value; }
        }

        public bool IsValidLogFileName(string fileName)
        {
            if (!fileName.ToLower().EndsWith(".slf"))
            {
                wasLastFileNameValid=false;
                return false;
            }

            wasLastFileNameValid = true;
            return true;
        }
    }
```

The bolded areas mark a new feature that we would like to test: `LogAnalyzer` remembers what the last outcome of a validation check was.

To test this new functionality, we can't simply write a test that gets a return value from a method, we have to use alternative means in order to see if the logic works. First we have to identify where the logic we are trying to test is located. Is it in the new property called WasLastNameValid? Not really – it's in the `IsValidLogFileName` method, so our test will start with the name of that method.

Listing 2.4 shows a simple test to see if the outcome is remembered:

Listing 2.4: Testing a class by calling a method and checking the value of a property.

```
 [Test]
        public void IsValidFileName_ValidName_RemembersTrue()
        {
            LogAnalyzer3 log = new LogAnalyzer3();
            log.IsValidLogFileName("somefile.slf");
            Assert.IsTrue(log.WasLastFileNameValid);
        }
```

Notice that we are testing the functionality of the `IsValidLogFileName` by asserting against a *different* location than the one we are testing.

Listing 2.5 shows another example to make things clearer. This time we'll be looking into the functionality of a built in memory calculator (Take a look at Calculator.cs and `CalculatorTests.cs` for the sample code.)

Listing 2.5: The Add() method cannot be tested directly. The test has to also call Sum() to see if the Add() works correctly.

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

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

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

Calculator works a lot like the pocket calculator you know and love – you can press a number, then hit "`add`', then press another number, then press "add" again. When you're done you can simply press 'Equals' and you'll get the total sum so far.

Where do you start testing the `Sum()` function? We always need to consider the simplest test to begin with. For example – testing that by default, Sum() returns zero, as shown in listing 2.6:

Listing 2.6: The simplest test for Calculator's Sum().

```
 [Test]
        public void Sum_NoAddCalls_DefaultsToZero()
        {
            Calculator calc = new Calculator();
            int lastSum = calc.Sum();
            Assert.AreEqual(0,lastSum);
        }
```

For any other test we'd like to write, we can't really do it without first invoking the Add() method. So our test will have to call Add and then assert against the number returned from `Sum`. Listing 2.7 shows our test class with the new test we're adding.

Listing 2.7: The two tests, with the second one calling the Add() method.

```
[SetUp]
        public void Setup()
        {
            calc = new Calculator();
        }

        [Test]
        public void Sum_NoAddCalls_DefaultsToZero()
        {
            int lastSum = calc.Sum();
            Assert.AreEqual(0,lastSum);
        }

        [Test]
        public void Add_CalledOnce_SavesNumberForSum()
        {
            calc.Add(1);
            int lastSum = calc.Sum();
            Assert.AreEqual(1,lastSum);
        }
```

Notice that this time I've structured the tests to initialize the `Calculator` object in a `[Setup]` method. This is a good idea because it saves time writing the tests, makes the code smaller and makes sure `Calculator` is always initialized the same way. It's also better for test maintainability because if tomorrow the constructor for Calculator changes, I only need to change the initialization in one place instead of going though each test and changing the `new` call.

So far so good. But what happens when the method we are testing depends on an external resource such as the file system, a database, as web service, or anything else that hard for us to control? That's when we start creating test stubs, fake objects and Mock Objects, which the next few chapters discuss.

2.7 Summary and best practices to take away from this chapter

Use this list of bullets as a friendly reminder of the various topics that you *must never* forget to check.

- Arranging tests in your solution:
  It is common practice to have a test class per tested class, a test project per tested project and at least one test method per tested method.
- Naming tests: [MethodUnderTest]_[Scenario]_[ExpectedBehavior]
- Use [SetUp] and [TearDown] attributes to re-use code in your tests such as creating and initializing common objects all your tests use.
- Don't use [Setup] and [TearDown] to initialize or destroy objects which are not shared *throughout* the test class in all the tests, as it makes the tests less understandable (the reader won't know which tests use the setup method and which don't).

In this chapter we learned about using NUnit to write simple tests against simple code. We used `Setup` and `TearDown` to make sure our tests always use new and untouched state, and we learned how to Ignore tests that need time to fix.

Test Categories help us divide the tests in a logical way rather than by class and namespace, and `[ExpectedException]` helps us make sure our code throws Exceptions when it should. Finally we started talking about what happens when you are not facing a simple method with a return value, and need to test the *end state* of an object. This method is very handy and you'll it in many of your future tests. It's not enough though.

Most test code has to deal with far more difficult coding issues, which is what our next chapter deals with.

We still haven't gotten to the "Art" part in this book. Every artist must first one the basic tools of the trade. This chapter and the next couple of chapters will give you the basic tools you'll work with when you start writing unit tests. Without that basic knowledge you won't be able to pick and choose what to do and when, for various difficult scenarios you'll come across.

In the next chapter we'll dive into more real world scenarios, where the code to be tested is a little more realistic than the one code you've seen so far. It has dependencies and testability problems, and we'll start discussing the notion of Integration tests vs. unit tests, and what that means to us as developers who write tests and want to ensure our code's quality.

3 Using Stubs to break dependencies

In the previous chapter we took a stab at writing our first unit test using NUnit, and explored the different kinds of attributes that are available to us as a test developer such as ExpectedException, SetUp, and `TearDown`. We also built tests for very simple use cases, where all we had to check on were simple return values from simple objects.

In this chapter we'll take a look at some more real world examples where the object under test actually relies on another object over which we have no control (or it does not work yet). That could be a web service, the time of day or threading, or many other things – the important point is that our test will have a hard time to control what that dependency returns to our code under test. That's when we use *stubs*.

One of the most difficult tasks man has created is flying people to outer space. It brings interesting challenges to the engineers and astronauts, one of the more difficult ones being how you make sure you're ready, as an astronaut to go into space and operate all the machinery.

Obviously you can't do a full on *integration test* for a space shuttle to find out if the astronauts do their job right, let alone simulate problematic situations. That's why NASA has full simulators that mimic the surroundings of a space shuttle control deck, thus removing the external dependency on having to be in outer space.

Space is a good example of what this section and most of this book will be dealing with – how do you control external dependencies in your code so that it would be easier to do things with it (like testing it).

If you take this notion of replacing something with something into the world of unit testing, that something is called a *stub*. Let's define that concept.

STUB: DEFINITION

A stub is a replacement for an existing dependency in the system, which your test can have control over. A stub will help you test your code without dealing with the dependency directly.

DEFINITION: EXTERNAL DEPENDENCY

An external dependency is an object in your system which your code under test interacts with, and over which you have no control (File System, Threads, Memory, time etc..)

So let's make things a bit more complicated for our `LogAnalyzer` by trying to untangle a dependency against the file system, as our next section describes.

3.1 Removing a file system dependency from LogAnalyzer

In our real application, the LogAnalyzer can be configured to handle multiple log file name extensions using a special adapter for each file. For the sake of simplicity let's assume that the allowed filenames are stored somewhere on disk as a configuration setting for the application, and that the `IsValidFileName` method looks like this:

```
public bool IsValidLogFileName(string fileName)
```

```
            {
              //read through the configuration file
              //return true if configuration says extension is supported.
            }
```

The problem that arises, as depicted in figure 3.1, is that once this test depends on the file system, we're dealing with an integration test, with all the problems associated with one – slower to run, needs configuration, tests multiple things and more.

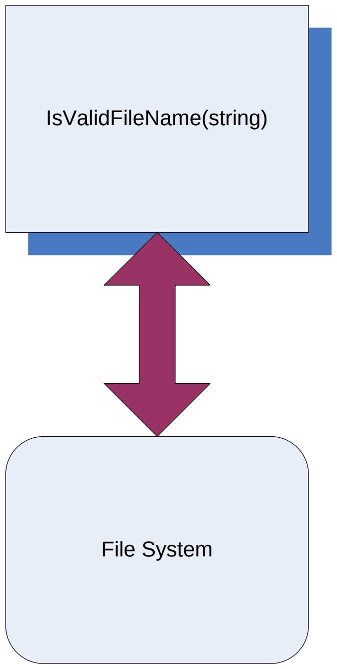


Figure 3.1 – Our method has a direct dependency on the file system. Our design inhibits us from testing it as a unit test – it promotes Integration Testing.

This is the essence of *test-inhibiting* design– it has some dependency that's hard to break on an external resource which may break the test even though the logic is perfectly valid. In legacy systems a single class or method may have many dependencies on external resources over which or test code has little if any control over. We'll be touching more on the notion of *Designing For Testability* later in chapter 11.

3.2 Our Goal: Easily Test this code

"There is no object oriented problem that cannot be solved by adding a layer of indirection, except, of course, too many layers of indirection.."

--A FRIEND OF MINE

I like the quote above because a lot of what the "Art" in the "Art of unit testing" is about is finding the right place to add or use a layer of indirection to achieve testability of the code base. You can't test something? Add a Layer that wraps up the calls to that something and then mimic that layer in your tests. Or make that something itself replaceable (so it is itself a layer of indirection) . The art is also to figure out when a layer of indirection is already there instead of having to invent it. Or to know when not to use it because it complicates things too much. But let's take it one step at a time.

The only way we can write a test for this code as it is, is to actually have something in the file system that helps our test – a configuration file. Since we're trying to avoid these kinds of dependencies, we want it to be easily testable without resorting to integration testing.

If we look at the first analogy in this section – the one about the astronauts, we'll see that there is a definite pattern for breaking the dependency:

Find the *interface* that the object under test works against; in our case this was the joysticks and monitors of the space shuttle, as depicted in figure 3.2.

Replace the *underlying implementation* of that interactive interface with something that you have control over: in our case it was hooking up the various monitors, joysticks and buttons into a control room in which test engineers were sitting and were able to control all the aspects of the space shuttle that the *interface* was showing to the astronauts under test.

Figure 3.2 Simple Space Shuttle Simulator, with joysticks and screens to simulate the "outside world".

Transferring this pattern to our code requires at least one more step:find the *interface* that the method under test works against; in our case this was the file system configuration file

If the interface is *directly connected* to our method under test (it is – we are calling directly into the file system), we make the code testable by adding a level of indirection to the interface:

1. Move the direct call to the file system to a separate class would be one way to add a level of indirection. There are others which will be covered. (See Figure 3.3 for how the design might look after this step.)
2. Once we have replaced the *underlying implementation* of that interactive interface with something that you have control over; in our case we will be replacing the instance of the class that our method calls to a class that we can control, thus giving our test code control over external dependencies.
3. Once we can replace the instance of the class, we can tell it *not* to talk to the file system at all, thus break out the dependency on the file system. Since we aren't testing the class that talks to the file system, but the calling code to this class, it's OK if that *stub* class we just created doesn't do anything but make "happy noises" when running inside the test.

See figure 3.4 for the design diagram after this alteration

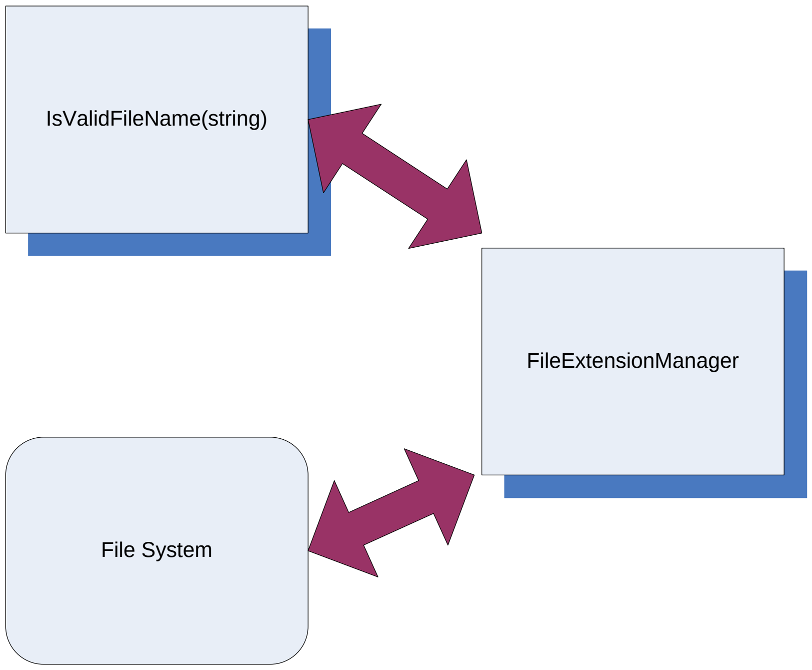


Figure 3.3 Introducing a layer of indirection to avoid a direct dependency on the file system. The code that calls the file system is separated into a FileExtensionManager class which does the work. Later we will replace that class with a stub in our test.

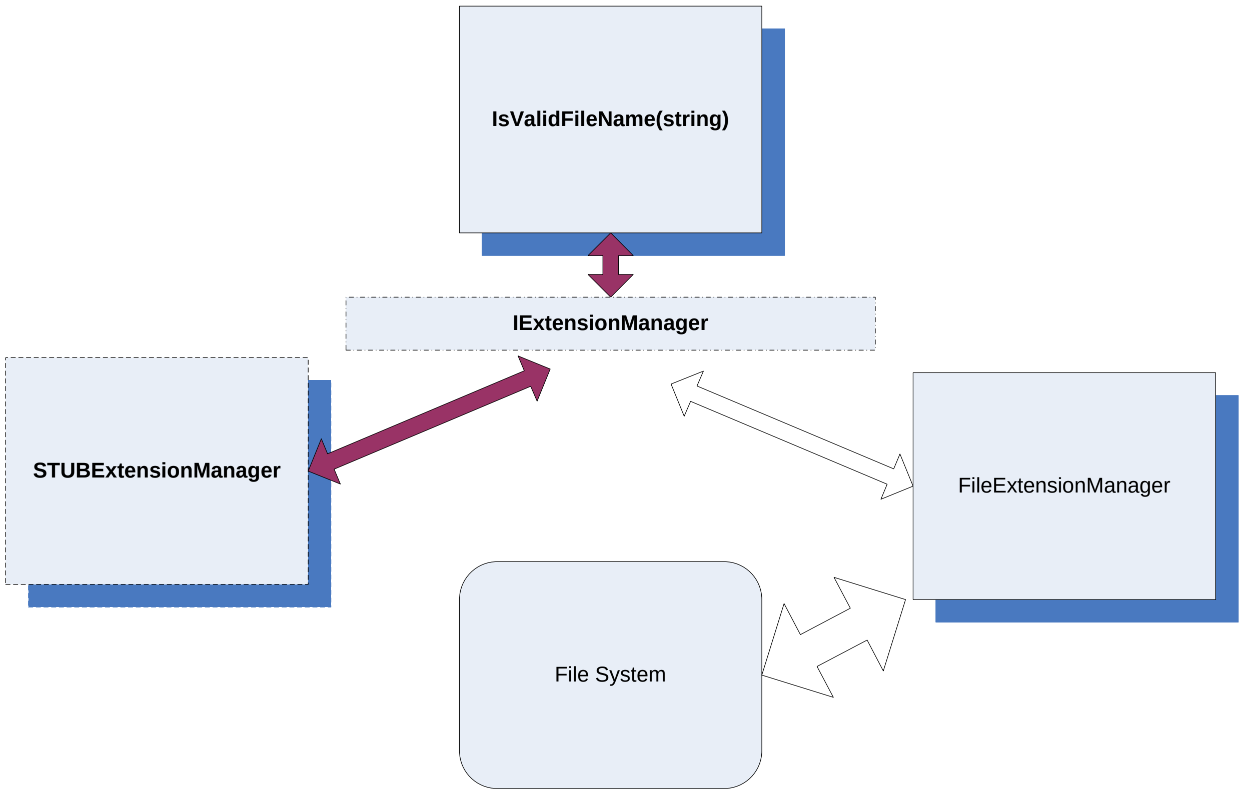


Figure 3.4 Introducing a stub to break the dependency. Instead of talking to the “real” FileExtensionManager class, our method will

talk with an abstract implementation of IExtensionManager (interface). Our test will replace the implementation of that interface with a STUB extension manager which will not depend in any way on the file system.

We've just gone through one way of introducing testability into our codebase(added IExtensionManager interface to our code), by creating a new interface. Let's take a closer look at the idea of code *refactoring* and introducing *seams* into our code, as described in the next section.

3.3 Refactoring our design to be more testable

I'm going to introduce two new terms to be used throughout the book. 'Refactoring' will refer to the act of changing the code's design without breaking existing functionality, and the 'Seam' will refer to that place in the code that allows us to 'plug in' our Stub classes.

REFACTORING: DEFINITION

The definition of "Refactoring" is improving a code's design and structure without changing the code's functionality. If you've ever renamed a poorly named method, you've done refactoring. If you've ever split a long method into several small method calls which do exactly the same thing you've refactored your code.

SEAMS: DEFINITION

Places in your code where you can plug in different functionality

If we wanted to break the dependency between our test and the file system, we can use some common design patterns in our code to help with this task and make our design more testable by introducing one or more *seams* into the code (see the definition of *seam* above). We just need to make sure that the current code does exactly the same thing. Here some patterns for breaking dependencies:

3.3.1 Extract an interface to allow replacing underlying implementation

First, we need to break out the code that touches the file system into a separate class, so that we can easily distinguish it and later replace the call to that class from our tested function (see figure 3.3 for a diagram). Listing 3.1 shows in bold the places where we change the code.

Listing 3.1: Extracting a class that touches the file system, and calling it from the original code.

```
public bool IsValidLogFileName(string fileName)
{
    FileExtensionManager mgr = new FileExtensionManager();
    return mgr.IsValid(fileName);
}

class FileExtensionManager
    {
        public bool IsValid(string fileName)
        {
            //read some file here and determine if it is valid.
        }
    }
```

Next, we can tell our method that instead of using the concrete FileExtensionManager class, it will deal with *some* form of ExtensionManager, without knowing its concrete implementation. In .NET this could be accomplished by either using a base class or an interface that `FileExtensionManager` would extend.

Listing 3.2 shows using a new interface in our design to make it more testable.

Listing 3.2: Extracting an interface from a known class.

```
public class FileExtensionManager : IExtensionManager
    {
        public bool IsValid(string fileName)
        {
          …
        }
    }
public interface IExtensionManager
    {
        bool IsValid (string fileName);
    }

public bool IsValidLogFileName(string fileName)
        {
            IExtensionManager mgr = new FileExtensionManager();
            return mgr.IsValid(fileName);
        }
```

See figure 3.4 for a visual diagram of this implementation.

We've simply created an interface with one IsValid (string) method, and made `FileExtensionManager` implement it (it still stays exactly the same, only now we can replace the "real" manager with our own "stub" manager to support our test.

We still haven't created the stub extension manager, so let's create that one right now, as seen in listing 3.3:

Listing 3.3: Simple stub code that always returns true. We simply implement the interface and return whatever value we'd like

```
public class StubExtensionManager:IExtensionManager
    {
        public bool IsValid(string fileName)
        {
            return true;
        }
    }
```

This stub extension manager will always return true no matter what the file extension is. We can use it on our tests to make sure that no test will ever have a dependency on the file system, but also to be able to create new and bizarre scenarios where we can simulate serious system errors like making the stub manager throw an `OutOfMemoryException` and see how the system deals with it.

So now we have an interface, and two classes implementing it, but our method under test still calls the "real" implementation directly:

```
public bool IsValidLogFileName(string fileName)
        {
            IExtensionManager mgr = new FileExtensionManager();
            return mgr. IsValid (fileName);
        }
```

We are still not there. We somehow have to tell our method to talk to our implementation rather than the original implementation of `IExtensionManager`. We need to introduce a *seam* into the code, where we can plug in our stub.

3.3.2 Inject Stub Implementation into a class under test

There are several proven ways to create seams – places in our code that allow us to inject an implementation of an interface into a class to be used in its methods. Here are some of the most notable:

Receive an interface at the constructor level and save it in a field for later use

- Receive an interface as a `property get/set` and save it in a field for later use

- Receive an interface just before the call in the method under test using
    - A factory method
    - A factory class
    - Variations on the existing techniques

Let's go through these possible solutions one by one and see why you'd want to use each.

3.3.3 Receive an interface at the constructor level

In this scenario we are adding a new constructor or adding a new parameter to an existing constructor that will accept an object of the interface type which we have extracted earlier (IExtensionManager).

The constructor then sets a local field in the class of the interface type for later use by our method or any other. Figure 3.5 shows the flow of injection in a more visual way.

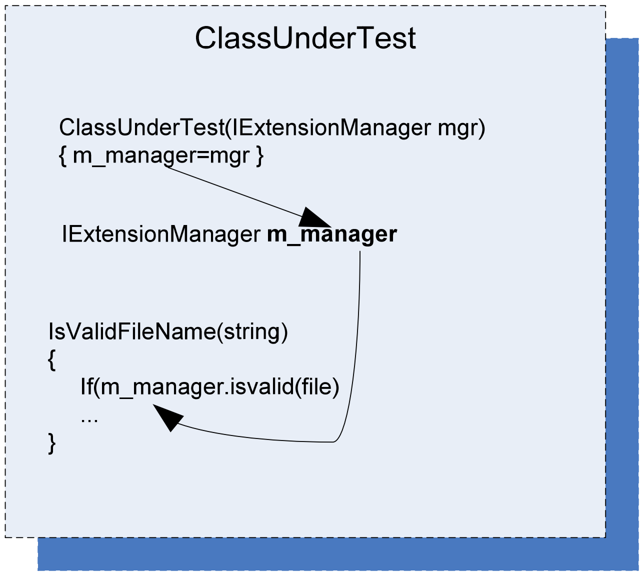

ClassUnderTest
ClassUnderTest(IExtensionManager mgr)
{ m_manager=mgr }
IExtensionManager **m_manager**
IsValidFileName(string)
{
If(m_manager.isvalid(file)
...
}

Figure 3.5 Flow of injection via a constructor.

This might get messy if you end up trying to replace many dependencies for a single test. See other methods for solving this problem.

Listing 3.4 shows how we would write a test for our `LogAnalyzer` class using a constructor injection technique:

Listing 3.4: Injecting our stub through a new constructor we created.

Production Code:

```
public class LogAnalyzer
    {
        private IExtensionManager manager;

        public LogAnalyzer ()
        {
                //this is what happens in production
            manager = new FileExtensionManager();
        }
        public LogAnalyzer(IExtensionManager mgr)
        {
          //     this constructor can be called by tests
            manager = mgr;
        }

        public bool IsValidLogFileName(string fileName)
        {
            return manager.IsValid(fileName);
        }
    }

public interface IExtensionManager
    {
        bool IsValid(string fileName);
    }

Test Code:

[TestFixture]
    public class LogAnalyzerTests
    {
        [Test]
        Public void
IsValidFileName_NameShorterThan6CharsButSupportedExtension_ReturnsFalse()
        {
            //setup the stub to use, make sure it returns true
            StubExtensionManager myFakeManager =
                                     new StubExtensionManager();
            myFakeManager.ShouldExtensionBeValid = true;

            //create analyzer and inject stub
            LogAnalyzer log =
                new LogAnalyzer (myFakeManager);

            //Assert logic assuming extension is supprted
            bool result = log.IsValidLogFileName("short.ext");
            Assert.IsFalse(result,
                              "File name with less than 5 chars should have
                              failed the method, even if the extension
                              is supported");
        }
    }

    internal class StubExtensionManager : IExtensionManager
    {
        public bool ShouldExtensionBeValid;

        public bool IsValid(string fileName)
        {
            return ShouldExtensionBeValid;
        }
    }
```

Notice that in the code sample, my test is actually testing some domain logic that is built **on top** of the call to the ExtensionManager. The domain logic in `IsValidFileName` should not only make sure that the extension is supported, but that the file name length is long enough to be considered.

You'll also notice another interesting fact relating to the stub object: it can be configured by the test code as to what Boolean value to return when its method is called. Listing 3.5 shows an example of configuring the stub from the test.

Listing 3.5: configuring the return value from a stub using the test code makes the test much more readable and understandable, and reveals the tester's intent.

```
public bool ShouldBeValid;

        public bool IsValid(string fileName)
        {
            return ShouldBeValid;
        }
```

Test:

```
            //setup the stub to use, make sure it returns true
            StubExtensionManager myFakeManager =
                                 new StubExtensionManager();
            myFakeManager.ShouldBeValid = true;

            //create analyzer and inject stub
            LogAnalyzer log =
                new LogAnalyzer (myFakeManager);
```

That way that stub class's source code can be reused in more than one test case, with the test setting the values for the stub before using it on the object under test. This also helps the readability of the test code, as the reader of the code does not have to go anywhere else but simply read the test and find out all the things going on in one place. Readability of tests is a very important aspect of writing unit tests, and we'll cover it much more later in the book.

There is one main problem that can arise from using constructors to inject implementations: once your code under test requires more than one stub to work correctly without dependencies, adding more and more constructors, or more and more constructor parameters becomes quite a hassle, and can even make the code less readable and less maintainable; here's a simple example:

Suppose `LogAnalyzer` also had a dependency on a web service, a logging service and of course, the file extension manager. The constructor might look like this:

```
public LogAnalyzer(IExtensionManager mgr, ILog logger, IWebService service)
 {
   //     this constructor can be called by tests
          manager = mgr;
          log= logger;
          svc= service;
   }
```

One possible solution to this maintenance problem is introducing the usage of Inversion of control containers – you can think of them as "smart factories" for your objects. You can ask such a container factory to give you an instance of an object (usually called by calling some "resolve" function) and it will try to automatically look at the constructor parameters of a class and initialize them with the correct variable instances that you configure it with. If your class needs an ILogger interface at its constructor, you can configure such a container object to always return the same ILogger object that you give it, when resolving this interface requirement.

The end result of using containers is usually a much simpler way of handling and retrieving objects, and less worry about the actual dependencies or maintaining the constructors. Some of the more popular containers in the .NET world are "Castle Windsor", Microsoft Unity and Spring.NET. There are many up and coming containers implementations such as Autofac, NInject and StructureMap which try to use a more fluent interface API, so be sure to look at them as well as you read more about this topic.

Dealing with containers is out of the scope of this book, but you can get started with your reading about them by reading Scott Hanselman's list here:

http://www.hanselman.com/blog/ListOfNETDependencyInjectionContainersIOC.aspx

WHY IS THE STUB ANALYZER LOCATED IN THE SAME CODE FILE?

Currently the Stub is used only from within this test class. It is far easier to located, read and maintain that stub in the same file than putting it in a different one. If later on I have an additional class that needs to use this Stub, I can move it to another file easily.

Now, imagine that you now have 50 tests against this constructor, and that you now find out that there is another dependency you had not considered (designs change all the time so this is not a remote possibility), such as a *factory service* for creating special objects of some sort that works against a database.

You'd have to add an interface for that, and of course, add that interface as a parameter to the current constructor. But you'd also have to change the call in 50 other tests that initialize the code. Also, your constructor is starting to look like it could use a facelift – it takes many parameters, and whatever logic it does have is beginning to be hidden by the many parameters. This is not a good place to be. Luckily, using property getters/setters can solve this problem easily, as we'll see in the next section.

When should you use this method vs. using properties?

My personal feeling is that without using helper frameworks, using constructor arguments as part of the initialization process for an object can make your testing code more cumbersome and likely to change. Every time you add another dependency to the class under test you have to create a new constructor that takes all the other arguments plus a new one, then make sure it calls the other constructors correctly, and also probably make sure other users of this class initialize it with the new constructor. That's why I usually go for the method listed next – using property getters and setters, which is a much more relaxed way to define "optional" dependencies.

But there are people who disagree with my approach. Those who do define constructor arguments as "non optional" arguments, and using properties strictly for optional ones. That way the semantics of the class also imply the proper usage of that class. I tend to agree somewhat with this point of view (you'll find that dilemmas on what to use when are very prevalent in the world of unit testing, which is a wonderful thing. Always question your assumptions, you might learn something new!) but my problem is with the maintainability aspect of this approach.

If you *do* however choose to use this approach, the proponents of this also likely use another library that helps them initialize those objects more easily. These are usually called Inversion Of Control (IoC) containers. A couple of well known containers that help with this are Spring.NET and the Castle project's Windsor Container. Both provide special factory methods that take in the type of object you'd like to create and any dependencies that it needs, and initialize the object using special configurable rules such as what contructor to call, what properties to set, in what order and so on. They are very powerful when put to use on a complicated composite object hierarchy where create an object requires creating and initializing objects several levels down the line.

That would be a great solution – if every code in the world were using IoC containers – but the sad truth is that most people don't really know what the *Inversion Of Control* pattern is, let alone what tools you can use to help make it a reality. I do think that the future of unit testing will see more and more use for these frameworks, and as that happens, we'll see clearer and clearer guidelines on how to design classes that have dependencies.

Constructor stubs are just one way to go. Properties as are often used as well.

3.3.4 Receive an interface as a property get/set

In this scenario we are adding a property get/set for each dependency we'd like to inject. We then use this dependency when we need it in our code under test. Figure 3.6 shows a visual flow of injection with properties:

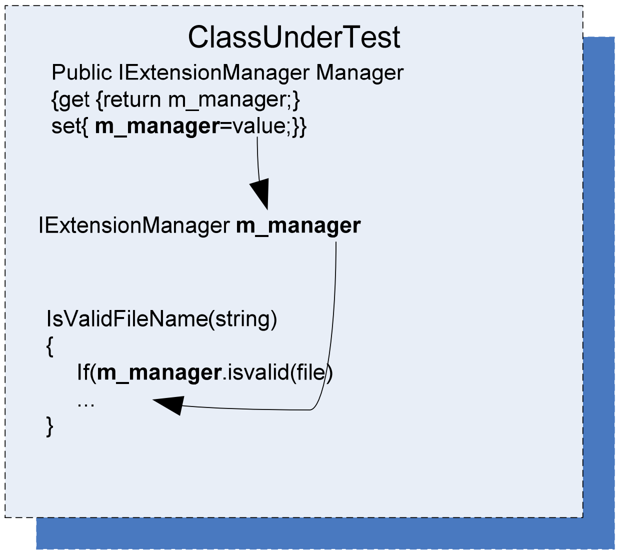

ClassUnderTest
Public IExtensionManager Manager
{get {return m_manager;}
set{ **m_manager**=value;}}
IExtensionManager **m_manager**
IsValidFileName(string)
{
If(**m_manager**.isvalid(file)
...
}

Figure 3.6: Using properties to inject dependencies. Much simpler than using a constructor because each test can set only the properties that it needs to get the test underway.

Using this technique (also called *Dependency Injection*, which can be used to describe the other techniques in this chapter as well) our test code could look quite similar to that in the earlier version with a constructor injection, but it is more readable and simpler to achieve, as shown in listing 3.6.

Listing 3.6: Injecting a stub by adding properties setters to the class under test.

```
public class LogAnalyzerPropertyStub
    {
        private IExtensionManager manager;

        public LogAnalyzerPropertyStub()
        {
            manager = new FileExtensionManager();
        }

        public IExtensionManager Manager
        {
            get { return manager; }
            set { manager = value; }
        }

        public bool IsValidLogFileName(string fileName)
        {
            return manager.IsValid(fileName);
        }
    }

        [Test]
        Public void
IsValidFileName_NameShorterThan6CharsButSupportedExtension_ReturnsFalse()
        {
            //setup the stub to use, make sure it returns true
                 ...

            //create analyzer and inject stub
            LogAnalyzer log =
                new LogAnalyzer ();
                log.ExtensionManager=myFakeManager;

               //Assert logic assuming extension is supported
                   ...
            }
        }
```

WHEN SHOULD YOU USE THIS?

If you don't have any factory classes that create the dependencies in your object, or the object itself, this is a good option, as is the first one (see my discussion on constructor arguments vs. properties on the previous section).

3.3.5 Use a Factory Class

In this scenario we go back to the basics where a class initializes the manager in its constructor, only it gets the instance from a Factory class. The *Factory* pattern is a design that allows a class to be responsible for creating objects. Our tests will configure the Factory class (which in this cases uses a static method to return an instance of an object that implements `IExtensionManager`) to return a STUB instead of the real implementation. Figure 3.7 shows this in a more visual manner.

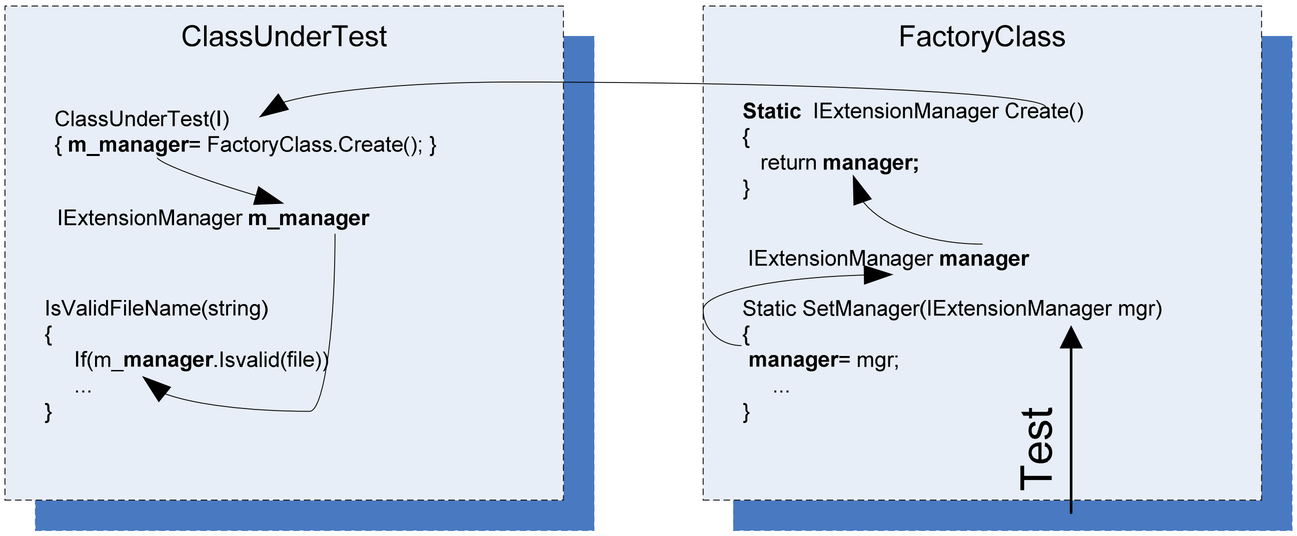

ClassUnderTest
ClassUnderTest(I)
{ m_manager= FactoryClass.Create(); }
IExtensionManager m_manager
IsValidFileName(string)
{
If(m_manager.Isvalid(file))
...
}
FactoryClass
Static IExtensionManager Create()
{
return manager;
}
IExtensionManager manager
Static SetManager(IExtensionManager mgr)
{
manager= mgr;
...
}
Test

Figure 3.7: A test configures the Factory class to return a Stub object. The Class under test uses the Factory class to get that instance, which in production code would return an object which is not a stub.

This is a very clean design and many object oriented systems use Factory classes to return instances of objects. However, most systems do *not* allow anyone outside the factory class to change the instance being returned, in order to protect the encapsulated design of the inner workings of this class.

In this case we've added a new setter method (our new *seam)* to the factory class so that our tests will have more control over what instance gets returned.

Using this technique our test code is easy to read, and there is a clear separation of concerns between the classes – each one is responsible for a different action.

Listing 3.7 shows how you would write code that uses the factory class in `LogAnalyzer` (including the tests)

Listing 3.7: Using and setting a factory class tor return a stub when our test is running.

```
public class LogAnalyzer
    {
        private IExtensionManager manager;

        public LogAnalyzer ()
        {
            manager = ExtensionManagerFactory.Create();
        }

        public bool IsValidLogFileName(string fileName)
        {
            return manager.IsValid(fileName)
                 && Path.GetFileNameWithoutExtension(filename).Length>5;
        }
    }

        [Test]
        Public void
IsValidFileName_NameShorterThan6CharsButSupportedExtension_ReturnsFalse()
        {
            //setup the stub to use, make sure it returns true
                 ...
                 ExtensionManagerFactory.SetManager(myFakeManager);
            //create analyzer and inject stub
            LogAnalyzer log =
                new LogAnalyzer ();

            //Assert logic assuming extension is supprted
                 ...
        }
    }
```

Also note that the implementation of the Factory class can vary greatly, and the examples show here only represent the simplest example of a Factory. For more and better examples of Factories, you can read about the *Factory Method* and the *Abstract Factory Design Patterns* online at

[http://www.dofactory.com/Patterns/Patterns.aspx].

The only thing you need to make sure of is that once you use these patterns, you add a seam to the factories you make so that they can return your stubs instead of the default implementations.

Many systems actually have a global #debug switch that, when turned on, many seams automatically send in fake or testable objects instead of default implementations. Achieving this in systems can be hard work, but it's worth it when it's time to test the system.

What if I don't want the seams to be visible in Release mode? There are several ways to achieve that. In .Net for example, you can put the seam statements (the added constructor or the setter or the factory setter) under a conditional compilation argument like this:

```
#if DEBUG
    MyConstructor(IExtensionManager mgr)
    {…}
#endif
```

There's also a special attribute in .NET that can be used for these purposes:

```
    [Conditional("DEBUG")]
    MyConstructor(IExtensionManager mgr)
    {…}
```

When should you use this?

First you need to realize that we are dealing with a different layer of indirection here.

Table 3.1 shows tree of indirection layers and possible seam you can use in this scenario:

Table 3.1 Layers of indirection and the seams that can be created from them. The deeper you go down the rabbit hole, the harder the test may be understand, but you also gain more manipulation power over your code under test.

Code under test	Possible Seam
1. Class Constructor	You can add a constructor argument to take a fake dependency
2. A call to the factory class	Replace the instance of the factory class with a factory that returns your fake dependency
3. The factory class returns the dependency you want to replace	Tell the factory class to return your fake dependency by setting a property.

The thing to understand about layers of indirection is that the deeper you go down the rabbit hole (that is, down the code base execution call stack) the better manipulation power you have over the code under test. But of course, this wouldn't be the wonderful world of code if there want's also a bad side to this, and there is. The deeper you get down the layers, the harder the test will be to understand, and the harder it will be to find the right place to put your seam in(please do *not* utter those two words together quickly).

The trick is to find the right balance between complexity and manipulation power so that your test remain readable but you get full control of the situation under test at the same time.

In this specific scenario adding a contructor level argument would complicate things when we already have a good possible target layer for our seam – layer number 3. Why is layer number 3 the simplest to use in this scenario? Because the changes in code it requires are minimal:

Layer 1: You would need to add a contructor, set the class in the contructor, and set it from the test, and worry about future uses of that API in the production code. You actually change the semantics of using the class under test which is preferably avoided unless you have a good reason to do so.

Layer 2: You would need to create your own version of a factory class, which many or may not have an interface, which means also creating an interface for it. Then you create your fake factory instance, tell it to return your fake dependency class (a fake returning a fake – take note!) and then set the fake factory class on our class under test. A fake returning a fake is always a bit of a mind boggling scenario which we'd like to avoid if we can, it makes the test less understandable.

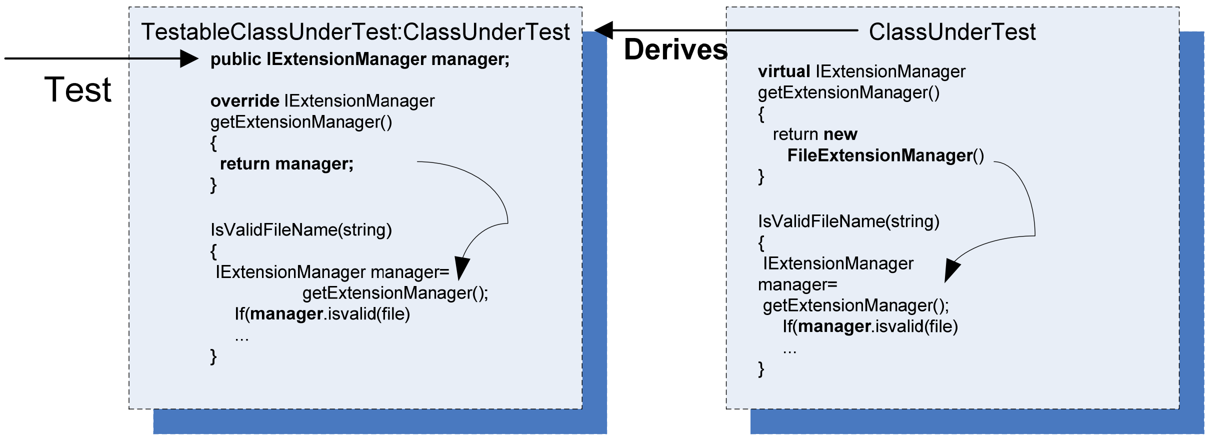
Layer 3: Very easy – add a setter to the factory, set it to a fake dependency of your choice. No changing semantics of the code base, everything stays the same, and the code is dead simple. The only con is that this requires to understand intimately who calls the factory and when – which means it needs some research before you can implement this easily. Understanding a code base you never saw before is a daunting task, but it still feels to me more reasonable than the other options.

3.3.6 Use a Local Factory Method

If you take a look at the table at the end of the previous section, this method does not reside in any of the layers. In fact we create a whole new layer of indirection very close to the surface of the code under test. The closer we get to the surface of the code the less we need to muck around with changing dependencies. In this case, the class under test is both under test and sort of a dependency that we need to manipulate.

In this scenario we use a local *virtual* method in the class under test, which is used as a factory to get the instance of the Extension Manager. The *seam* lies in the fact that the method is marked as *virtual* and thus can be overridden in a derived class. The way we inject our stub into the class is by *inheriting* a new class from the class under test, then *overriding* the virtual factory method, and finally returning whatever instance the new class is configured to return in the overriding method. The tests are then preformed on the new derived class. The factory method could also be called a *stub method*, that returns a *stub object*.

Figure 3.8 shows the flow of object instances more visually.



```
public IExtensionManager manager;

override IExtensionManager
getExtensionManager()
{
  return manager;
}

IsValidFileName(string)
{
 IExtensionManager manager=
              getExtensionManager();
    If(manager.isvalid(file)
    ...
}
```

```
virtual IExtensionManager
getExtensionManager()
{
   return new
     FileExtensionManager()
}

IsValidFileName(string)
{
 IExtensionManager
manager=
 getExtensionManager();
    If(manager.isvalid(file)
    ...
}
```

Figure 3.8: We inherit from the Class under test so that we can override its virtual factory method and return whatever object instance we want, as long as it implements IExtensionManager. Then we perform our tests against the newly derived class.

Here are the steps to using a factory method in your tests.

In the class under test:

- Add a virtual factory method that returns the real instance
- Use the factory method in your code as usual

Create a new class in your test project:

- The new class will inherit from the class under test
- Create a **public** field (no need for property get/set) of the interface type you want to replace (`IExtensionManager`)
- Override the virtual factory method
- Return the public field

In your test code:

- Create an instance of a stub class that implements the required interface (`IExtensionManager`)
- Create an instance of the **newly derived class**, and *not* of the class under test
- Configure the new instance's public field you created earlier and set it to the stub you have instantiated in your test.

When you test your class now, your production code will be using your stub through the overridden factory method.

Here's how the code might look when using this method. The Production class can be seen in listing 3.8.

Figure 3.8: Using and configuring a factory method to return a value requested by the test.

```
public class LogAnalyzerUsingFactoryMethod
    {
        public bool IsValidLogFileName(string fileName)
        {
            return GetManager().IsValid(fileName);
        }

        protected virtual IExtensionManager GetManager()
        {
            return new FileExtensionManager();
        }
    }
The Test Code:

[TestFixture]
    public class LogAnalyzerTests
    {
        [Test]
        public void overrideTest()
        {
            StubExtensionManager stub = new StubExtensionManager();
            stub.ShouldBeValid = true;

            TestableLogAnalyzer logan = new TestableLogAnalyzer();
            logan.Manager=stub;
            bool result = logan.IsValidLogFileName("file.ext");
            Assert.IsFalse(result,
          "File name should be too short to be considered valid");
        }
    }

    class  TestableLogAnalyzer:LogAnalyzerUsingFactoryMethod
    {
        public IExtensionManager Manager;

        protected override IExtensionManager GetManager()
        {
            return Manager;
        }
    }
    internal class StubExtensionManager : IExtensionManager
    {
          //no change from the previous samples
        ...
    }
```

The technique we are using here is called “Extract & Override” and you will find it extremely easy to use once you’ve done it a couple of times. It is a very powerful technique and one I will put to other uses throughout this book. You can find more about this dependency breaking technique and more techniques in a book I have found to be worth its weight in gold, called “Working Effectively with Legacy Code” By Michael Feathers.

WHEN SHOULD YOU USE THIS?

Extract & Override is a very powerful technique because it lets you deal directly with replacing the dependency without going down the rabbit hole (changing dependencies deep inside the call stack). That makes it very quick and clean to perform, almost to the point where it corrupts your good sense of object oriented aesthetics, leading you to code that might even have less interfaces but more virtual methods. I like to call this method “Ex-CRACK & Override” since it’s such a hard habit to let go of once you know it.

E&O is great for simulating *inputs* into your code under test, but it’s cumbersome when you want to verify interactions that are coming *out* of the code under test into your dependency.

For example, it’s great if your test code calls some web service and gets a *return value*, and you’d like to simulate your own return value. But it gets bad quickly if you want to test that your code *calls out* to the web service correctly with the correct value as an input to the web service. *That* requires lots of manual coding. mock frameworks are better suited for such tasks(as you’ll see in the next chapter). So, E & O is good if you’d like to simulate return values or simulate whole interfaces as return values, but not to check interactions between objects.

When I need to simulate inputs into my code under test, I use this technique a lot since it helps keep the changes to the semantics of the code base (new interfaces, constructors etc…) a little more manageable (since you need to make much less of them to get the code into a testable state). The only times I don't use this technique is when the code base clearly shows that there's a path already laid out for me: there's already an interface somewhere ready to be faked, there's already a place where a seam can be injected. When these things don't exist, and the class itself is not sealed (or can be made non-sealed without too much resentment from your peers) I would check out this technique first and only then move to more complicated pastures.

3.4 Variations on refactoring techniques

There are many variations on these simple techniques to introduce *seams* into source code. For example, instead of adding a parameter to a constructor, you can add it directly to the method under test. Instead of sending in an interface, you could send a base class and so on. Each has its own strength and weaknesses.

One of the reasons you may want to avoid using a base class instead of an interface is that a base class from the production code may have (probably has) already built in production dependencies, which you will have to know about and override as necessary. It makes things harder than to simply implement an interface, which lets you know exactly what the underlying implementation is and gives you full control over it.

In chapter 4 we'll talk about techniques that can help you avoid writing manual stub classes that implement interfaces, and instead use frameworks that can helps do this at runtime.

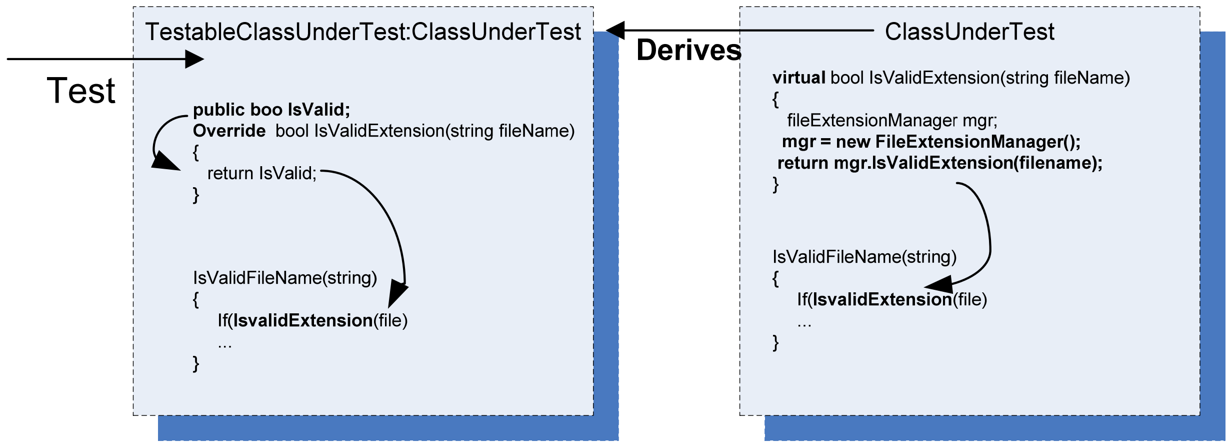
But for now, let's see another way to gain control over the code under test *without* using interfaces. You've already seen a variation of this in the previous pages, but this method is so effective, it deserves a full section of its own.

3.4.1 Extract & Override to create Stub Results

If you've read the previous section, you've already seen an example of Extract & Override in figure 3.15. We derive from the class under test so that we can override a virtual method and force it to return our stub.

But why stop there? What if you were unable or unwilling to add a new interface every time you needed control over some behavior in your code under test? In those cases, E&O can help simplify things, as it does not require writing and introducing new interfaces, but only the ability to derive from the class under test, and to override some behavior in the class.

Figure 3.9 shows a different way we could have forced the code under test to always return true about the validity of the file extension:



```
public boo IsValid;
Override  bool IsValidExtension(string fileName)
{
   return IsValid;
}

IsValidFileName(string)
{
    If(IsvalidExtension(file)
    ...
}
```

```
virtual bool IsValidExtension(string fileName)
{
   fileExtensionManager mgr;
  mgr = new FileExtensionManager();
 return mgr.IsValidExtension(filename);
}

IsValidFileName(string)
{
     If(IsvalidExtension(file)
     ...
}
```

Figure 3.9: Extract & Override to simply return a logical result instead of calling an actual dependency. No stub, just a fake result.

This time, in the class under test, instead of *virtualizing* a factory method, we *virtualize* the actual calculation result. That means that in our derived class, we override the method and simply return whatever value we want, without needing to create an interface, or creating a new stub. We simply inherit and override the method to return the desired result.

Listing 3.9 shows how our code might look using this technique without resorting to stub objects:

Listing 3.9: Returning a result rather than a stub object from an extracted method. This sometimes makes things very simple for the tests.

Class under test:

```
public class LogAnalyzer
    {
        public bool IsValidLogFileName(string fileName)
        {
            int len = fileName.Length;
            return this.IsValid(fileName) && len>5;
        }

        protected virtual bool IsValid(string fileName)
        {
             FileExtensionManager mgr = new FileExtensionManager();
           return mgr.IsValid(fileName);
        }
    }

Test Code:

[Test]
        public void overrideTestWithoutStub()
        {
            TestableLogAnalyzer logan = new TestableLogAnalyzer();
            logan.IsSupported = true;

            bool result = logan.IsValidLogFileName("file.ext");
            Assert.IsFalse(result,"...");
        }

class TestableLogAnalyzer: LogAnalyzerUsingFactoryMethod
    {
        public bool IsSupported;

        protected override bool IsValid(string fileName)
        {
            return IsSupported;
        }
    }
```

WHEN SHOULD YOU USE THIS?

Take a look at the previous section's "When should you use this?" header for the basic motivation. This technique is even simpler than the previous one and the same motivations apply. If and when I can, I use this technique over the previous one.

By now you may be thinking to yourself that adding all these constructors, setters and factories simply for the sake of testability is problematic. It breaks some serious Object Oriented principles, especially the idea of encapsulation (which basically says "hide everything that the user of your class does not need to see"). Appendix A deals with testability and design issues.

3.5 Overcoming the encapsulation problem

In every team that I get to meet where I present my ideas for testable designs, there are always those who feel that opening up the design to be more testable is a bad thing because it hurts the "object Oriented" principles the design is based on. I can wholeheartedly say to those people now "Don't be silly".

Object oriented techniques are there to try to enforce some constraints on the end user of the API (end user being the programmer who will use your object model) so that the object model is used properly and is protected from unforeseen ways of usage. OO also has a lot to do with Reuse of code and the single responsibility principle (each class has only one single responsibility).

When we write unit tests for our code, we've just added another end user to the object model – that end user is just as important as the previous one, but it has different goals when using the model. That end user (the test) has specific requirements from the object model that seem to defy the basic logic behind a couple of OO principles. Mainly, encapsulation. Encapsulating those external dependencies somewhere without the ability for anyone to change them, having private constructors or sealed classes, having non virtual methods that canot be overridden, all these are classic signs of what may sometimes be over protective design (Security related designs are a special case which I forgive). The problem is that the 2nd end user of the API, the test, needs them as a feature in our code.

I call the resulting design that emerges from designing with testability in mind – TOOP – Testable Object Oriented Design, and you'll hear more about TOOP in chapter 11 – Designing for Testability.

As I was saying, the concept of *testable designs* (which is what we are trying to achieve by refactoring our code base) has some collisions with the concept of *object oriented design*. If you *really really* need to consolidate both of these worlds and try to keep the cake and eat it too, here are a few tips and tricks you can use to make sure that the extra constructors and setters don't show up in release mode, or at least, don't play a part in release mode.

3.5.1 Using Internal and [InternalsVisibleTo]

If you dislike having a public constructor added to your class that everyone can see, you can make it `internal` instead of `public`. You can then expose all `internal` related members and methods to your test assembly by using the `[InternalsVisibleTo]` assembly level attribute. Listing 3.10 shows this more clearly:

Listing 3.10: Making a constructors internal and using an assembly level attribute to expose all internals in this assembly to the tests assembly.

```
public class LogAnalyzer
    {
      …
      internal LogAnalyzer (IExtensionManager extentionMgr)
        {
            manager = extentionMgr;
        }
    …
}

//this can go into the assembly.info file
using System.Runtime.CompilerServices;
[assembly: InternalsVisibleTo("AOUT.CH3.Logan.Tests")]
```

It's important to note that using conditional compilation constructs in your production code can quickly deteriorate its readability and increase its "spaghetti-ness". Beware.

3.5.2 Using the [Conditional()] attribute

The `System.Diagnostics.ConditionalAttribute` attribute is notable in its non intuitive action. When you put this attribute on a method, you initialize the attribute with the string signifying a conditional build parameter that is passed in as part of the build (`"DEBUG"` and `"RELEASE"` are the two most common ones and Visual Studio uses them by default according to your build type).

If the build flag is *not* present during the build, all the *callers* to the annotated method will not be included in the build. For example, the method:

```
[Conditional("DEBUG")]
Public void DoSomething()
{
}
```

Will have all the callers to it removed during a release build, but the method itself will stay on.

You can use this attribute on methods (but not on constructors) that you only want called in certain debug modes. But those methods won't be hidden from the production code, which is different from how our next method behaves.

3.5.3 Using #if and #endif with conditional compilation

Putting your methods or special test-only constructors between #if-#endif constructs will make sure they only compile when that build flag is set as shown in listing 3.11:

Listing 3.11 : placing the test and the special test-only constructor under special build flags makes sure the constructor only appears when not in release mode.

```
#if DEBUG
        public LogAnalyzer (IExtensionManager extentionMgr)
        {
            manager = extentionMgr;
        }
#endif
…
#if DEBUG
        [Test]
        public void IsValidFileName_NameShorterThan6CharsButSupportedExtension_ReturnsFalse()
        {
...
            //create analyzer and inject stub
            LogAnalyzer log =
                new LogAnalyzer (myFakeManager);
            ...
        }
#endif
```

Using this method is quite common, but can sometimes lead to code that looks very messy. Consider using the `InternalsVisibleTo` attribute where you can, for clarity.

3.6 Summary and Best Practices to take away from this chapter

In this chapter we started diving into the world of Stubs. There are several basic ways to implement a *seam* in your code, a place where you can plug-in a different implementation of behavior like a class or method. When you use interfaces you gain the ability to use stub objects that you pass in as part of your test, but you don't want to bother with interfaces you can use the Extract & Override method to simply return final results from various dependent operations (instead of talking to a web service, for example, you simply simulate the fact that it returns a result without actually making the call to the web service).

We started writing simple tests in the first couple of chapters, but saw that sometimes we have dependencies in our tests that we need to find a way to override. We learned how to 'Stub' out those dependencies in this chapter, using interfaces and inheritance.

A stub can be injected into your code in many different ways. The real trick is to locate the right layer of indirection, or to create one, and then use it as a *seam* from which you can inject your stub into running code.

The deeper you go down the layers of interactions, the harder it will be to understand the test, and to also understand the code under test and it's deep interactions with other objects. The closer to the surface of the object under test you are, the easier your test will be to understand and manage, but you may be also giving up some of your power to manipulate the object under test's environment.

Testable Object Oriented Design can present some interesting advantages vs. “classic” Object Oriented designsuch as allowing maintainability while still allowing tests to be written against the codebase. .

Learn the different ways of injecting a stub into your code. When you master them, you will have a much better position to pick and choose what to use when.

Extract & Override is great for simulating inputs into the code under test, but if you are also testing interactions between objects (topic of the next chapter) be sure to have it return an interface rather than an arbitrary return value. It will make your testing life easier.

If you really have to, you have several ways of “hiding” your testable design in release mode. See the [InternalsVisibleTo] attribute and the `[Conditional]` attribute.

In the chapter 4 we’ll take a look at some other issues relating to dependencies, and find ways to resolve them:

- How can you avoid writing manual stubs for interfaces?
- How do you test interaction between objects as part of your unit tests?

4
Interaction testing using Mock Objects

Mock Objects. *Scary.* Not really. It's actually a much simpler concept than you think. Just follow my lead as you read this chapter. But first, a recap.

In the previous chapter we solved the problem of our code under test depending on other objects to run correctly. We used *stubs* to make sure that the code under test got everything it needed so that we could test its logic independently of any other logic it relies upon.

In this chapter we will attempt to solve a bigger issue; how do you test that an object simply calls other objects correctly? It may not return any result, or save any state, but it has complex logic which results in correct calls to other objects to do the correct work. How do you test that your object indeed *interacts* with other objects correctly? That's the subject of this chapter – we'll use *Mock Objects*.

Before we continue though, let's try to define what *Interaction Testing* is, and how it's different from the testing we've done so far – *State-Based testing*.

4.1 State vs. Interaction Testing

Let's define the two concepts, and then see how we use them in our unit tests.

INTERACTION TESTING: DEFINITION

Interaction Testing is the act of testing how an object under test calls and receives input from other objects.

STATE-BASED TESTING: DEFINITION

Also called "State Verification", we determine whether the exercised method worked correctly by examining the state of the system under test and its collaborators (dependencies) after the method was exercised.

Let's say that you had a watering system, and you had given your system specific instructions on when to water the tree in your yard, how many times a day, and what quantity of water each time. Here's how you'd test that it's working correctly in State and Interaction Based testing:

- **State based testing**—You'd fire off the system for a specific number of hours (say 12). At the end of the 12 hours you'd need to check the state of the tree being irrigated: Is the land moist enough, is the tree doing well, are its leaves green, and so on. It may be quite a difficult test to perform, but assuming you can do it, you can find out if your watering system works.
- **Interaction testing**—You'd set up at the end of the irrigation hose a specific device that records when irrigation starts and ends, and how much water had flown through the device. At the end of the day you'd check with the device that it has been called exactly 3 times, with the correct quantity at each time, without worrying about checking the tree. In fact, you won't even need a tree to check that it works. You can also go further and modify the system clock on the irrigation unit, so that it thinks that the time to irrigate had arrived, and will do the irrigation at your will. That way you won't have to wait 12 hours to

find out if it works.

Sometimes State based testing is the best way to go because interaction testing is just too damn difficult to pull off – that's the case with crash test dummies: The car gets crashed into a standing target at a specific speed, and after the crash, both car and dummies' states are checked to see what are the ending outcomes. Pulling this sort of test as an *interaction test* in a lab can prove to be simply too complicated and a real world state based test is called for (although people are working on simulated crashes in computers, it's still not even close to testing the real thing)

So, back to the irrigation system. What is that thing that records how many times it was called? It's sort of a fake water hose, or a Stub, you could say. But it's a smarter breed of stub (Stub++!) . It's a stub that records the calls made to it.

Interaction Testing can be done with a *stub* that can record the calls made to it; That's partly what a Mock Object is.

MOCK OBJECT: DEFINITION

A Mock object replaces a real object under test conditions, and allows verifying the calls (interactions) against itself as part of a system or unit test. A mock object implements the same interface as the replaced object, but can be controlled, created and injected into the system under test by the unit test.

A Mock object may not sound like it's different in any way from a stub, but the differences are large enough to warrant a discussion on this topic, and special syntax in various frameworks, as you'll see in chapter 5. The next section deals exactly with what that difference is.

4.2 The difference between Mocks and Stubs

At first look, the difference between Mocks and Stubs may seem small or nonexistent. In fact, many people and articles use the term "Mocks" to describe Stubs, and vice versa. The distinction is subtle, but important. It's important because many of the *Mock Object frameworks* which we will deal with in the next chapters use both of these terms to describe different behaviors in the framework. If not for anything else, know the difference so that you know what they mean. Here's the point of this paragraph: Stubs can't fail tests. Mocks can do (or they'd be called Stubs).

Stubs simply replace an object so that we can test the object under test without problems. Figure 4.1 shows the interaction between the test and the class under test:

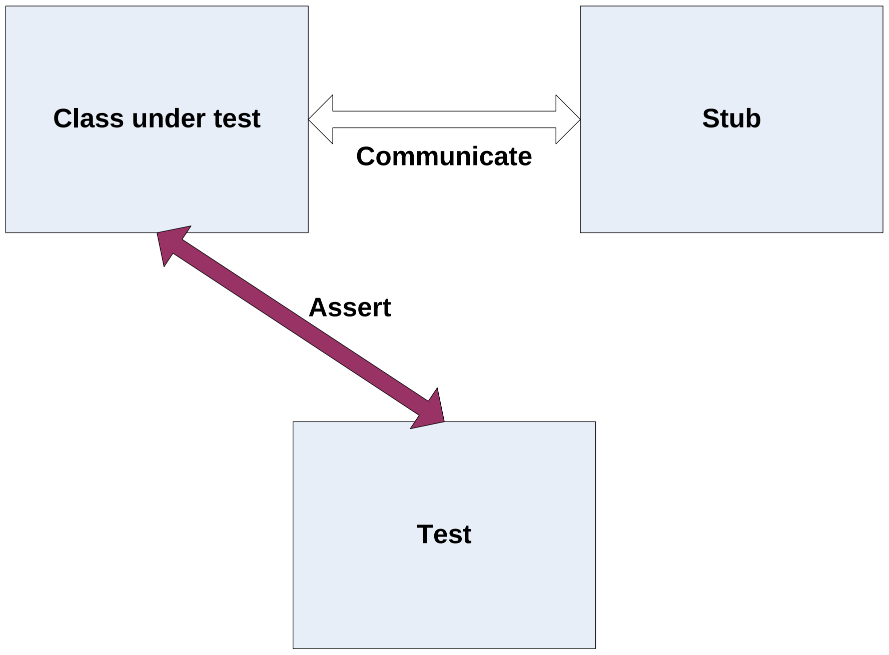


Figure 4.1: When using a stub, the Assert is performed on the class under test. The *Stub* simply aids in making sure the test runs smoothly and according to plan.

The easiest way to tell we are dealing with a Stub is to notice that the Stub can never ever *fail* the test. The asserts the test uses are always against the class under test.

When using *Mocks*, on the other hand, The test will use the Mock object to verify if the test failed or not. Again, when using a Mock, the Mock object is the object that we use to see if the test failed or not.

Figure 4.2 shows the interaction between the test and the mock object. Notice the Assert is performed on the Mock.

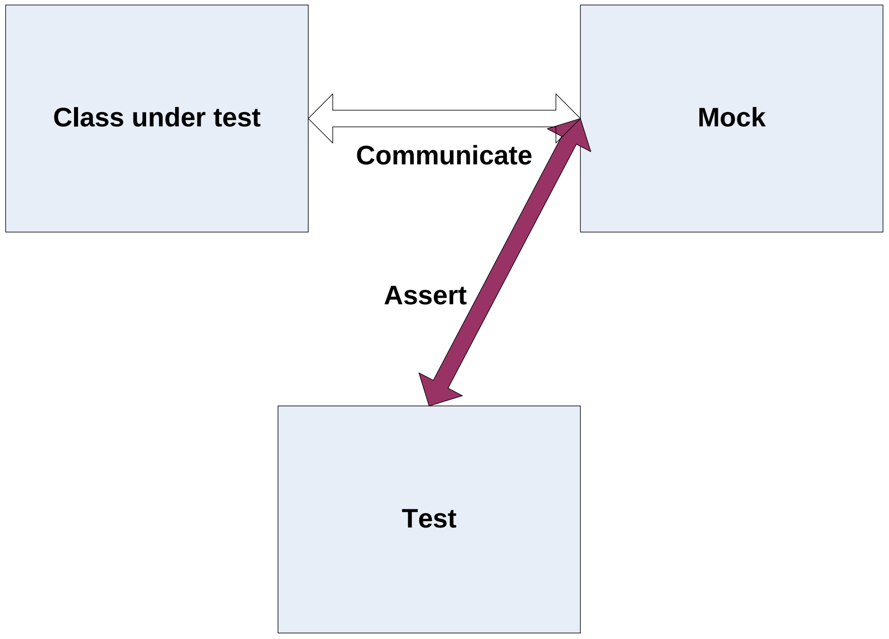


Figure 4.2 – The test uses the Mock object to verify the test passes. The class under test communicates with the mock objects, and all communication is recorded in the Mock, later verified by the test.

Let's take these ideas and practice them for real by building our own mock object in the next section.

4.3 A Simple Manual Mock Example

Creating and using a Mock object is semantically very much like using a Stub, except that a Mock will do a little more than a stub – it will save the history of communication so that it can later be verified.

Let's add a new requirement to our LogAnalyzer class. This time, it will have to interact with an external web service that will receive an error message whenever the `LogAnalyzer` encounters a filename whose length is not too long.

Unfortunately, the web service we'd like to test against is still not fully functional, and even if it were, it would have taken too long to use it as part of our tests. Because of that, we'll refactor our design and create a new interface that we can use to later create a Mock object of. The interface will have the methods we will need to call on our web service, and nothing else.

Figure 4.3 shows how the test will work with our `MockWebService`.

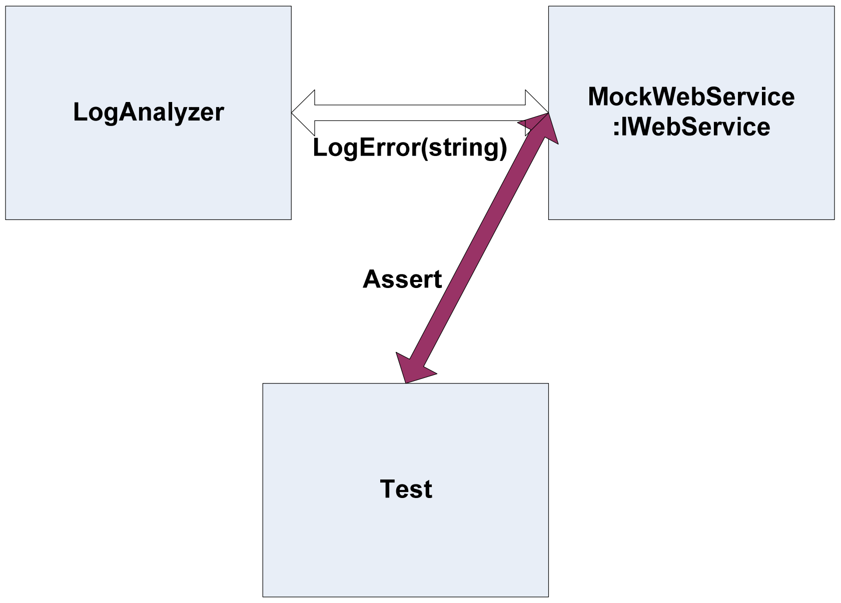


Figure 4.3 Our test will create a MockWebService to record messages that LogAnalyzer will send to. It will then assert against the MockWebService.

First off, let's extract a simple interface that we can use in our code under test, instead of talking directly to the web service.

```
public interface IWebService
 {
     void LogError(string message);
 }
```

This interface will serve us when we want to create stubs as well as mocks, without needing to have an external dependency on a project we have no control of.

Next, we'll create the Mock object itself. It may look like a stub, but it contains one little bit of extra code that makes it a mock object.

```
public class MockService:IWebService
 {
     public string LastError;

     public void LogError(string message)
     {
         LastError = message;
```

```
        }
    }
```

Our Mock implements an interface just like a stub, but saves some state for later, so that our test can then assert and verify that our mock was called correctly.

Listing 4.1 shows what the test might look like:

Listing 4.1: The log analyzer under test actually calls our mock object, which holds on to the string that is passed into the implemented method, and later asserted in the test.

```
 [Test]
public void Analyze_TooShortFileName_CallsWebService()
 {
     MockService mockService = new MockService();
     LogAnalyzer log = new LogAnalyzer(mockService);
     string tooShortFileName="abc.ext";
     log.Analyze(tooShortFileName);

          Assert.AreEqual("Filename too short:abc.ext",  |#1
                                    mockService.LastError);    |#1
}

public class LogAnalyzer
{
    private IWebService service;

    public LogAnalyzer(IWebService service)
    {
        this.service = service;
    }

    public void Analyze(string fileName)
    {
        if(fileName.Length<8)
        {
            service.LogError("Filename too short:"  |#2
                                           + fileName);  |#2
        }
    }
}
```

(annotation)<#1 the assert is done against the mock service, and not against the class under test>
(annotation)<#2 the method part we're going to test >

Notice how the assert is performed against the Mock object, and not against the LogAnalyzer class. That's because we are testing the *interaction* between LogAnalyzer and the web service. We still use the same dependency injection techniques from chapter 3, but this time the stub also makes or breaks the test. That's why it's a Mock object.

Also notice that we are not writing the tests directly inside the Mock Object code. There are several reasons for this:

We'd like to be able to re-use the Mock Object in other test cases, with other asserts on the message

If the assert is found inside the Mock Object, whoever is reading the test will have no idea what we are asserting. We will be hiding essential information from the test code, which hinders the readability and maintainability aspect of the test.

In many scenarios in your tests, you might find that you need to replace more than one object for the test to be able to work. We'll cover the concept of combining mocks and stubs next.

4.4 Using a Mock and a Stub together

Let's consider a more elaborate problem which we'll need to overcome. This time our `LogAnalyzer` not only needs to talk to a web service, but if the web service thrown an error, it has to log the error to a different external dependency – send it by email to the web service administrator. Figure 4.4 shows the flow visually.

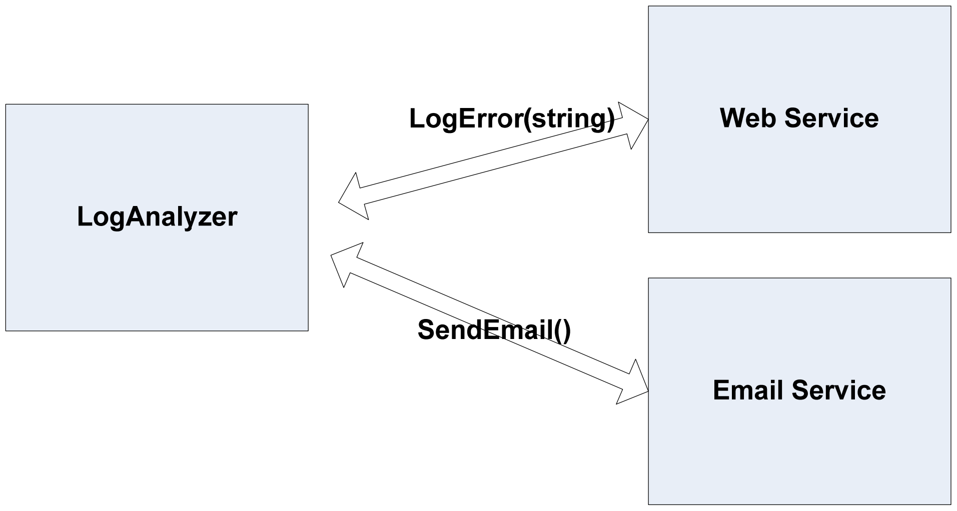


Figure 4.4 5: Log Analyzer has two external dependencies: Web Service and Email Service. We'll need to test LogAnalyzer's logic when calling them.

Here's the logic we'd like to test inside `LogAnalyzer`:

```
if(fileName.Length<8)
{
   try
   {
     service.LogError("Filename too short:" + fileName);
   }
   catch (Exception e)
   {
       email.SendEmail("a","subject",e.Message);
   }
}
```

Notice that there is logic here that only applies to *interacting* with external objects. No end result to the caller in sight. How do you test that indeed our analyzer calls the email service correctly when the web service throws an exception?

Here are the problems we are faced with:

- How do you replace the web service
- How can you simulate an exception from the web service so that you can test the call to the email service?

We can solve these questions by using a *stub* for the web service.

- How do you know that the email service was called correctly or at all?

We can solve this by using a *Mock Object* for the email service.

So, in our test we'll have two replacement object. One will be the email service Mock, which we'll use to verify that the correct parameters were sent to the email service, and a *stub* which we will use to simulate an exception thrown from the web service. It's a *stub* because we will not be using the Web service replacement to verify the test result, only to make sure the test runs correctly. The email service is a Mock because we will assert against it that it was actually called correctly. Figure 4.5 show this visually:

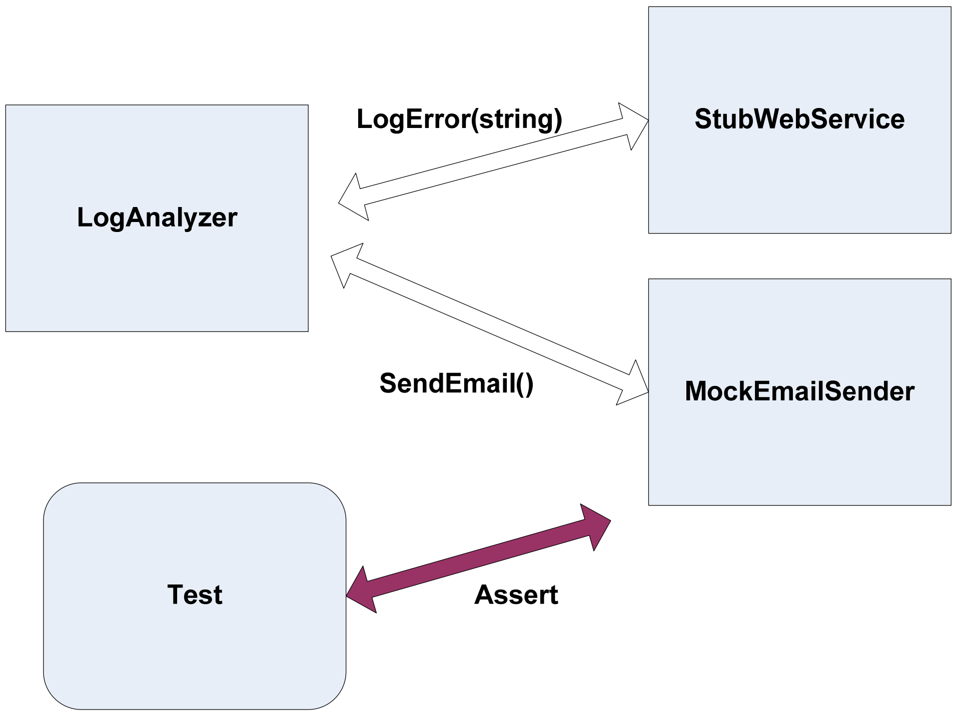


Figure 4.5 The Web service will be stubbed out to simulate an exception. Then the email sender will be mocked to see if it was called correctly. The whole test will be about how LogAnalyer interacts with other objects.

Here's the interface for the email service, followed by the Mock Email Service:

```
public interface IEmailService
  {
      void SendEmail(string to, string subject, string body);
  }
```

Listing 4.2 shows the code that implements the diagram shown in figure 4.5.

Listing 4.2: A Stub for the web service generates an exception, then the Mock for the email service is queried to verify that the interaction happened correctly.

```
public class LogAnalyzer2
    {
        private IWebService service;
        private IEmailService email;

        public IWebService Service
        {
            get { return service; }
            set { service = value; }
        }

        public IEmailService Email
        {
            get { return email; }
            set { email = value; }
```

```
     }

public void Analyze(string fileName)
     {
         if(fileName.Length<8)
         {
             try
             {
             service.LogError("Filename too short:" + fileName);
             }
             catch (Exception e)
             {
                 email.SendEmail("a","subject",e.Message);
             }
         }
     }
 }

 [TestFixture]
 public class LogAnalyzer2Tests
 {
     [Test]
     public void Analyze_WebServiceThrows_SendsEmail()
     {
         StubService stubService = new StubService();
         stubService.ToThrow=  new Exception("fake exception");

         MockEmailService mockEmail = new MockEmailService();

         LogAnalyzer2 log = new LogAnalyzer2();
       //we use setters instead of
       //constructor parameters for easier coding
         log.Service = stubService;                    |#1
         log.Email=mockEmail;                          |#1

         string tooShortFileName="abc.ext";
         log.Analyze(tooShortFileName);

         Assert.AreEqual("a",mockEmail.To);
         Assert.AreEqual("fake exception",mockEmail.Body);
         Assert.AreEqual("subject",mockEmail.Subject);
     }
 }

 public class StubService:IWebService                  |#2
 {
     public Exception ToThrow;
     public void LogError(string message)
     {
         if(ToThrow!=null)
         {
             throw ToThrow;
         }
     }
 }

 public class MockEmailService:IEmailService   |#3
 {
     public string To;
     public string Subject;
     public string Body;

     public void SendEmail(string to,
string subject,
string body)
        {
            To = to;
            Subject = subject;
            Body = body;
        }
    }
(annotation)< #1 Setters are used instead of constructors for mock and stub
injection >
(annotation)< #2 Handwritten stub  >
(annotation)< #3 Handwritten mock  >
```

Note that the public setters we've added will be used instead of using a constructor. We'll set the stubs and mock using them, which can make the code easier to read and maintain later.

There are some interesting questions arising from looking at this code:

- Why are we doing several asserts in a single test? How easy would it be to separate this test into 3 different tests with one assert each? Isn't the parameter checking one single test?
- It can be quite tedious to create these manual mocks and stubs for each test, or test class. How do you overcome that?
- Couldn't we have used the `MockService` from the earlier example as a stub?
- Is there a more elegant way to achieve the same results without going through all the hard labor?

I'll answer these questions at the end of this chapter, as they lead directly into the next chapter about Mock Object Frameworks. For now, rest assured that these all have good answers associated with them.

An important thing to consider is how many mocks and stubs can you use in a test? The next section deals with this question.

4.5.1 One Mock per test

In any given test in which you test only one thing (which is how I recommend people to write), there will never be more than one Mock Object. All other "replacement" objects will act as *stubs*.

This is important to realize because when you get to a more complicated test, you can always ask yourself, "Which one is my Mock Object?" Once you've identified it, you can leave the others as stubs and not worry about assertions against them.

Next we'll deal with a more complex scenario: using a stub to return a stub or a mock that will be used by the application.

4.6 Stubs that produce Mocks or other stubs

One of the most common scamming techniques online these days follows a very simple path: A fake email is sent to a massive number of recipients. The fake email is from a fake bank or online service claiming that the potential customer need to have some balance checked or change their details on the online site.

All the links in the email point to a *fake* site which looks exactly like the real thing, but its only purpose is to collect data from innocent customers of that store. This simple "chain of lies" is known as a "phishing" attack, and is more lucrative that you'd imagine. Most people respond instinctively to these emails and do what's expected. Consequently, it's one of the biggest threats to identity theft in the world.

How does this "chain of lies" matter to us? Sometimes we'd want to have a fake component return another fake component, thus, producing our own little chain of stubs in our tests.

In a sense, the fake email is nothing more than a stub. The stub is connected and leads to another fake location, used to gather details. The stub begets a Mock object which records data. This is a useful technique which we can use when we come to test our systems today.

In many systems, the design of the system allows for complex object chains to be created. It's not uncommon to find code like this hanging around:

```
IServiceFactory factory = GetServiceFactory();
```

```
IService service = factory.GetService();
```

Or something like this:

```
String connstring =
 GlobalUtil.Configuration.DBConfiguration.ConnectionString;
```

What if you wanted to replace the connection string with one of your own during a test?

You could setup the ‘`Configuration`’ property of the `GlobalUtil` object to be a stub object. You could then set the ‘`DBConfiguration`’ property on that object to be another stub object and so on and so forth.

It’s a powerful technique, but you need to ask yourself whether it may not be better to simply refactor your code to do something like this:

```
String connstring =GetConnectionString();
Protected virtual string GetConnectionString()
{
   Return GlobalUtil.Configuration.DBConfiguration.ConnectionString;
}
```

And then simply override the virtual method as described in the “Extract & Override” section in chapter 3. It can make the code easier to read and maintain, and does not require adding new interfaces just so you can insert two more stubs into the system.

4.6.1 Avoiding long call chains with wrapper APIs

Another good way to avoid call chains altogether is to create special *wrapper* classes aroung the API that simplify using it, and testing it. For learning more about this method I’ll refer you to Michael Feathers’ book “Working Effectively with Legacy Code”. The pattern in the book is called “Adapt Parameter”.

andwritten mocks and stubs have pros, but they also have their share of problems. We’ll dive into them in the next section.

4.7 The problems with handwritten Mocks/Stubs

As noted earlier, there are several issues that crop up when doing manual mocks and stubs.

- It’s going to be harder and harder to write stub s and mocks for classes and interfaces that have many methods, properties and events.
- How do you save state for multiple calls of a mock method? You need to start writing a lot of boilerplate code to save the data for each parameter.
- They take time to write.
- If you want to verify all parameters on a method call, you need to write multiple asserts. The problem is than if the first assert fails, the others will never run, as a failed assert throws an exception.
- It’s hard to reuse Mock and Stub code that we’ve written for other tests

These problems are inherent from the way we write the mocks: Manually. There are other ways to create mocks and stubs, as you’ll see in the next chapter.

4.8 Summary and best practices to take away from this chapter

This chapter walked you through the finer distinction between what a Stub and what a Mock object is. Basically, a Mock object is a stub that helps you to assert something in your test. A stub can never fail your test, and is strictly there to make happy noises, or to simulate various situations. This distinction between these two concepts is very important because many of the Mock Object Frameworks you’ll see in the next chapter, have the two notions engrained in them, and you’ll need to realize when to use which.

The notion of combining stubs and mocks in the same test is a powerful one, but you’ll have to be careful to always have no more than one mock in your test. The rest of the “fake” objects should be stubs that cannot break your test. Following this practice can lead to more maintainable and less brittle tests that break less often if internal code changes.

Stubs that produce other stubs or mocks can be a powerful way to inject your fake dependencies into code that uses other objects to get its data. It's a great technique for use with various factory classes and methods.

One of the most common problems people who write tests encounter is that they use mocks too much in their tests. 99% of the time, you should not verify calls to mock objects that are also used as stubs. In other words, if you see "verify" and "stub" on the same variable in the same test, you most likely are over specifying your test which will make it more brittle.

You can go to town and have stubs that return other stubs that return other stubs and so on, but at some point you'll wonder if it's all worth it. In that case, take a look at the techniques described in chapter 3 for injecting stubs into your design. Only one mock framework today allows stubbing a full call chain in one line of code (creating stubs that return stubs), and that's Typemock Isolator.

You can have multiple stubs in a test, since a class may have multiple dependencies. It's ok. Nothing to be ashamed of. Just make sure your test remains readable – structure your code nicely to make sure the reader of the test understands what's going on.

You may find that writing manual mocks and stubs sucks for large interfaces or for complicated interaction testing scenarios. It does, and your intuition is correct – there are better ways to do this – see the next chapter. But mark my words – many times you'll find that hand written mocks and stubs still beat frameworks for simplicity and readability. The "Art" lies in the "When" to use which. Hopefully that is answered in chapter 5.

Our next chapter deals with the notion of Mock Object Frameworks – which allow to automatically create, at *run time*, a stub or mock object, and use them with at least the same power of manual mocks/stubs, if not more.

5 Mock Object Frameworks

In the previous chapter I outlined some of the problems associated with writing Mocks and Stubs manually. In this chapter we'll see some very elegant solutions for these problems in the form of a *Mock Object framework* – a reusable library that can be used to create and configure Stubs and Mock Objects *at runtime* – these are usually referred to as *Dynamic Mocks* and *Dynamic Stubs*.

We'll start this chapter off by understanding what Mock Frameworks are and what they can do for us. We'll then take a closer look into one specific Mocking framework known as *RhinoMocks*. We'll see how we can use it to test various things, using it to create stubs, mocks and other interesting things.

Later, we'll review and contrast Rhino.Mocks to other Mocking frameworks currently available to .NET developers, and finish up by covering things to watch out for when using Mocking frameworks in your tests. It's going to be an interesting ride, but we have to start at the beginning: What are Mock frameworks?

5.1 What are Mock Frameworks?

A Mock Object Framework is a set of programmable APIs that allow creating Mock and Stub Objects in an easier fashion. Mock Frameworks save the developer from the need to write repetitive code to test or simulate object interactions.

This definition may sound a bit bland, but it needs to be very generic in order to describe the ideas behind the various Mock frameworks out there.

Mock frameworks exist for most languages that have a Unit Testing framework associated with them. C++ has `Mockpp` and several other frameworks, for example. Java has `JMock`, `EasyMock` and several other mocking frameworks, and so does .NET, including `NMock`, `NUnit.Mocks`, `EasyMock.NET` and `RhinoMocks`.

5.2 Why use Mock Frameworks?

Using Mock Frameworks instead of writing Mocks and stubs manually as seen in the previous chapters has several advantages that make developing more elaborate and complex tests easier, faster and less error-prone. To really understand the value of a Mock framework, however, there is no better way than to actually see a problem and a solution example.

One problem that might occur when using handwritten mocks and stubs is that of repetitive code.

Assume you have an interface a little more complicated than the ones shown so far:

```
public interface IComplicatedInterface
    {
        void Method1(string a, string b, bool c, int x, object o);
        void Method2(string b, bool c, int x, object o);
        void Method3(bool c, int x, object o);
    }
```

Creating a hand written stub or mock for it may start to be a little time consuming because we'd need to remember the parameters on a per-method basis, as listing 5.1 shows.

Listing 5.1 Implementing a complicated interfaces using handwritten stubs can become quite tedious.

```
class MytestableComplicatedInterface:IComplicatedInterface
    {
        public string meth1_a;
        public string meth1_b,meth2_b;
        public bool meth1_c,meth2_c,meth3_c;
        public int meth1_x,meth2_x,meth3_x;
        public int meth1_0,meth2_0,meth3_0;

        public void Method1(string a,
string b, bool c,
int x, object o)
        {
            meth1_a = a;
            meth1_b = b;
            meth1_c = c;
            meth1_x = x;
            meth1_0 = 0;
        }

        public void Method2(string b, bool c, int x, object o)
        {
            meth2_b = b;
            meth2_c = c;
            meth2_x = x;
            meth2_0 = 0;
        }

        public void Method3(bool c, int x, object o)
        {
            meth3_c = c;
            meth3_x = x;
            meth3_0 = 0;
        }
    }
```

Not only is this test time consuming and cumbersome to write , what happens if we'd like to test that a method was called X amount of times? Or have it return a specific value based on the parameters it receives? Or remember all the values for all the method calls on the same method (parameter history)? The code gets ugly fast.

Using a Mocking framework, the code to do this becomes trivial, readable and much shorter, as you'll find out when you create your first dynamic mock object.

5.3 Your first Dynamic Mock Object

Let's define what a *Dynamic Mock Object* is, and how it's different than a regular Mock.

DEFINITION

A Dynamic Mock object is any Stub or Mock that is created at runtime without needing to use a hand written implementation of an interface or sub class.

I'll use *RhinoMocks*, a Mocking framework that is open source and freely downloadable from www.ayende.com. It is simple, easy and quick to use, with little overhead in learning how to use the API. Once we get through the initial examples we'll explore other Mock frameworks and discuss the differences between them. You will also find a reference chapter about RhinoMocks as an appendix at the end of the book written by Oren Eini, the creator of RhinoMocks.

5.3.1 Introducing RhinoMocks into your tests

The only thing you'll need to do (assuming you have NUnit installed on your machine) is to add a reference to the `Rhino.Mocks.Dll` in the Add reference dialog of the test project, choose "Browse" and locate the download dll (you can get it from [http://www.ayende.com/projects/rhino-mocks/downloads.aspx])

LOOKING AT AN EARLIER HAND WRITTEN MOCK EXAMPLE

Let’s take an earlier example of a Mock Object we used to check whether a call to the log was performed correctly.

First, here’s the test class we used to check this with a manual mock in listing 5.2.

Listing 5.2 Asserting against a handwritten mock object

```
 [TestFixture]
    public class LogAnalyzerTests
    {
        [Test]
        public void Analyze_TooShortFileName_CallsWebService()
        {
           ManualMockService mockService = new ManualMockService ();
            LogAnalyzer log = new LogAnalyzer(mockService);
            string tooShortFileName="abc.ext";
            log.Analyze(tooShortFileName);
          Assert.AreEqual("Filename too short:abc.ext",
                 mockService.LastError);
        }
    }
    public class ManualMockService:IWebService
    {
        public string LastError;

        public void LogError(string message)
        {
            LastError = message;
        }
    }
```

I’ve highlighted with bold the parts in the code that are going to change when we’ll start using dynamically created mock objects.

5.3.2 Changing the code to use RhinoMocks

RhinoMocks introduces a new class called `MockRepository` that has a special method called `CreateMock()`: it will take in its constructor or as a generic parameter the type of an interface or class that you’d like to simulate, and will *dynamically* create an object which adheres to that interface at run time, without needing to implement that new object in real code.

We’ll now go over creating a mock step by step, eventually replacing the earlier test with one that uses dynamic mocks.

Listing 5.3 shows a simple piece of code to create a simulated object based on an interface using Rhinomocks:

Listing 5.3 Creating a dymamic mock object using RhinoMocks requires no handwritten mocks to be created.

```
 [Test]
Public void Analyze_TooShortFileName_ErrorLoggedToService()
{
MockRepository mocks = new MockRepository();
IWebService simulatedService =
mocks.CreateMock<IWebService>();

 using(mocks.Record())
 {
     simulatedService.LogError("file name was too short ");
 }

    LogAnalyzer log = new LogAnalyzer(simulatedService);
      string tooShortFileName="abc.ext";
      log.Analyze(tooShortFileName);

mocks.VerifyAll();
}
```

The first two lines are the ones who rid us of the need to use a hand written stub or mock, because they generate one dynamically:

```
MockRepository mocks = new MockRepository();
IWebService simulatedService = mocks.CreateMock<IWebService>();
```

The `simulatedService` object instance is a dynamically generated object which implements the interface `IWebService`, but does not have any implementation inside any of the implemented `IWebService` methods. Before we continue, let's talk about expectations.

DEFINITION: MOCK AND STUB EXPECTATIONS

An expectation on a mock object is an ad-hoc rule that is set on the object. The rule will usually tell the object that a specific method call on that object is *expected* to happen later. It may also define how many times it is expected to be called, whether or not to thrown an exception when that call arrives, or perhaps that is never expected. Expectations are usually set on mocks objects during the recording phase of the mock object, and are verified during the end of the test where the mock object lives.

Expectations that are set on *stub* objects can tell those stubs what values to return based on expected method calls, whether to throw exceptions or other things. Usually you won't need to *verify* expectations set on *stubs*. Only on *Mocks.*

RECORDING EXPECTATIONS

Next we are setting the simulated service object into a "record" state. This is a special state where each method call on the simulated object is actually recorded as an *expectation* that something should happen to our simulated object.

```
using(mockEngine.Record())
 {
     simulatedService.LogError("file name was too short");
 }
```

These *expectations* will later be verified by calling `mockEngine.VerifyAll()`. In this case we only have one expectation—that `LogError()` will be called with the exact input of `"file name was too short"`.

Next, we actually invoke the object under test – our `LogAnalyzer` by injecting it with our mock object, and sending in a short file name that should make it invoke the logger internally.

```
LogAnalyzer log = new LogAnalyzer(simulatedService);
  string tooShortFileName="abc.ext";
  log.Analyze(tooShortFileName);
```

The last step in this test is to do some sort of Assert. In this case, we will need to find a way to Assert that all the expectations have been met by our mock object (that `LogError` was indeed called with the correct string). We can do this by using `MockRepository.VerifyAll()`.

```
mockEngine.VerifyAll();
```

The `VerifyAll()` method will go through each of the *expectations* that we've set on our mock object and make sure they are true (called correctly).

Here's a question: What happens if the implementation of `log.Analyze()` contains an unexpected call to the service?

```
public void Analyze(string fileName)
{
    if(fileName.Length<8)
    {
        service.LogError("bad string);
    }
}
```

The test will fail, but it won't fail when calling the VerifyAll() method. It will fail before that, during the test run when the call to "LogError" is executed. The test will never get to the `VerifyAll()` line because an exception will be thrown before that. To understand why, we'll review the idea of strict and non strict Mocks In the next section.

5.4 Strict vs. Non Strict Mock Objects

Let's review what strict and non strict mocks mean, and why non strict mocks are better for most tests.

5.4.1 Strict Mocks

A strict Mock Object will only allow calling methods on it that were explicitly set via Expectations. Any call that is made which either differs by the parameter values defined, or by the method name, will usually be handled by throwing an exception while the method is being called. That means the test fails on the first *unexpected* method call to a strict mock object. (I used the term "usually" because whether or not the mock throws an exception depends on the implementation of the Mock framework in question. Some frameworks allow you to define whether to hold off all exceptions until calling `verify()` in the end of the test. A strict Mock can fail in two ways: while an unexpected method is being called on it, or when expected methods were not called on it (which is determined by calling *VerifyAll*()).

It's important to note that with RhinoMocks, unexpected method call exceptions will always be thrown, even if your test contains a global try-catch clause.

5.4.2 Non Strict Mocks

This is the preferable choice 99% of the time, as *non strict* mocks make for less brittle tests.

A *Non Strict* Mock will allow any call to be made to it even if it was not expected. As long as the call does not require a return value, it will, "make happy noises" and only do what's necessary for everything in the test to work out.. If a method which needs to return a value is called and we did not setup a return value as part of setting up that mock object, a RhinoMocks *stub* object can return the default value for that method's return type (`0` or `null` usually). Other frameworks may take different approaches and may throw an exception when the method is not configured to return anything.

A *non strict* Mock can *only* fail a test if an expected method was *not called*. You have to call the `VerifyAll()` method to find out if such a call is missing from the interaction or the test will pass.

The example at the end of the previous section uses a *strict* mock approach which is why running the test fails mid test instead of when calling *VerifyAll()*. You can create *Non Strict* Mocks using RhinoMocks by calling `repository.DynamicMock<type>()` instead of calling `repository.CreateMock<Type>()`.

Listing 5.4 shows how the test would look if we used a non strict mock object with RhinoMocks:

Listing 5.4 Creating a non-strict Mock means calling a different creation method. The 'DyanmicMock' in the method name can be misleading. It should really mean "non strict mock".

```
 [Test]
Public void Analyze_TooShortFileName_ErrorLoggedToService()
{
MockRepository mocks = new MockRepository();
IWebService simulatedService = mocks.DynamicMock<IWebService>();

 using(mocks.Record())
 {
     simulatedService.LogError("file name was too short ");
 }

    LogAnalyzer log = new LogAnalyzer(simulatedService);
      string tooShortFileName="abc.ext";
      log.Analyze(tooShortFileName);

mocks.VerifyAll();
}
```

Mocks created with A Mock framework can also be used as stubs. We can tell them to return fake values and other interesting effects. The next section shows how to do that.

5.5 Returning values from Mock Objects

We can instruct the Mock Object to return a value based on a given method call by using a special class called `LastCall`. Listing 5.5 shows how you would return a value from a mock object when the interface method has a non void return value:

Listing 5.5 Returning a value from a mock object by using the LastCall class.

```
            [Test]
            public void ReturnResultsFromMock()
            {
               MockRepository mocks = new MockRepository();
                IGetRestuls resultGetter = mocks.DynamicMock<IGetRestuls>();
                using(mocks.Record())
                {
                    resultGetter.GetSomeNumber("a");
                    LastCall.Return(1);                                 |#1

                    resultGetter.GetSomeNumber("a");
                    LastCall.Return(2);

                    resultGetter.GetSomeNumber("b");
                    LastCall.Return(3);
                }

                int result = resultGetter.GetSomeNumber("b");
                Assert.AreEqual(3, result);

                int result2 = resultGetter.GetSomeNumber("a");
                Assert.AreEqual(1, result2);

                int result3 = resultGetter.GetSomeNumber("a");
                Assert.AreEqual(2, result3);
    }
```

(annotation) <#1 force a value to be returned from this method call later on>

During the recording stage, we use the `LastCall` class to set the return value for that method call, when that specific input ('a' in this case) is sent to the method. To illustrate this, there are three expectations set on the mock object, and after each one we are setting the result to be returned from these method calls. Our Mock object will be smart enough to return the correct return value based on the input that was set in the expectation. If the same input is set with different return values, they will be returned in the order the code has added them.

You'll notice that after the Recording stage, I'm actually calling `GetSomeNumber` with the "b" input, but the test will still pass, which means the order of the calls does not matter (if I wanted it to matter there is the idea of ordered mocks which will be discussed in the related RhinoMocks chapter in the books appendix).

However, If I change the last two asserts in the test (which both input "a") the test will fail because the order they were recorded in matters when the input is the same for the expectations.

You can use `LastCall` to also set the method call to throw a specific exception
(LastCall.Throw(excepction object))

or to even execute your own delegate
LastCall(Do(yourdelegatehere))

But this functionality is usually more relevant to *Stubs* than Mock Objects, which is exactly what our next section is about.

5.6 Creating Smart Stubs with a Mock Framework

As I said before, a *Stub* can also be created using Mocking frameworks. Semantically speaking, a *stub* is a simulated object, just like a mock object, that will *support* our test so that we can test *other things*. In short, a Stub will always make "happy noises" when being called, and can never be used to see if the test passed or not. Calling `VerifyAll()` will not verify anything against stub objects. Only against mocks. Most Mocking frameworks contain the semantic notion of a *stub* and RhinoMocks is no exception. Listing 5.6 shows how you create a stub object.

Listing 5.6 Creating a stub looks remarkably similar to creating a mock object. Only the creation method call changes.

```
 [Test]
        public void ReturnResultsFromStub()
        {
            MockRepository mocks = new MockRepository();
            IGetRestuls resultGetter = mocks.Stub<IGetRestuls>();
            using(repository.Record())
            {
                resultGetter.GetSomeNumber("a");
                LastCall.Return(1);

            }
            int result = resultGetter.GetSomeNumber("a");
            Assert.AreEqual(1, result);
        }
```

The Syntax looks almost the same as using mock objects But what would happen when running a test that does *not* call an expected method on the stub, but still verifies expectations as shown in listing 5.7.

Listing 5.7 verifying excpectations on a stub object cannot fail a test. That's what Mock objects are for.

```
public void StubNeverFailsTheTest()
        {
            MockRepository mocks = new MockRepository();
            IGetRestuls resultGetter = mocks.Stub<IGetRestuls>();
            using(mocks.Record())
            {
                resultGetter.GetSomeNumber("a"); |#1
                LastCall.Return(1);

            }
            resultGetter.GetSomeNumber("b");
            mocks.VerifyAll(); |#2
        }
```

(annotation) <#1 is this really an expectation?>
(annotation) <#2 will this fail the test?>

The test should still pass, because the Stub, by definition, is *incapable* of breaking a test, and that's also how it behaves in Rhino.Mocks. Any expectations set on it are purely so we can determine the return value or the exceptions to throw from them.

RhinoMocks contains a feature that is not supported by most frameworks (Except Typemock Isolator) which is very handy. for simple properties on stub objects. Get-set properties are automatically implemented and can be set and used as if this was a real object, without needing to setup a return value. You can still set up a return value for a property, but you don't need to do it for each and every property on a stub object As shown in the next piece of code:

```
            ISomeInterfaceWithProperties stub =
                mocks.Stub<ISomeInterfaceWithProperties>();

            stub.Name = "Itamar";
            Assert.AreEqual("Itamar",stub.Name);
```

Let's see how we can simulate an exception using expectations. Listing 5.8 shows an example of how you would simulate an `OutOfMemoryException` in a test from a stub:

Listing 5.8 Faking an exception with a dynamically created stub is very easy and very readable when using the LastCall class..

```
public void StubSimulatingException()
        {
            MockRepository mocks = new MockRepository();
            IGetRestuls resultGetter = mocks.Stub<IGetRestuls>();
            using(mocks.Record())
            {
                resultGetter.GetSomeNumber("A");
                LastCall.Throw(
new OutOfMemoryException("The system is out of memory!")
);
            }
            resultGetter.GetSomeNumber("A");
        }
```

This test will fail due to a nasty out of memory exception. That’s how easy it is.

Let’s use this ability to simulate errors by testing scenario with a little more complexity in the next section.

5.6.1 Combining Dynamic Stubs and Mocks

We’ll use the example from chapter 4, where we talked about `LogAnalyzer` using a `MailSender` class and a `WebService` class, as shown in Figure 5.1:

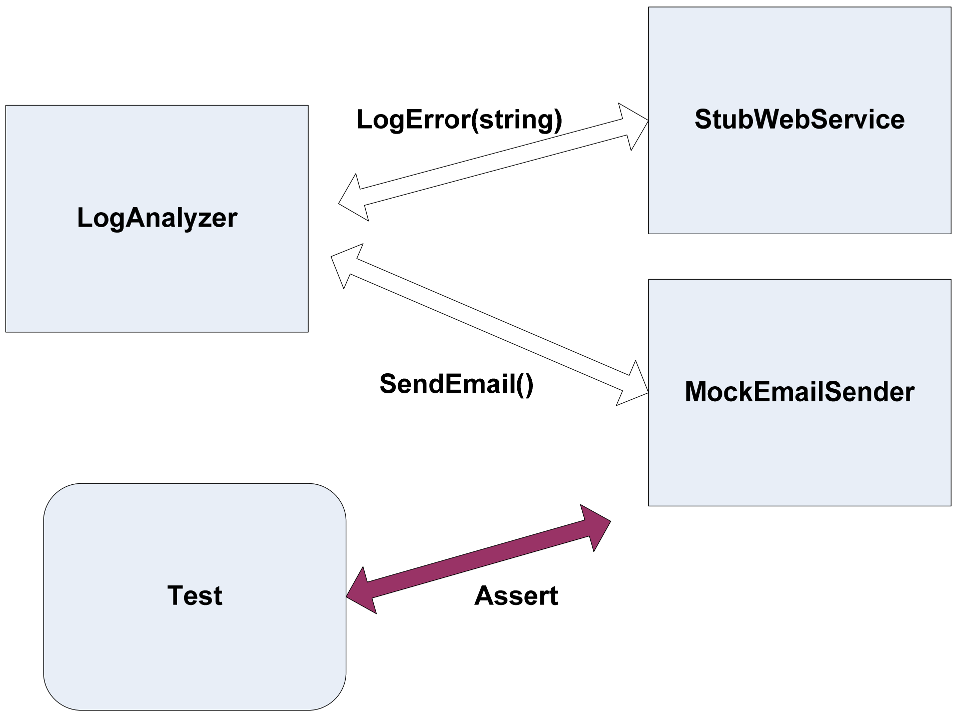


Figure 5.1 The Web service will be stubbed out to simulate an exception. Then the email sender will be mocked to see if it was called correctly. The whole test will be about how LogAnalyer interacts with other objects.

We want to test that if the service throws an exception, the log analyzer will use mail sender to send an email to an administrator. Here’s what the logic should look like with all the tests passing (listing5.9):

Listing 5.9 The method we'd like to test and a test that uses handwritten mocks and stubs to test it. The code starts to look quite cumbersome.

```
public void Analyze(string fileName)
        {
            if(fileName.Length<8)
            {
                try
                {
                service.LogError("Filename too short:" + fileName);
                }
                catch (Exception e)
                {
                    email.SendEmail("a","subject",e.Message);
                }
            }
        }
    }

          [Test]
        public void Analyze_WebServiceThrows_SendsEmail()
        {
            StubService stubService = new StubService();
            stubService.ToThrow=  new Exception("fake exception");

            MockEmailService mockEmail = new MockEmailService();

            LogAnalyzer2 log = new LogAnalyzer2();
          //we use setters instead of
          //constructor parameters for easier coding
            log.Service = stubService;
            log.Email=mockEmail;

            string tooShortFileName="abc.ext";
            log.Analyze(tooShortFileName);

            Assert.AreEqual("a",mockEmail.To);
            Assert.AreEqual("fake exception",mockEmail.Body);
            Assert.AreEqual("subject",mockEmail.Subject);
        }
    }
    public class StubService:IWebService
    {
          ...
    }

    public class MockEmailService:IEmailService
    {
          ...
    }
```

And listing 5.10 shows what the test could look like if we used RhinoMocks:

Listing 5.10 Converting the previous test into one that uses dynamic mocks and stubs makes it more readable and maintainable, and takes less time to write.

```
 [Test]
        public void Analyze_WebServiceThrows_SendsEmail()
        {
            MockRepository mocks = new MockRepository();
            IWebService stubService =
                              mocks.Stub<IWebService>();
            IEmailService mockEmail =
                              mocks.CreateMock<IEmailService>();

            using(mocks.Record())
            {
                stubService.LogError("whatever");
                LastCall.Constraints(Is.Anything());
                LastCall.Throw(new Exception("fake exception"));

                mockEmail.SendEmail("a","subject","fake exception");
            }

            LogAnalyzer2 log = new LogAnalyzer2();
            log.Service = stubService;
            log.Email = mockEmail;

            string tooShortFileName = "abc.ext";
            log.Analyze(tooShortFileName);

            mocks.VerifyAll();
        }
```

The nice thing about this test is that it requires no handwritten mocks, and it is still very readable to the developer.

You might notice a line in there that you haven't come across yet:

```
LastCall.Constraints(Is.Anything());
```

That's a parameter constraint that makes sure that no matter what we send the stub as parameter, we will still throw the exception. Parameter Constraints are, fortunately, what the next section is about.

5.7 Parameter constraints for Mocks and Stubs

Testing the value of parameters being passed in to our mock objects is a large part of what Mock frameworks enable to do very easily. In the coming sections we'll go through the various ways you can check parameters as such as strings, properties and full object models using very little code. After that we'll also go over testing event related logic, and go over the differences between RhinoMocks and other mock frameworks in the .NET space. The chapter ends with some common traps and pitfalls when using Mock objects, and what you can do about them.

Consider the following scenario: We'd like our `LogAnalyzer` to send a message of this nature:
`“[Some GUID] Error message”`

For example:
`“33DFCC9D-D6C5-45ea-A520-A6018C88E490 Out of memory”`

We don't really care about the `Guid` at the beginning, but we care what the error message is. In fact we don't really have control of the `Guid` at the start of the string (we could gain control by creating some sort of a “`IGuidCreator`” interface and stubbing it out in the test, but that might prove a little too much work for what we need.

Parameter Constraints allow us to tell our Mocks or stubs to have specific demands or rules about what to expect from each specific parameter to a method call. Mock Mocking frameworks have this ability, and each has its own syntax for doing it. With RhinoMocks, Constraints are reachable by using the “`Is`” class which contains various constraint-checkers as methods or properties.

The simplest way to use constraints, as shown by listing 5.11, is by using the `LastCall` class in conjunction with tone of the Constraints classes. In our case it would be the Contains class which takes as a constructor the inner string to search for:

Listing 5.11 Using a String constraint in a test.

```
 [Test]
        public void SimpleStringConstraints()
        {
            MockRepository mocks = new MockRepository();
            IWebService mockService = mocks.CreateMock<IWebService>();
            using (mocks.Record())
            {
                mockService.LogError("ignored string");
                LastCall.Constraints(new Contains("abc"));
            }

            mockService.LogError(Guid.NewGuid() + " abc");
            mocks.VerifyAll();
        }
```

Using the `LastCall.Constraints()` method we can send in a constraint object (which has to inherit from `AbstractConstraint` defined as part of RhinoMocks) for each parameter the method expects. There are many types of constraints. For string related constraints we have `Contains`, `EndsWith`, `Like` and `StartsWith`. All classes which take a string at the constructor. To ease the use of these Constraints, a helper class called Text exists in RhinoMocks which has static methods that return these constraint objects.

Here's the same row using the Text class:

`LastCall.Constraints(`**`Text.Contains`**`("abc"));`

There are four major "helper" classes for constraints in Rhino.Mocks , listed in table 5.1.

Table 5.1 The four types of constraints in RhinoMocks

Helper Class	**Description**	**Methods**

`Text`	String related constraints	`Contains(string)`
		`EndsWith(string)`
		`StartsWith(string)`
		`Like(string)`

`List`	Collection related constraints	`Count(constraint)`
		`Element(int, constraint)`
		`Equal(ICollection)`
		`IsIn(object)`
		`OneOf(ICollection)`

`Is`	Check direct value of parameter passed in	`Anything()`
		`Equal(object)`
		`GreaterThan(IComparable)`
		`LessThan(IComparable)`
		`Matching<T>(Predicate<T>)`
		`NotNull()`
		`Same(object)`
		`TypeOf(Type)`

`Property`	Checks the value of a specific Property On an object that is passed in as a parameter	`IsNull()`
		`Value(Type, PropertyName,object)`
		`ValueConstraint(Type, PropertyName, constraint)`

Some of the more interesting constraints are `Property` constraints, `And\Or` Constraints and `Callback` constraints. Let's review them one by one.

5.7.1 Checking parameter object properties with constraints

Assume that the `IWebService` interface has a method that expects to take in a `TraceMessage` object, which contains specific rules about it's properties. We could easily check the values of the passed in trace Message object properties using the Property related constraints. This is shown in listing 5.12.

Listing 5.12 Using the Property constraints by using the Property static class.

```
[Test]
        public void ConstraintsAgainstObjectPropeties()
        {
            MockRepository mocks = new MockRepository();
            IWebService mockservice = mocks.CreateMock<IWebService>();
            using (mocks.Record())
            {
                mockservice.LogError(new TraceMessage("",0,""));
                LastCall.Constraints(
                    Property.Value("Message", "expected message") &&
                    Property.Value("Severity", 100)               &&
                    Property.Value("Source", "Some Source"));
            }
            mockservice.LogError(new TraceMessage("",1,"Some Source"));
            mocks.VerifyAll();
        }
```

COMBINING CONSTRAINTS TOGETHER WITH AND AND OR

Notice the usage of the && operators here. They essentially are overloaded to use a special `And` constraint, which makes sure that all of these constraints have to be true at the same time for the test to pass. You could also use the || overload to set a special `Or` constraint which only makes sure *one* of the constraints turns out to be true.

The `And` and `Or` Constraints both take two `AbstractConstraint` objects in their constructor. The above example actually combines together two `And` Constraints and could also have been written like listing 5.13:

Listing 5.13 Combining Constraints with And and Or Constraints

```
And combined1 = new And(
Property.Value("Message", "expected message" ),
                  Property.Value("Severity", 100));
And combined2 = new And(
combined1,
Property.Value("Source", "Some Source"))
LastCall.Constraints(combined2);
```

COMPARING FULL OBJECTS AGAINST EACH OTHER

Of course, if we are just going to test things in the most simplistic way, we could just compare two objects. We send the "expected" object with all the expected properties set as part of the recording process (no need for constraints) and then just call `verify()`, as shown in listing 5.14:

Listing 5.14 Comparing a full object has some drawbacks, but it makes for very readable tests.

```
[Test]
        public void TestingObjectPropertiesWithObjects()
        {
            MockRepository mocks = new MockRepository();
            IWebService mockservice = mocks.CreateMock<IWebService>();
            using (mocks.Record())
            {
                mockservice.LogError(
new TraceMessage("Some Message",100,"Some Source"));
            }
```

```
                mockservice.LogError(new TraceMessage("",1,"Some Source"));
                mocks.VerifyAll(); //this should fail the test
            }
```

This only works for cases where:

- It's easy to create the object with the expected properties
- You'd like to test all the properties of the object in question
- You know the exact values of each constraint.

The `Equals()` method is implemented correctly on the object being compared. It's usually bad practice to simply rely on the out of the box implementation of `object.Equals()` for less trivial object structures.

5.7.2 Executing Callbacks for parameter verification

The `Is.Matching<T>(Predicate<T>)` constraint is a very powerful feature which allows the developer of the test to test whatever they want against the passed in parameter and return true or false based on very complex rules.

For example, assume that the `IWebService` interface has a method that expects to take in a TraceMessage object, which in turn has a property that holds an object that you'd like to check. If we had a `ComplexTraceMessage` class with an `InnerMessage` property, and a complex verification on it, it might look like listing 5.15:

Listing 5.15 using an anonymous delegate to verify a parameter, and sending that delegate as a callback to RhinoMocks.

```
     LastCall.Constraints(
    Is.Matching<ComplexTraceMessage>(
               delegate(ComplexTraceMessage msg)
                       {
                       if (msg.InnerMessage.Severity < 50
                               && msg.InnerMessage.Message.Contains("a"))
                                   {
                                     return false;
                                   }
                                     return true;
                       }));
```

We are actually creating a delegate that holds the logic to verify the complex parameter structure. Instead of using a delegate we could create a method with the same signature that does the same thing, as shown in listing 5.16:

Listing 5.16 Using a regular method instead of an anonymous delegate makes for less redable tests for some people. Use at your own discretion. [Test]

```
    public void ComplexConstraintsWithCallbacks()
            {
                ...
                using (mocks.Record())
                {
                    mockservice.LogError(new TraceMessage("", 0, ""));
                    LastCall.Constraints(
                        Is.Matching<ComplexTraceMessage>(verifyComplexMessage));
                }
    ...
            }
            private bool verifyComplexMessage(ComplexTraceMessage msg)
            {
                if (msg.InnerMessage.Severity < 50
                    && msg.InnerMessage.Message.Contains("a"))
                {
                    return false;
                }
                return true;
            }
```

RhinoMocks has a simpler syntax if you're only testing a method that accepts a single parameter. Instead of using this:

```
                 LastCall.Constraints(
                        Is.Matching<ComplexTraceMessage>(verifyComplexMessage));
```

You can write this:

```
                 LastCall.Callback(verifyComplexMessage);
```

ARRANGE-ACT-ASSERT SYNTAX FOR RHINO MOCKS

Lately, Rhino Mocks has come out with an API that adds support for AAA (Arrange-act-assert) style tests, ridding you of the need to use the record-replay model which many people find a bit confusing. Rhino Mocks still supports the record replay syntax, which I feel is easier to learn the basics with, so I have not used the new API in this book. To learn more about the new API, go to http://ayende.com/Blog/archive/2008/05/16/Rhino-Mocks--Arrange-Act-Assert-Syntax.aspx .

Another common task you might want to check for in your tests is whether objects raise events properly, or whether other objects have registered for an event propely. This is covered in the next section.

5.8 Testing for Event Related Activities

Testing event related actions has always been one of the gaping holes in Mock frameworks and required manual work around to test various things such as whether some object indeed registered to get an event from another object, or whether an object had triggered an event when it should have. RhinoMocks has facilities to help ease the pain in these areas as well.

The first scenario we'd like to tackle deals with checking if an object actually registered to receive some event from another object.

5.8.1 Testing that an event had been subscribed to

Assume a Presenter class that needs to register to the Load event of a view class which it receives. The code for presenter might look like listing 5.17:

Listing 5.17 testing that an event was registered to require both an event source (that raises the event) and a subscriber which we'd like to test if it actually subscribed to that event.public class Presenter

```
{
    IView view;
    public Presenter(IView view)
    {
        this.view = view;
        this.view.Load += new EventHandler(view_Load);
    }

    void view_Load(object sender, EventArgs e)
    {
        throw new Exception("Not implemented.");
    }
}
```

```
//And here’s the test that makes sure the event was indeed registered to:
[Test]
        public void VerifyAttachesToViewEvents()
        {
            MockRepository mocks = new MockRepository();
            IView viewMock = (IView)mocks.CreateMock(typeof(IView));
            using (mocks.Record())
            {
                viewMock.Load += null;
                LastCall.IgnoreArguments();
            }
            new Presenter(viewMock);
            mocks.VerifyAll();
        }
```

During the recording stage, we simply overload the `Load` event. Then we make sure to ignore the arguments in the call, and just make sure the call indeed happened.

Some people find this to be helpful, but personally I’ve found that just being able to see if someone has indeed registered to an event, does not necessarily mean they did something meaningful with it. It’s just mechanics. It’s not a real functional requirement that is tested here.

To test this as part of a functional requirement, we could say that upon getting the Load event, the presenter will do something (write to a log, for example). Which means we need to find a way to trigger the event from within our (in this case) *stub object* and see if the presenter did something functional with it.

What if we actually want to trigger the event as part of our test and see what happens to the subscriber? That’s what the next section is about.

5.8.2 Triggering events from Mocks and Stubs

Listing 5.18 shows a test that makes sure that `Presenter` writes to a log upon getting an event from our *stub*. It also used a class called `EventRaiser` which is used to trigger the event from the interface:

Listing 5.18 Triggering an event via the EventRaiser class in Rhino.Mocks

```
 [Test]
        public void TriggerAndVerifyRespondingToEvents()
        {
            MockRepository mocks = new MockRepository();
            IView viewStub = mocks.Stub<IView>();
            IWebService serviceMock = mocks.CreateMock<IWebService>();
            using (mocks.Record())
            {
                serviceMock.LogInfo("view loaded");
            }
            new Presenter(viewStub,serviceMock);
            IEventRaiser eventer = EventRaiser.Create(viewStub, "Load");
            eventer.Raise(null,EventArgs.Empty);

            mocks.VerifyAll();
        }
```

Another option of getting an EventRaiser object is by using the recording mechanism:

```
IEventRaiser eventer;
         using (mocks.Record())
           {
viewStub.Load += null;
eventer = LastCall.GetEventRaiser();
           }
```

Notice we are using a stub to trigger the event, and a mock to check that the service was written to. The `EventRaiser` takes a stub or a mock, and the name of the event to raise from that stub or mock. The `Raise()` method of `EventRaiser` takes a `params object[]` array that requires you to send the same amount of parameters as the event signature requires.

The verification happens on the mock service, whether it got the message or not.

Let's take a look at the opposite end of the testing scenario. Instead of testing the subscriber, we'd like to make sure that the event source avtually triggers the event at the right time. The next section shows how we can achieve that.

5.8.3 Testing that an event was triggered

To test that an event was triggered, you obviously need someone to register to that event and tell you if it was fired or not. We'll have to, again, have two objects in our tests. One will be the event source we're trying to test, and the other will register for the event and tell us if it was raised. The next section shows a very simple way of accomplishing that using a hand written anonymous method.

5.8.3.1 Testing event firing manually

A simple way to register for the event is by registering to it manually inside the test method using an anonymous delegate. Here's a simple example shown in listing 5.19:

Listing 5.19 Using an anoynymous delegate to register to an event from the class under test.

```
 [Test]
        public void EventFiringManual()
        {
            bool loadFired = false;

            SomeView view = new SomeView();
            view.Load+=delegate
                               {
                                    loadFired = true;
                               };
            Assert.IsTrue(loadFired);
        }
```

The delegate simply records that the event was fired or not. It could also have recorded what the values were and they could later be asserted as well. That code could also become quite cumbersome.

The next section shows a less cumbersome way to test the event triggering and the values that are passed in.

5.8.3.2 Using EventsVerifier for event testing

Another approach is to use a class called `EventsVerifier` which can be downloaded from `[http://weblogs.asp.net/rosherove/archive/2005/06/13/EventsVerifier.aspx]`. `EventsVerifier` will dynamically register against the required delegate and verify the values that are passed in by the fired event. Listing 5.20 shows an example.

Listing 5.20 Using the EventsVerifier class makes it very simple to test for event values without lots of cumbersome handwritten code.

```
[Test]
        public void EventFiringWithEventsVerifier()
        {
            EventsVerifier verifier = new EventsVerifier();
            SomeView view = new SomeView();
            verifier.Expect(view, "Load",null,EventArgs.Empty);

view.TriggerLoad(null, EventArgs.Empty);

            verifier.Verify();
        }
```

This test assumes we have a class that implements `IView`, and has a method that simply triggers the event. The verifier takes the object to test against, and the name of the event, and any parameter values that are passed in as part of the event signature.

Rhino.Mocks currently doesn't have a decent enough API to test event verification like `EventsVerifier`. You should consider using EventsVerifier alongside using RhinoMocks in your tests.

It's time to compare RhinoMocks to other Mocking Frameworks in the .NET World. Having this information should be critical in making a choice which framework to use in your tests. It's usually a good idea to pic one and stick with it as much as possible, for the sake of readability and to lower the learning curve of team members. Our next section has the information necessary to help make that choice.

5.9 RhinoMocks vs. Other Mocking Frameworks

RhinoMocks is certainly not the only mocking framework around. However, at a very unofficial poll that I held on my blog I asked my blog readers "Which mock object framework do you use?" I was surprised to see that even though at the time Rhino Mocks was a relative new comer to the mocking field, close to 45% of the 300 answers were using Rhino.Mocks over other frameworks that exist out there such as NMock, NMock2, NUnit.Mocks and TypeMock Isolator.

What follows is a short review of the current mocking frameworks in .NET, and why a tool such as RhinoMocks has managed to take over the lead in this field despite its young age and the fact that only a single developer has been maintaining it so far.

5.9.1 NUnit.Mocks

NUnit.Mocks is a side project and part of the NUnit framework. It was originally created to provide a simple and lightweight mocking framework that can be used to test NUnit itself without having to reply on external libraries and version dependencies. As part of NUnit is open source but it was never regarded as public, and there is little no no documentation about using it on the web today. Charlie poole, the current maintainer of NUnit has said that he is considering either removing it completely from the distribution of NUnit or to make is public in version 3.0 of NUnit. NUnit.Mocks does not support *Stub* syntax, requires strings to expect calls on method names, has no support for testing or firing events or parameter constraints (expected parameter values are hard coded in the test).

5.9.2 NMock

NMock if a port of the `JMock` framework from the Java language. As such it has been around quite a long time and has many users. Still, it is currently un-maintained for about past year or so, while the developers have gone to work on something bigger and better: Nmock2. NMock is open source.

NMock supports the *stub* syntax but still requires strings for method name expectations. It has no event raise or test support but does contain support for parameter constraints.

5.9.3 NMock2

NMock2 is a large leap ahead from NMock, in that the APIS have changed greatly to accommodate a more "fluent" calling interface. NMock2, unfortunately, by the time of this writing, has not released a version in the past year or so, which drives many people away from using it. NMock is open source.

NMock2 supports most if not all the features that RhinoMocks has, with the main difference being that method expectations are string based while in Rhino.Mocks they are call based (you call the method as part of the recording process). This version also features Parameter constraints, event raising and testing and callback abilities in the test.

5.9.4 TypeMock Isolator

TypeMock is a commercial Mocking framework, although it contains a free edition with the same features for use in open source project development. Because it is commercial it also has the advantage of having very good documentation which is always up to date, a support program, and continually updated versions. Isolator builds on top of the existing abilities of the other frameworks, and also allows to mock (named "fake" in the isolator api) classes that have private constructors, static methods and much more.

It does this by using a very powerful technique – attaching to the .NET Profiler APIS-- a set of APIs that allows intercepting a call to anything anywhere, including private members, statics, events – anything that goes on in the .NET runtime can be intercepted. Isolator is a perfect fit for testing not only new code, but also legacy code (untested, existing code) where testability can be quite impossible in many situations.

As a framework that has such powerful abilities, it's interesting to note that it has actually raised quite a stir in the Unit Testing/Mocking world of .NET. Some people claim that it may even be too powerful, in that it makes it very easy to simulate and break the dependencies of any object in your existing code, and in that way does not force you to think about how your design might need to change.

Others feel that it provides a sort of bridge to get started with testing even if the design is untestable, allowing one to learn better design abilities as they go, instead of having to refactor and learn better design skills as a prerequisite to testing.

The inability to mock an object in your code can mean that there is a chance to better the design to be more decoupled in natyre.That is why many people like to treat their tests as a way of flushing out design problems. See Appendix I on testability for more about this issue.

TypeMock Isolator came out second in my usage poll. The main reason it wasn't first is the fact that it is a commercial product and not free.

It is also the only other framework other than RhinoMocks that allows writing expectations as strongly typed method names, without needing to use strings. I'll touch more on that in the next sub section.

5.9.5 RhinoMocks

Rhino.Mocks, as I said earlier, is relatively new to the field (close to 2 and a half years), but already has gained a massive user base. It is open source and is continuously being worked upon and has frequent releases. Currently it is maintained by a single developer(god help us all if he gets a girlfriend). It has powerful APIS and one of the things that it's most noted for is avoiding the usage of strings inside the tests. To understand why strings in tests are bad, see the next section.

5.10 Why are method strings bad inside tests?

Following is an example of using NUnit.Mocks Vs. Rhino.Mocks to perform the same thing. Through that we'll see the differences in using strings for method names and the difference in using the mock framework syntax.

Given this interface that we'd like to mock using either RhinoMocks or NUnit.Mocks:

```
Interface ILogger
{
Void LogError(string msg, int level, string location);
}
```

Listings 5.21 and 5.22 show the syntax of tests that use this interface using NUnit.Mocks and Rhino.Mocks. What follows is an explanation of these approaches.

Listing 5.215.22 Using NUnit.Mocks to create expectations requires putting a string for the method names.

```
//Using NUnit.Mocks
DynamicMock mock = new DynamicMock(typeof(ILogger));
//record expecting a method call
Mock.Expects("LogError",
"param value 1 is string",
2,
"param value 3 is a string as well");
ILogger myMockInterface = mock.MockInstance as ILogger;
MytestedClass.SetLogger(myMockInterface);
```

The RhinoMocks code looks vastly different than the code used with NUnit.Mocks, as shown in figure 5.22:

Listing 5.22 Using Rhino.Mocks to create expectations requires no string usage for method names

```
//Using Rhino.Mocks
MockRepository mocks = new MockRepository();
ILogger simulatedLogger = mocks.CreateMock<ILogger>();
```

```
simulatedLogger.LogError("param value 1 is a string", 2,
“param value 3 is a string”);
mocks.ReplayAll()

 MyTestedClass.SetLogger(simulatedLogger);
MytestedClass.DoSomething();
mocks.VerifyAll();
```

Notice how the highlighted lines are different in listings 5.22 and 5.23 If I were to change the name of the “`LogError`” method on the `ILogger` interface, any tests that I have using NUnit would still compile and would only break at runtime throwing an exception telling me that a method named “`LogError`” could not be found.

With Rhino.Mocks, this is not a problem since we are invoking the actual method API as part of our “recording” stage. Any method changes would make the test not compile and let me know immediately that there is a problem with the test. With automated Refactoring tools such as exist in VS 2005, renaming a method is easier, but still most refactorings will skip strings part of automated renaming across the whole project (Resharper for .NET is an exception, it also corrects strings, but that is only a partial solution which may even prove problematic in some scenarios).

Let’s recap what the advantages are in Mock frameworks over handwritten mocks. Later we’ll also discuss things to watch out for when using Mock Frameworks.

5.11 Advantages in mock Object Frameworks

From what we’ve covered so far we can start to see some distinct advantages in using Mock Object frameworks, as discussed in the next sub sections.

DECLARATIVE EXPECTATION SYNTAX

When we create Mock Objects, we are establishing *expectations* as to what calls will be made against our Mock Object, as well as define any return values if necessary. With Mock Frameworks, the concept of an *expectation* is a 1st class member, and is part of the Mocking syntax. This makes it far easier to write multiple expectations on a Mock instance but keeping the test readable.

5.11.1 Easier parameter verification

Using Handwritten mocks testing that a method was given the correct parameter values when called can be quite a tedious process that takes time and patience to do correctly. With most (if not all) Mock frameworks out there, checking the values of parameters passed into methods becomes a trivial process that takes very little typing even for a large amount of parameters.

5.11.2 Easier verification of multiple method calls

One of the problems you might encounter in manually written mocks is how to check that under one specific test multiple method calls on the same method were made with each one done correctly according to the different expected parameter values. As we’ll see later, this is a trivial process with Mock frameworks.

5.11.3 Easier stub creation

Mock Frameworks are ill-named. They should really be called something like “Object Simulation Frameworks” because they can be used both for creating Mock objects and for creating Stubs more easily. As you recall, the only difference between a *Stub* and a *Mock* is that a Mock can be used to see if the test has passed, and a *stub* is mainly used to make sure the test runs correctly and to simulate various conditions under test. A *stub* can never break a test. Only a *Mock* can break a test. We’ll see examples of this later.

5.12 Traps to avoid when using Mocking frameworks

While there are many advantages to using a Mocking framework, there are some possible dangers in using it, overusing the mocking framework when a manual mock object would suffice, for example. Making the tests unreadable because of overusing mocks in a test or not separating tests well enough is another. Here’s a simple list of things to watch out for:

- Unreadable test code

- Verifying the wrong things
- Have no more than one mock per test
- Overspecifying the tests

Let’s look at each of these in more depth.

5.12.1 Unreadable test code

Using one mock in a test already makes the test a little less readable, but still readable enough so that an outside person can look at it and understand what’s going on. Having many mocks, or many expectations in a single test can ruin the readability of the test to a point where it’s hard to maintain the test or even understand what it’s testing.

If you find that your test becomes unreadable or hard to follow consider removing some mocks, or some mock expectations or separating the test into several smaller tests that are more readable.

5.12.2 Verifying the wrong things

Mock objects allow us to verify that methods were called on our interfaces, but that does not necessarily mean that we are testing the right thing. Testing that someone had subscribed to an event does not tell us anything about the functionality of that object. Testing that when the event is raised, something meaningful happens is a better way to test that object.

5.12.3 Have no more than one mock per test

It is considered a good practice to only test one thing in a single test. Testing more than one thing can lead to confusion and problems maintaining the test. Having two mocks in a test is the same as testing several things. If you can’t name your test because it does to many things, it’s time to separate it into at more than one test.

5.12.4 Overspecifying the tests

If your test has too many expectations, you may create a test that, with even the lightest of code changes, breaks down, even though the overall functionality still works. Consider this a more technical way of not verifying the wrong things. Testing interaction is a double edged sword. Test it too much and you start to lose sight of the big picture, the overall functionality. Test it too little and you’ll miss the important interaction scenarios between objects. Some of the ways to balance this effect out are to:

- Use non-strict mocks when you can. The test will break less often because of unexpected method calls. This helps when the private methods in the production code keep changing.
- Use stubs instead of mocks when you can. You only need to test one scenario at a time; the more mocks you have, the more verifications will take place at the end of the test. But only one of them will usually be the important one. The rest will just be noise against the current test scenario.
- Try to avoid using stubs as mocks in the same test (in that you stub a result value from a stub, but also verify that the stub method was called).
- Overspecification does not have to be about mocks and stubs. If you are asserting that some calculation is correct in your code, make sure your test does not repeat the calculation in the test code, or the bug might be duplicated and the test will magically pass. Try to always use hard coded known return values to assert against production code, and don’t expected values dynamically.

Overspecification is a very common form of test abuse. Make sure to keep your eyes on this by doing frequent test reviews with your peers.

5.13 Summary and practices to take away from this chapter

Dynamic Mock objects are pretty cool beings, and you should learn to use them at will. But it is important to lean into state-based testing (vs. interaction testing) whenever you can, so that your tests assume as little as possible about internal implementation details. Mocks should be used only when there is no other way to test the implementation, as they eventually lead to tests that are harder to maintain if you’re not really really careful.

Learn how to use the advanced features of a mock framework such as RhinoMocks and Isolator, and the world is your oyster – you can pretty much make sure that anything happens or does not happen, as long as your code is testable.

You can also shoot yourself in the foot by having over-specified tests that are not readable or very likely to break. The “art” lies in then “when” to use dynamic vs. hand written mocks. My “when” is when the code that uses the mock framework just starts to look plain ugly. That’s a nice tell that you may want to simplify things – use a hand written mock, or even change your test to test a different result that proves your point just the same but is easier to test

When all else fails and your code is really hard to test – you have three choices: Use a “god” framework like *TypeMock*, change the design or quit your job to find a place where Testability is not a naughty word.

Mock Object Frameworks can help make your testing life much easier, readable and maintainable. It’s important to know about the different frameworks out there with their strengths and weaknesses so you can pick and choose the framework with the features that are right for you. Mock object frameworks are there to help. But it’s important to also know when they might hinder your development more than they help. In legacy situations, for example. You might want to consider using a different framework because it makes more sense for these situations (based on its abilities). It’s about picking the right tool for the right job, so be sure to always look at the big picture when considering how to approach a specific problem in testing.

That’s it! We’ve covered the core techniques any developer writing unit tests should know before going in to write tests in the real world. The next part of the book deals with managing the test code, arranging tests and patterns for tests that you can rely on, maintain easily and understand clearly.

6
Test hierarchies and organization

Unit tests are just an important part of the application as the production source code. Just like regular code in the application, give careful thought to where the tests actually reside, both physically, and logically, in relation to the code under test. if you put the wrong tests in the wrong place, you may end up with some serious consequences, like people not running the tests you have written so carefully. If you don't figure out good ways to reuse parts of your tests, create utility methods for testing or use test hierarchies, you'll end up with test code that is either un maintainable or hard to understand. These are the kinds of things this chapter addresses, with patterns and guidelines that will help you shape the way your tests look, feel, and run and how well they play with the rest of your code and other tests.

First, let's consider where the tests are located.

To examine where we should put the tests, we need to first realize where they will be used and who will run them.

- As part of the automated build and continuous Integration process
- Multiple times a day by developers working locally on their machines

The automated build process is so important, the next section deals exclusively with this topic.

6.1 Automated builds running automated tests

The power of the automated build is one that cannot and *should* not be ignored. If you're planning to make your team more agile, and you're planning on being able to handle requirement changes as they come into your shop (by the client, by yourself etc.) you want to be able to:

- Make a small change to your code
- Run all the tests to make sure you hadn't broken any existing functionality
- Make sure your code can still integrate well and not break any of the other projects you depend upon

Running those tests gives you feedback from the code letting you know if you've broken any of the existing or new functionality. *Integrating* your code with the other projects will give you the feedback that you did not *break* the compilation of the code or things that are dependent on your code logically.

Integrating your code means usually:

- Getting the latest version of everyone else's source code from the source control repository
- Trying to compile it all locally
- Running all tests locally
- Fixing what you have found that has been broken; and only then,
- Checking in your source code

An automated build process is the idea of taking all of these steps that people do manually (or at least claim they do) and combining them under a special build script of sorts, that will make sure all these things are done without human interaction. If anything breaks in the process, the build server will notify the relevant parties of a "build break".

6.1.1 Anatomy of an automated build

An automated build should contain *at least* the bolded bullets from the following list, but may also include many other things:

1. **Get the latest version of all the projects in questions**
2. **Compile all the projects in the latest version**
3. Deploy build output to a test server
4. Run tests locally or on the test server
5. Create an archive of build outputs based on date and build number
6. Deploy outputs to staging or even production server
7. Configure and install components on target servers
8. **Notify relevant people if any of the steps have failed**
9. Merge databases
10. Create reports on build quality, history and test statuses
11. Create tasks or work items automatically if specific tasks have failed
12. Many other things

The easiest way to get an automated build going is by creating such a process and scripts as soon as the project starts out. It's much easier to create an automated build for a small project and then keep adding to it as the project grows than it is to start later in the game.

There are many tools that can help to create an automated build system. Some are free or open source; some are commercial.

A short list of tools you can look at is

- CruiseControl.NET
- NAnt and MSBuild files
- FinalBuilder and Visual Build Pro

These are all XML configuration based programs that allow you to create a series of steps that will be run in a hierarchy structure. You can create custom commands that can be run and you can schedule these builds to run nightly or even hourly, or even every time someone checks in source code to the system.

6.1.2 Triggering builds and continuous integration

The term "Continuous Integration" is literally about making the automated build and integration process run continuously. For example, have the build be triggered and run every time someone checks in source code to the system, or every 45 minutes.

One popular continuous integration tool is CruiseControl.NET. Is it fully open source and supports both the idea of "tasks" which are individual commands to run during a build, and the concept of "triggers" which is a configurable way to start a build automatically on events such as source control update.

Looking at commercial tools, Visual Studio Team System 2008 supports the notion of automated builds and continuous integration out of the box. For teams who feel it is a bit out of their budget range I'd recommend looking at *FinalBuilder* (www.FinalBuilder.com) and *Visual Build Pro* (www.kinook.com) . These two commercial and highly successful build tools allow visual editing and maintenance of automated build projects. Being visual means easier maintenance of the build file, which can get pretty scary for larger projects.

6.1.3 Automated build types

You may choose to configure many types of automated builds. Having done this at several companies and projects, I've found I needed to have builds that either produce different results or builds that run in specific amounts of time (but they all compile the code first):

- A nightly build
- Runs all the long running tests
- Runs system tests
- A release build
- Runs the nightly build
- Deploys to server and archives
- A CI (Continuous Integration) build
- Runs all the fast-running tests
- Finishes in less than 10 minutes
- Other kinds of builds based on specific scenarios of the product

In the end, though, it is important to realize that automated builds usually fall into two categories:

- Too long to run every 15 minutes
- Can be run every 15 minutes or less

The main thing you'll need to start doing as your start writing tests in your project is to categorize the tests that you write into these two types of running time:

- Fast running tests
- Slow running tests

These will usually map out nicely to:

- Integration tests
- Unit tests

The reason for this mapping is that integration tests run slower than unit tests which usually happen in memory.

Once you've categorized out the tests, the short and quick builds that run continuously are the ones that will run only a subset of the tests – the quick ones. If you can afford it, run *all* tests, which is a much better route. But, if you have to be picky and your tests really slow down a build, a subset that works fast is the next best thing.

6.2 Mapping out tests based on speed and type

It's quite easy to just run the tests and see which are integration tests and which are unit tests. Once you do, make sure to organize them in separate places. It does not have to be a separate test project. A separate folder and namespace should be enough.

Figure 6.1 shows a simple folder structure you can use inside your Visual studio projects:

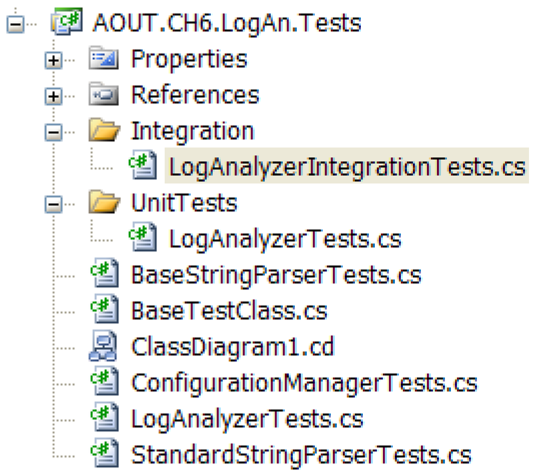


Figure 6.1 Integration tests and unit tests reside under a different folder and namespace, but remain under the same project.

Some companies, based on the build software and unit test framework they use, find it easier to just 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 6.2 shows how you'd set up two separate kinds of test projects under a single solution:

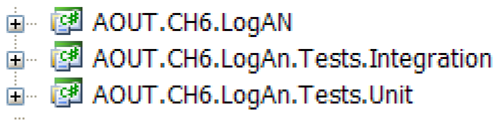


Figure 6.2: The unit test project and integration projects are unique for the Logan Project and have different namespaces.

Even if by some small chance 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. The worst is people not running your tests when the two types are mixed together, as we'll see in the next section

6.2.1 The human factor of separating unit from integration tests

Why would people not run the tests if they exist? One cause for that is that developers are lazy by default. That's not a bad thing, it's a reality.

If a developer on your team gets the latest version of the source code and tries to run all unit tests in it, and some of them fail, the causes might be:

- An actual bug is found in the code under test
- The test has a problem in the way it is written
- The test is no longer relevant
- The test requires some configuration in order to even run

All but the *last* point are valid reasons for a developer to stop and ask, "Why is this failing?" The last one is not relevant because a developer will often choose to ignore the test failure and go on to other more important things they are working on (sometimes while cussing a little bit for having to dig through and find that the problem is not even in the source code).

In many ways, having these "hidden" integration tests spread around your test project with unknown or unexpected configuration requirements (like a database connection), are bad form. These tests are less approachable, waste time and money on finding problems that aren't there, and generally discourage the developer from ever trusting the set of tests again. Like bad apples in a bunch, they make all the others lookd bad.

Next time something similar happens, the developer may not even look for a cause for the failure, and simply say, “Oh, that test *always/sometimes* fails; it’s *ok*.”

A technique that can help make sure this does not happen is one I like to call the *safe green zone*, as introduced next.

6.2.2 The safe green zone

Separate your integration and unit tests into separate “places”. By doing that you’ll give the developers on your team a “green test area” where they *know* that they should be able to get the latest version and run *all* tests in that namespace or folder, and they should all be *green*. If some of them are not passing there is a real problem, not a configuration problem in the test.

In addition, creating a separate *integration zone* (the opposite of a *safe green zone*)for the integration tests gives you a place to share not only that these tests may run slow , 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 that 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 all the quick tests (green zone).

Of course, none of the above matters if you don’t have your tests inside the source control tree, as we’ll see next.

6.3 Tests are part of source control

My editor told me that this section is very short. I agree. Tests *are* part of source control. Not much else to say about it. Just do it. It’s a section because it is that much a part of the development cycle, that it should not be missed.

If you haven’t figured this one out, let me be completely clear: The test code that you write should reside in a source control repository just like your real production code. In fact, you should treat your test code just as well as your 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 get automatically when they do *get latest version.*

Because unit tests are so intimately familiar with the code and API, they should always stay attached to the same version of the code they are testing. Getting version 1.0.1 of your product will get also version 1.0.1 of your tests for your product, which will be different from version 1.0.2 of your product and tests.

Also, being part of the source control tree is what allows your automated build processes to make sure they run the correct version of the tests against your software.

6.4 Mapping test classes to code under test

There are several patterns that you can use to create test classes so that the following properties of those classes remain true:

- It’s easy to look at a *project* and find all the tests that relate to it
- It’s easy to look at a *class* and find all the tests that relate to it
- It’s easy to understand what this test class is testing
- It’s easy to browse through and understand the tests in the class
- It’s easy to look at a *method* and find all the tests that relate to it

Let’s go through these one by one.

6.4.1 Mapping tests to projects

It may sound like a crude way of doing it, but it’s the easiest way I’ve come across to allow a developer to easily find all the tests for a specific project. I simply create a project to contain tests with the same name of the project under tests and adding `[.Tests]` to the end of the name.

So, if I had a project named Osherove.MyLibrary I would also have a test project named Osherove.MyLibrary.Tests.Unit and `Osherove.MyLibrary.Tests.Integration`

`See figure 6.2 as an example.`

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

6.4.2 Mapping tests to classes

There are several ways to go about mapping the tests for a single class you're testing. We'll start with the simplest and most common one and move our way up to more complex scenarios. The two main scenarios are one-to-one between classes and test classes and test-class-per-method.

ONE-TO-ONE CORRELATION

You want to be able to easily and quickly locate all tests for a specific class. The solution is not far from the previous pattern of 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 post fixed with [Tests

Let me make this clear – *Tests* and not "Test". This is a class that holds *multiple* tests for the class under test, and not just one test. Be accurate. Readability and language matter a lot when it comes to test code, and once you start rounding corners in one thing you soon round corners in many others, which leads to bad, bad things.

So, for a class called LogAnalyzer you'd create a test class in your test project named `LogAnalyzerTests.`

The one-to-one mapping (also known as *test class per class* pattern (Meszaros, 2007) is the simplest one and the most common patterns for organizing your tests. You write all the tests for all methods of a class under test in one big test class. Sometimes, some methods in the class under test will have so many tests that you may begin to notice that the test class becomes much less readable or browseable. Sometimes the tests for one method "drown out" the other tests for other methods.

TEST READABILITY

Why is this important? Because writing tests is as much about the person who is going to read the tests as it is about the computer who will run them. (A friend once said, "Imagine the person who will read your tests is a psychotic killer, and they know where you live. Don't annoy them to the degree they will have to use that knowledge.")

If the person who reads the test gets too busy browsing the test code instead of *understanding* it, it's a symptom that will cause maintenance headaches, as your code gets bigger and bigger. That's why you might think about doing it differently, as noted in the next sub section.

TEST CLASS PER METHOD

Test class per method is also known as *Testcase class per feature* (Meszaros 2007). If you seem to have lots of test methods that make readability of your test class problematic, find the specific method whose tests are drowning out the other tests for that class, and then create a separate test class especially for it.

For example, a class named LoginManager has a method `ChangePassword` you'd like to test but it has so many test cases that you want to separate it to a separate test class.

You might end up with two test classes named LoginManagerTests, which contains all the other tests and LoginManagerTestsChangePassoword which contains only the test for the `ChangePassword` method.

Other test class patterns exist, and I suggest you take a look at Gerard Meszaros' *Xunit Test Patterns* book for more. In my experience, I only used the two mentioned above.

6.4.3 Mapping tests to methods

Remembering that our main goals (other than making the test name readable and understandable) are to be able to easily find all test methods for a specific method under test, mapping a test method using a meaningful name is a very simple technique. Use the method name as part of the test name. See listing 6.1 for an example.

Listing 6.1: A test method that contains the main three parts of a readable test name. The first one is the method under test.

```
Class LoginManagerTests
{
    [Test]
    Public void ChangePassword_scenario_expectedbehavior()
{
}
}
```

For a full explanation on the naming convention in this test, see chapter 9, Readable Tests.

6.5 Building a test API for your application

Sooner or later, as you start writing tests for your application, your are bound to refactor them, create utility methods, utility classes, and many other constructs that exist either in the test projects or in the code under test itself, but are there solely for the purpose of testability or test readability and maintenance.

Here are some of the things you may find you 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 the developers.

First, let’s take a look at ways you can use inheritance to your favor when writing test classes.

6.5.1 Test class inheritance patterns

One of the most powerful arguments for Object Oriented code is the idea of reusing existing functionality instead of recreating it over and over again in other classes (also called the Don’t repeat yourself [DRY] principle [Andy Hunt 1999]). Because in .NET and most OO languages the unit tests you write are in an OO 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 by:

- Reusing utility and factory methods.
- Running the same set of tests over different classes. (I’ll go though that one more deeply.)
- Using common setup or teardown code (also useful for integration style testing).
- Creating testing guidance for the programmers who will derive from the base class

I’ll introduce you to three pattern variations based on test class inheritance, each one building on top of the one before it. I’ll also explain when you might want to use it and what are the pros and cons in each of them. The basic three patterns are:

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

Abstract Test Infrastructure

Definition: Create an abstract test class that will contain 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/teardown code to having special custom asserts, which are used throughout multiple test classes. The next example will show how we can re-use a setup method between two test classes.

Here’s the scenario: All tests need to override the default logger implementation in the application so that logging would be done in memory instead of to a file (that is, all tests need to *break the logger dependency* in order to run correctly).

Listing 6.2 shows these classes:

- The class and method we’d like to test (`LogAnalyzer`)

- `LoggingFacility` class: holds the logger implementation we'd like to override in our tests
- `ConfigurationManager` class: Another user of LoggingFacility we'll test later.
- The initial test class and method we're going to write(LogAnalyzerTests)
- `StubLogger` is an internal class that will replace the real logger implementation
- ConfigurationManagerTests class: holds test for ConfigurationManager

Listing 6.2: Example of not following the DRY principle in your test classes. The Setup method in both test classes repeats the same initialization process for the LoggingFacility class.

```
//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
    {
        [SetUp]
        public void Setup()
        {
            LoggingFacility.Logger = new StubLogger();
        }

        [Test]
        public void Analyze_EmptyFile_ThrowsException()
        {
            LogAnalyzer la = new LogAnalyzer();
            la.Analyze("myemptyfile.txt");
            //rest of test
        }
    }
    internal class StubLogger : ILogger
    {
        public void Log(string text)
        {
            //do nothing
        }
    }

  [TestFixture]
    public class ConfigurationManagerTests
    {
        [SetUp]
        public void Setup()
        {
            LoggingFacility.Logger = new StubLogger();
        }

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

Important points about this code are that:

The `loggingFacility` class is probably going to be used by many classes. We've designed it 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, and we'd like to test both of them – the LogAnalyzer and `ConfigurationManager` classes.

One possible way to refactor this code into a better state is to find a way to reuse the setup method that is essentially the same for both test classes. They both replace the default logger implementation. We could *refactor* the test classes into creating a base test class that contains the setup method as shown in listing 6.3.

Listing 6.3: A refactored solution. The SetUp method's behavior is inherited and run automatically.

```
public class BaseTestClass
    {
        [SetUp]
        public void Setup()
        {
            Console.WriteLine("in setup");
            LoggingFacility.Logger = new StubLogger();
        }
    }

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

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

Figure 6.3 shows this pattern more clearly:

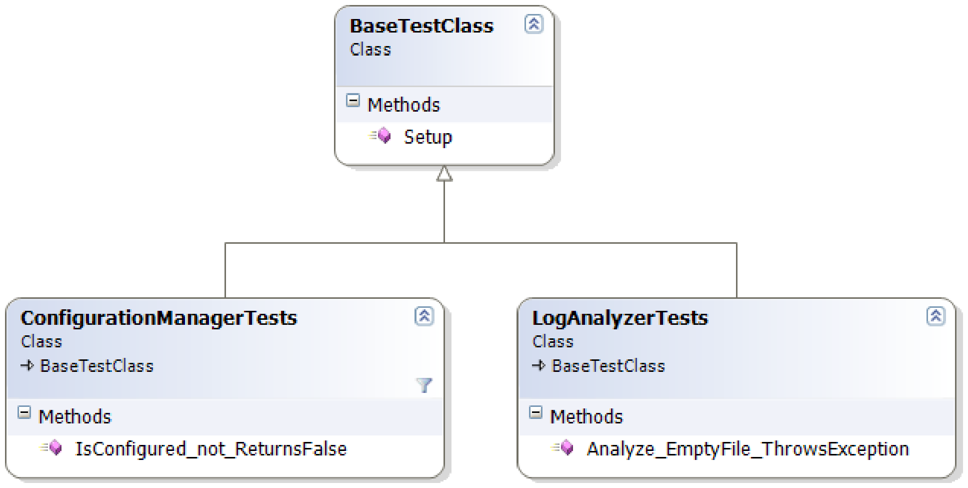


Figure 6.3 One base class with a common setup method, and two test classes that reuse that setup method.

The `Setup` method from the base class is now automatically run before each test in either of the derived classes. We've definitely reused some code, but this also has cons. The main problem we now introduced into the derived test classes is that the *reader* can no longer easily understand what happens every time setup is called. They will 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, but it also leads to more code reuse. There are pros and cons in every technique, unfortunately.

What if you wanted to have your own derived setup in one of the derived classes?
Most of the unit test frameworks (including NUnit) will allow you to make the setup method virtual and then override it in the derived class. Listing 6.4 shows how a derived class can have its own setup method but still keep the original setup method.

Listing 6.4 A derived test class with its own setup method that adds setup steps to the inherited setup.

```
       public class BaseTestClass
        {
            [SetUp]
            public virtual void Setup()
            {
                Console.WriteLine("in setup");
                LoggingFacility.Logger = new StubLogger();
            }
        }

    [TestFixture]
        public class ConfigurationManagerTests :BaseTestClass
        {
            [SetUp]
            public override void Setup()
            {
                base.Setup();
                Console.WriteLine("in derived");
                LoggingFacility.Logger = new StubLogger();
            }

               //…
        }
```

This notation is actually easier for the reader of the test class, because it specifically tells the reader that there is a base setup method that is called each time. You may be doing a good thing for your team by requiring to always override base methods and call their base class's implementation in your tests for the sake of readability, as seen in listing 6.5.

Listing 6.5 Override a setup method for clarity purposes even if you have nothing to add to it.

```
    [TestFixture]
        public class ConfigurationManagerTests :BaseTestClass
        {
            [SetUp]
            public override void Setup()
            {
                base.Setup();
    //we don't do anything else, this is for the sake of readability
            }

               //…
        }
```

This type of coding may feel a bit weird at first, but the reader of the tests will thank you for making sure they know what's going on.

Template Test Class Pattern

Definition: Create an abstract class that contains abstract test methods that derived classes will have to implement.

The driving force behind this pattern is being able to *dictate* to deriving classes which tests they should always implement. It is usually used when there is a need to create one or more test classes for a set of classes that implement the same interface (behavior contract), for example, a set of parsers all implementing a "parse" method that act the same but each one acting on different input types.

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

Figure 6.4 shows an example for a base class that helps to test data-layer create, retrieve, update, delete (CRUD) classes:

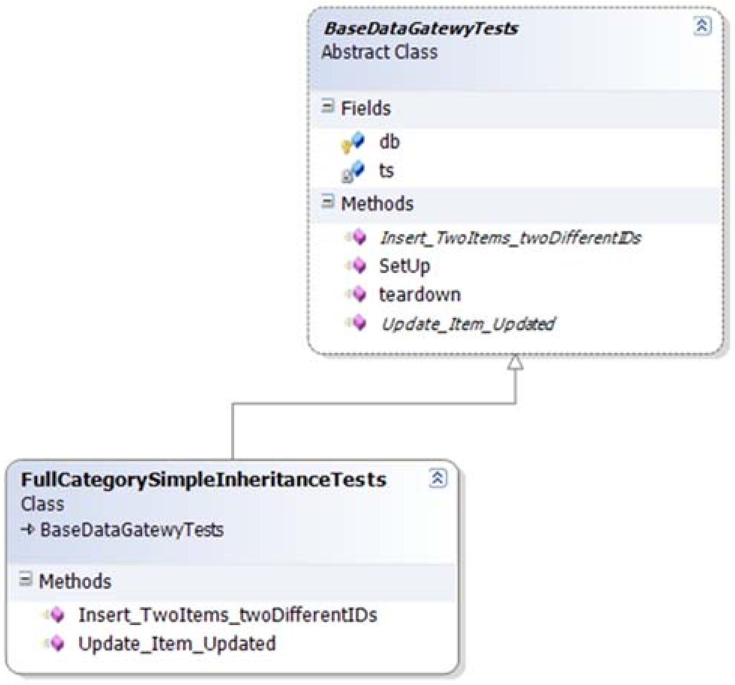


Figure 6.4: Template test pattern. Base class contains abstract tests which derived class must implement. Makes sure people don't forget important tests.

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 implement, as well as provide guidance on what kinds of tests you'd want to write next. It's essential that the naming of the tests is understandable.

The next sub section deals with an interesting question – what if you were to inherit *real tests* from the base class, and not abstract ones?

Abstract Test Driver Class

Definition: Create an abstract test class that contains test method implementations that all derived classes inherit be default without needing to re-implement them.

In other words, instead of having *abstract* test methods, you implement *real tests* on the abstract class which your derived classes will inherit. It is essential that your tests will not test explicitly one class type, but instead test against an interface or base class in your product code under test. Let's see a real scenario.

Let us say you have the following object model that you would like to test, shown in figure 6.4.

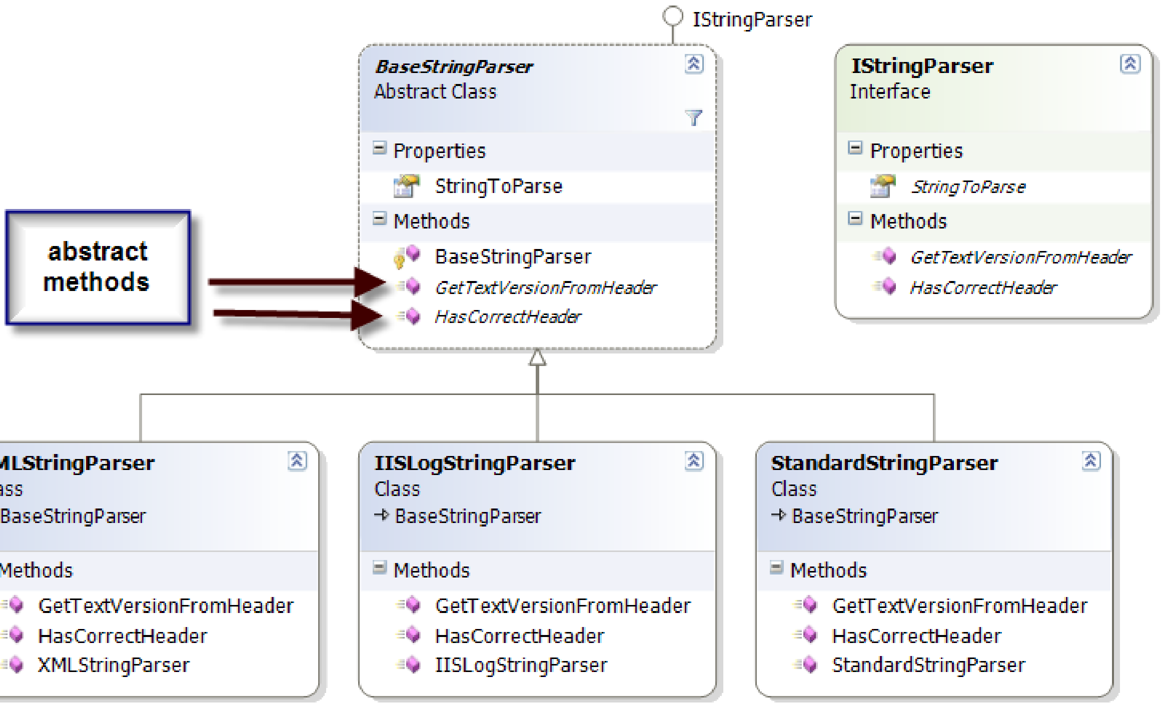


Figure 6.4 A typical inheritance hierarchy that we'd like to test comprises of an abstract class and

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) we can get some sort of version from a custom header we look for and to check weather that header is indeed valid for the purposes of our application.

The XMLStringParser, IISLogStringParser and StandardStringParser classes derive from this base class and implement the methods with specific logic regarding the specific string type.

You'd usually start off with a set of tests for one of the derived classes (assuming the abstract class has no logic to test in it) and then you'd have to write mostly the same kinds of tests for the other class that has the same functionality over different implementation.

Here's a possible set of tests for the StandardStringParser we might start out with *before* we refactor our test classes shown in listing 6.6.

Listing 6.6: a possible 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.GetTextVersionFromHeader();
        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
        }


        [Test]
        public void HasCorrectHeader_NoSpaces_ReturnsTrue()
        {
            string input = "header;version=1.1.1;\n";
            StandardStringParser parser = GetParser(input);

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

        [Test]
        public void HasCorrectHeader_WithSpaces_ReturnsTrue()
        {
            string input = "header ; version=1.1.1 ; \n";
            StandardStringParser parser = GetParser(input);

//rest of the test
        }

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

//rest of the test
        }


    }
```

Note how we use the “GetParser()” helper method to refactor away the creation of the parser object, which we use in all the tests. The reason that we use the helper method and not a [setup] method is that the constructor takes the input string to parse and so each test needs to be able to create the version of the parser for its own specified inputs to tests against.

Soon enough, you’ll start writing tests for the other classes in the hierarchy, and you’ll find you’d want to repeat the same tests that you did for this specific parser class for all the other parsers as well. After all, all the other parsers should have the same outward behavior – getting the header version and validating that the header is Valid. It’s *how* they do it that changes based on various internal things in the class, but the behavior semantics are the same. That means that for each class that derives from BaseStringParser we’d write at least the same basic tests, with only the type of the class under test that changes.

Instead of repeating all those tests manually, we can create our own ParserTestsBase class that contains all the basic tests we’d like to perform on anything that implements the IStringParser interface (or anything that derives from BaseStringParser). Listing 6.7 shows an example of this base class. Bolded parts are things that have changed.

Listing 6.7 Abstract test base class that contains the test logic for an IStringParser interface.

```
        public abstract class BaseStringParserTests
        {
            protected abstract IStringParser GetParser(string input);

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

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

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

            [Test]
            public void GetStringVersionFromHeader_WithRevision_Found()
            {
                string input = "header;version=1.1.1;\n";
                IStringParser parser = GetParser(input);
    //...
            }

            [Test]
            public void HasCorrectHeader_NoSpaces_ReturnsTrue()
            {
                string input = "header;version=1.1.1;\n";
                IStringParser parser = GetParser(input);

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

            [Test]
            public void HasCorrectHeader_WithSpaces_ReturnsTrue()
            {
                string input = "header ; version=1.1.1 ; \n";
                IStringParser parser = GetParser(input);
    //...
            }

            [Test]
            public void HasCorrectHeader_MissingVersion_ReturnsFalse()
            {
                string input = "header; \n";
                IStringParser parser = GetParser(input);
    //...
            }
        }
```

The things that are different and important in the implementation of the base class are that:

We made the GetParser() method abstract, and changed its return type to `IStringParser`. That was we can override this factory method in derived test classes and return the type of the parser we'd like to test.

The test methods themselves are only getting an `IStringParser` interface and do not know the actual class they are running against.

A derived class can choose to add tests against a specific sub class of `IStringParser`, by adding another test method in its own test class (as we'll see in the next example).

Once we have the base class in order, we can add tests to the various sub classes very easily. Listing 6.8 shows how we can write tests for the StandardStringParser by deriving from `BaseStringParserTests`.

Listing 6.8: Derived test class makes it very easy to test full hierarchies of objects by overriding a small number of factory methods.

```
        [TestFixture]
        public class StandardStringParserTests : BaseStringParserTests
        {
            protected override IStringParser GetParser(string input)
            {
                return new StandardStringParser(input);
            }

    [Test]
            public void GetStringVersionFromHeader_DoubleDigit_Found()
            {
                //this test is specific to the StandardStringParser type
    string input = "header;version=11;\n";
                IStringParser parser = GetParser(input);

                string versionFromHeader = parser.GetTextVersionFromHeader();
                Assert.AreEqual("11", versionFromHeader);
            }

        }
```

Note how we only have two methods in the derived class:

- The factory method that tells the base class what instance of the class to run tests on
- A new test which we added which may not belong into the base class, or may only be specific to the current type we are testing.

Figure 6.5 shows the visual inheritance chain that we have just created:

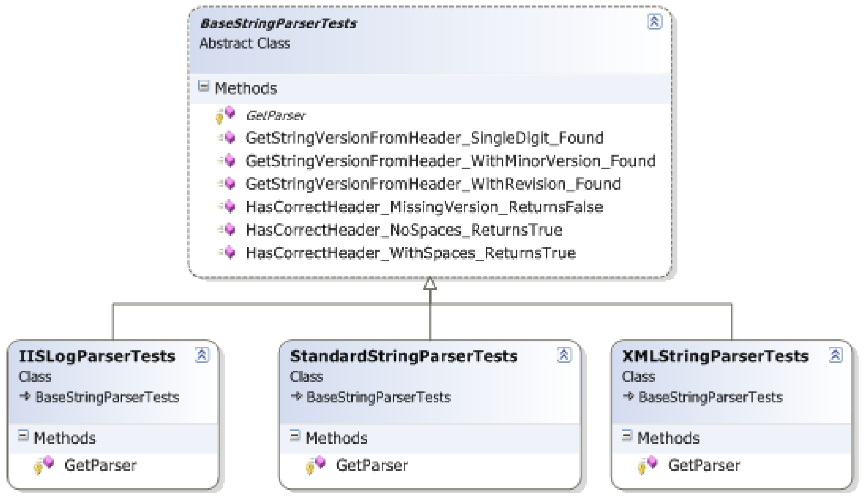

BaseStringParserTests
Abstract Class
Methods
GetParser
GetStringVersionFromHeader_SingleDigit_Found
GetStringVersionFromHeader_WithMinorVersion_Found
GetStringVersionFromHeader_WithRevision_Found
HasCorrectHeader_MissingVersion_ReturnsFalse
HasCorrectHeader_NoSpaces_ReturnsTrue
HasCorrectHeader_WithSpaces_ReturnsTrue
IISLogParserTests
Class
BaseStringParserTests
Methods
GetParser
StandardStringParserTests
Class
BaseStringParserTests
Methods
GetParser
XMLStringParserTests
Class
BaseStringParserTests
Methods
GetParser

Figure 6.5: A standard Test Class Hierarchy implementation. Most of the tests are in the base class but derived classes can add their own specific tests.

So how do you turn existing code into this pattern? Here's how.

REFACTORING YOUR TEST CLASS INTO A TEST CLASS HIERARCHY

Most developers don't start out writing their tests with these inheritance patterns in mind. Instead, they write the tests normally as shown in listing 6.7. The steps to make it into a base class are fairly easy, especially if you have some of the known IDE Refactoring tools available like the ones found in Eclipse, IntelliJ IDEA or Visual Studio 005 (JetBrains Resharper or Refactor! From DevExpress). Here is a list of possible steps on the road to refactoring your test class:

1. Refactor: Extract super class
   a. Create a base class (BaseXXXTests)
   b. Move the factory methods (like `GetParser`) into the base class
   c. Move all the tests to the base class
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 Generics to create the inheritance patterns, as explained next.

VARIATION: USING .NET GENERICS TO IMPLEMENT TEST HIERARCHY

You can use generics as part of the base test class so that you don't even need to override any methods in derived classed, just declare the type you are testing against. Listing 6.9 shows both the generic version of the test base class and a derived class from it.

Listing 6.9: Implementing Test case inheritance with .NET Generics

```
        public abstract class StringParserTests<T>
            where T:IStringParser
        {
            protected T GetParser(string input)
            {
                return (T) Activator.CreateInstance(typeof (T), input);
            }

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

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

            //more tests
            //...
       }

    //and this is the derived test class:
        [TestFixture]
        public class StandardStringParserGenericTests
                            :StringParserTests<StandardStringParser>
        {
        }
```

There are several things that change in the generic implementation of the hierarchy:

The GetParser factory method no longer needs to be overridden. We simply create the object using `Activator.CreateInstance` (that allows creating objects without knowing their type) and sending in to the constructor the input string arguments.

The tests themselves do not use the `IStringParser` interface, but instead use the T generic type.

The Generic class declaration contains the where clause which specifies that the T type of the class must implement the `IStringParser` interface.

Overall I don't find more benefit in using generic base classes. Any performance gain we would get is insignificant to our tests, but I leave it to you to find your own voice and see what makes more sense to your team. It is more a matter of taste than anything else.

6.5.2 Test utility methods and classes

As you write the tests, you also get to create many simple utility methods that may or may not end up inside your test class. These utility classes end up being a big part of your test API and may end being a simple object model you would use as you develop your tests.

Types of utility methods you may end up with are:

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

Types of utility classes you may end up refactoring your utility methods into:

- Special assert utility class that contains 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.

Just having those utility methods around doesn't mean anyone will use them, as you may know. I've been to plenty of projects where developers kept "reinventing the wheel" and recreating utility methods because they didn't know they already exist. That's why making your API known is very important as a next step.

6.5.3 Make your API known

It is imperative that the people who write tests know about the various APIs that have been groomed and grown over the time spent writing the application and its tests, so that they don't fall into the "re-invent the wheel" trap and create their own utility methods. There are several ways to make sure your APIs are used:

- Have teams of two people write tests together (at least every once in a while), where one of the people is familiar with the existing APIs and can teach the other person, as they write new tests, about the existing benefits and code that could be used.
- Have a short (no more than a couple of pages) document or a cheat-sheet that details the types of APIs out there and where to find them. It can contain the categories listed above or your own, and point to where you can find them in the code. You can create such a short document 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 automatic its generation process:
- Have a known set of prefixes or postfixes on the API helpers' names (helperXX for example).
- Have a special tool that parses out the names and their locations and generates a document that lists them and where to find them, or just some simple directives that the special tool can parse from special comments you put on them.
- Automated the generation of this document as part of the automated build process
- Discuss changes to the APIs during team meetings, in the form of short one to two sentences about the main changes and where to go look for the "meat" about them. That way the team knows that this is important and is always on people's minds.
- Use employee orientation days to go over this document for new employees.

- Perform test reviews (as opposed to code reviews) that make sure tests are up to standards of readability, maintainability and correctness, but also that the right APIs are used when needed.

Following one or more of these recommendations can help make sure your team does not lose productivity for lack of knowledge, and will help create a “shared language” the team can use when writing their tests.

6.6 Summary and practices to take away from this chapter

Let’s look back and see what we can draw out from the chapter we’ve just been through.

- Whatever you do, however you do it, automate it, and use an automated build procedure to run it as many times as possible during day or night.
- Separate the integration tests from the unit tests (Slow from fast) so that your team can have their own “safe green zone” where they know all the tests must pass.
- Map out tests by project and by unit/integration type (slow/fast) into different directories or folders or namespaces (or all of the above). I usually do all three types of separation.
- A test class hierarchy is a very powerful technique to apply the same set of tests to multiple related types in a hierarchy, or that share a common interface\base class
- At the same time. A test class hierarchy can make tests less readable, especially if there is a shared setup method in the base class. If you find this to be the case, use helper classes and utility classes instead of hierarchies. Different people have different opinions on when to use which, but readability is usually the key cause for not using hierarchie.
- Make your API known to your team. If you don’t, you will lose time and money reinventing many of the APIs over and over again by team members who don’t know any better.

The next three chapters will deal practices you can use to make your tests more maintainable, readable and correct (test the right things).

6.7 Works Cited

Andy Hunt, Dave Thomas. *The Pragmatic Programmer.* Addison Wesley, 1999.
Meszaros, Gerard. *xUnit Test Patterns, Refactoring Test Code.* Addison Wesley, 2007.

7
The pillars of good tests

No matter how you organize your tests, or how many you have, if you can't trust them enough they aren't worth squat. In fact, the tests that you write should have three properties that together make them "good" in my eyes. They are:

- Trustworthy
- Maintainable
- Readable

This chapter deals with these three pillars of "Good" tests with a series of practices you can use as a checklist when doing test reviews in your team. Being able to trust a test will make sure your developers will actually *want* to run the tests they write, and will listen to the test results with confidence. It also means making sure the tests themselves don't have bugs and that they are testing the right thing. A non-maintainable test is the worst nightmare for projects that have them, since they pose a schedule risk, or the risk of losing the tests the minute the project goes into a more aggressive schedule. People will simply stop maintaining and writing tests that take to long to change. Readability is not only about being able to read a test but also about being able to figure out where the bugs are and are not. Without readability the other two pillars fall apart pretty quick – maintaining the tests becomes harder, and thus you can't trust them anymore.

Together, the three pillars make a sound investment of your time. Drop just one of them and you run the risk of wasting everyone's time.

7 .1 Writing trustworthy tests

A trustworthy test will have the following indications:

- If it passes, you:
- Don't tell yourself "I'll just step through the code in the debugger to make sure".
- Just trust that it passes and the code it tests works for that specific scenario.
- If it fails, you:
- Don't tell yourself "oh, it's supposed to fail" or "it doesn't mean the code is not working"
- Feel there is a problem in your code
- Don't feel like you have a problem in your test

In short, a trustworthy test is one that actually makes you feel like you know what's going on and can do something about it. In this chapter I'll introduce the following guidelines and techniques:

- Making sure you're testing the right thing
- When is it ok to remove or change tests
- Assuring code coverage

- Avoiding test logic
- Making it easy to run
- Testing only one thing

I've found that tests that follow these guidelines tend to be tests that I feel I can "trust" more than others, and that I feel confident that they will continue to find errors in my code. Let's begin with the first practice.

7.1.1 When to remove or change tests

In general, once you have tests in place, unit tests or otherwise, you should not change or remove them because they are there as your safety net to let you know if anything existing breaks as you change your code. That said, there are times you might feel compelled to change or remove existing tests. To understand when it might cause a problem and when it's reasonable to do so, let's look at the reasons for each. The main reason is of course, when a test fails.

A test can "suddenly" fail for several reasons:

- Production bug: A bug is found in the production code under test
- Test bug: There is a bug in the test
- Semantics: The semantics of the code under test changed, but not the functionality
- Conflict/invalid tests: Production code was changed to reflect a conflicting requirement

Other reasons for changing or removing tests might be:

- Renaming/refactoring the test
- Duplicate tests

Let's see how you might want to deal with each of these cases.

PRODUCTION BUG

A production bug occurs when you change the production code, and suddenly an existing test breaks. If indeed this is a bug in the code under test, your test is just fine, and you shouldn't need to change the test or touch it. This is the best and most desired outcome of having tests.

WHAT DO YOU DO?

Because production bugs are one of the main reasons we'd like to have unit tests in the first place, the only thing left to do is to fix the bug in the production code. Don't touch the test.

TEST BUG

If there is a bug in the test, you need to change the test. Bugs in test are notorious. Understanding that you actually have a bug in your test may be the most difficult part in this process. In fact, I've detected several stages of the developer psyche when a test bug is encountered:

4. *Denial*. The developer will keep chasing her tail and look for a problem in the code itself, changing it, and all the while other tests start failing as well as she introduces *new* bugs into production code as the result of hunting for the illusive bug that is actually in the test.
5. *Amusement*. By other developers. The developer will call another developer if possible and they will hunt for the bug together.
6. *Debuggerment*. The developer will patiently debug through the test and see that perhaps there is a problem in the test. This can take anywhere between an hour to a couple of days.
7. *Acceptance & Slappage.* Developer will eventually realize where the bug is and slap herself on the forehead for the missed work.

So, you've realized it and gone through all the stages of acceptance. Now what?

WHAT DO YOU DO?

If and when you reach the conclusion that the test has a bug, and you start fixing it, it's important to make sure that the bug is actually fixed, and that the test doesn't just magically pass by testing the wrong thing. This pattern is:

- Fix the bug in your test.
- Make sure the test fails when it should.
- Make sure the test passes when it should.

While the first line is quite straight forward, the last two steps may require a little more explanation.

This step makes sure you are still testing the correct thing, and that your test can still be trusted. If your code displays the bug that the test is supposed to catch, but the test never fails, how will you know? That is what this step is about.

Once you have fixed your test, go to the production code under test and change it so that it manifests the bug that the test is supposed to catch. Then run the test. If the test fails, that means its half working. The other half will be completed by the next step. If the test does not fail, you are most likely testing the wrong thing.[1]

Once you see the test fail, change back your production code so that the bug no longer exists. The test should now pass. If it does not, it means you either still have a bug in your test, or you are testing the wrong thing.

SEMANTICS/API CHANGES

A test can fail when the production code under test changes so that an object begin tested now needs to be *used* differently, even though it may still have the same end functionality.

Consider the simple test appearing in listing 7.1.

Listing 7.1 A simple test against the LogAnalyzer class.

```
[Test]
public void SemanticsChange()
{
    LogAnalyzer logan = new LogAnalyzer();
    Assert.IsFalse(logan.IsValid("abc"));
}
```

Now, let's say that later a new semantic change has been added to the LogAnalyzer class, in the form of an "Initialize" method. You have to call Initialize on the LogAnalyzer class before calling any of the other methods on it.

If you introduce this change in the production code, the second line of the test in listing 7.2 will throw an exception because initialize was not called. The test will be broken *but it is still a valid test*. The functionality is tests still works, only the semantics of using the object under test have changed.

WHAT DO YOU DO?

In this case, the test we will need to *change* the test to include the new semantics as shown in listing 7.2.

Listing 7.2 The changed test uses the new semantics of LogAnalyzer

```
            [Test]
            public void SemanticsChange()
            {
                LogAnalyzer logan = new LogAnalyzer();
    Logan.Initialize();
```

[1] I've even seen developers accidentally delete the Asserts from their tests when fixing the bugs in the tests. You'd be surprised how often that happens and how effective this technique is for these cases.

```
        Assert.IsFalse(logan.IsValid("abc"));
    }
```

Changing semantics make up most of the bad experiences that developers have had with writing and maintaining unit tests, because the burden of changing the tests as the API of the code under test keeps changing gets bigger and bigger. In chapter 8 we'll take a look at techniques that can ease the pain of semantic changes by applying refactoring techniques to the tests, not just the code.

Listing 7.3 is an example of a more maintainable version of the test in listing 7.2.

Listing 7.3: A refactored test uses a factory method to create and initialize the object under test.

```
[Test]
 public void SemanticsChange()
 {
     LogAnalyzer logan = MakeDefaultAnalyzer();
     Assert.IsFalse(logan.IsValid("abc"));
 }

 private LogAnalyzer MakeDefaultAnalyzer()
 {
     LogAnalyzer analyzer = new LogAnalyzer();
     analyzer.Initialize();
     return analyzer;
 }
```

In this case, because we refactored the body of the tests to use a utility method we can do the same for the other tests and have the use the same utility method. If the semantics of creating and initializing the object changes again, we won't need to change all the tests that create this object, just one little utility method. Again, we'll see more maintainability techniques further in this chapter.

CONFLICTING/INVALID TESTS

A conflict problem is when the production code introduces a new feature that was requested, which is in direct conflict with a different test. This means that, instead of the test discovering a bug, it discovered conflicting requirements.

Here's a short example:

13. Customer requests LogAnalyzer to not allow file names shorter than 3 letters. The analyzer should throw an exception in that case.
14. The feature is implemented and tests are written (not necessarily in that order)
15. Much later on customer realizes that 3-letter file names have a use and requests for them to be handled in a special way.
16. We add the feature and the production code is changed. Then we write new tests The production code no longer throws an exception.
17. Suddenly, an old test (the one with a 3-letter file name) breaks because it expected an exception. Fixing the production code to make that test pass breaks the new test that expects 3 letter file names to be handled in a special way.

As the last point explains, the *either-or* scenario, where only one of two tests can pass can serve as a red flag warning that these may be conflicting tests.

WHAT DO YOU DO?

In the case of conflicting tests, you first need to make sure this is the case in question. Once confirmed, you need to decide which requirement you need to keep. You should then remove[2] the invalid requirement and its tests.

This also serves as a reminder that sometimes tests can bubble up problems in understanding or conflicting requirements to the customer which gets to decide on the validity of each requirement.

[2] Removed, not commented out! That is why god created source control systems.

Up to now, we explored the reasons why a test may fail, but there are times when you'd like to change or remove a test even if it does not fail:

- Renaming or refactoring a test
- Duplicate tests

They may seem obvious, but the point is these should normally be the only cases where tests can be changed even if they pass and nothing is wrong with the code.

RENAMING/REFACTORING THE TEST

An unreadable test (as chapter 9 shows) is a test that is a problem more than a solution. It can hinder your code's readability and your understanding of what the problem it finds is.

If you encounter a test that has a bad name or can be made into a more maintainable test, I urge you to change the test code (but remember that the basic functionality of the test should not change, that is the meaning of refactoring). Listing 7.3 shows one such example of refactoring a test.

DUPLICATE TESTS

When dealing with a team of developers it is very common to come across multiple tests for the same functionality that different developers wrote over time. Should those duplicates be removed?

I'm not crazy about the idea of removing duplicate tests for a couple of reasons:

- The more (good) tests you have the more certain you are to catch bugs
- You can read the tests and understand the different ways or semantics of testing the same thing.
- There are cons in having multiple tests for the same things.

Let's explore some of the pros and cons for duplicate tests:

- Cons:
- It may be harder to maintain across different tests of the same thing.
- Some tests may have higher quality than others, and you still need to review them all for correctness.
- Multiple tests break when a single thing does not work. (Is that really a con?)
- Test names must be different, or tests may be spread across different classes
- Possible more maintainability issues due to multiple tests
- Pros:
- Tests may have little difference and overall make up for a larger and better picture of the object being tests.
- Some tests may be more expressive than others to compensate for badly written tests other developers have written, so more tests may mean a better chance of test readability.

While I am not crazy about the idea of removing duplicate tests, I usually go ahead and remove them anyway, because the cons outweigh the pros.

7.1.2 Avoiding logic in tests

In the previous section, we discussed the possibility of a test bug. This possibility increases in a Cartesian manner as you include more and more logic in your tests. I've seen plenty of tests that should have been simple, being turned into generically mutating, randomly generating, thread-creating, file-writing monsters that are little test engines in their own right. Sadly, because they had a *[Test]* attribute on them, the writer of those tests didn't even consider that they might have bugs, or to write them in any maintainable manner.

Eventually those *test monsters* take more time to debug and verify than they one they save. The thing to remember here is that all monsters start out small.

If you have any of the following inside a test method your test contains logic that should not be there:

- Switch/if/else statements
- Foreach/for/while loops

A test that contains logic usually means you're testing more than one thing at a time, which is not recommended since the test becomes less readable and more fragile. But test logic also adds just enough complexity to the test that it may contain a hidden bug you weren't aware of.

Those test bugs are one of the most annoying things you can imagine as a developer because you'll almost never try to search the cause for a failing test in the test itself, which is exactly the problem.

Sooner or later some guru in the company looks at the test and starts thinking "what if we just had the method loop and throw random number as input? We'd surely find lots more bugs this way!" Yep. Especially in your tests.

Making test methods be monster tests means you also remove the original simpler test in favor of the monster, and that just makes it harder to find bugs when they exist in the production code. If at all, test monsters should be *added* and not *replace* existing tests.

Tests should, as a general rule, be a series of method calls with no control flows, not even try-catch (so the test will raise them up and not swallow them), and with assert calls. Anything more complex than that brings on the following:

- Test is harder o read and understand
- Sometimes test is hard to re-create[3]
- Test is more likely to have a bug or test the wrong thing
- Naming the test may be harder (since it does multiple things)

7.1.3 Testing only one thing

If your test contains more than a single assert it may be testing more than one thing. That doesn't sound so bad until you consider:

- How will you name your test?
- Naming the test may seem like a simple task, but if you are testing more than one thing, giving the test a good name that tells the reader what is being tested becomes almost impossible. When you test just one thing, naming the test becomes pretty easy.
- What happens if the first assert fails?

In most test frameworks (NUnit included), a failed assert message throws a special type of exception that is caught by the test framework runner. When the test framework catches that exception, it means the test has failed. Unfortunately for us, exceptions, by design, do not let the code that is written after them to continue. The method exists on the same line the exception is thrown.

Here is an example in listing 7.4.

Listing 7.4 A test with multiple asserts. If the first assert fails, the second one never runs.

```
[Test]
        public void TestWithMultipleAsserts()
        {
            LogAnalyzer logan = MakeDefaultAnalyzer();

            Assert.IsFalse(logan.IsValid("abc"));
            Assert.IsTrue(logan.IsValid("abcde.txt"));
        }
```

If the first assert (IsFalse()) fails in this test, it will throw an exception, which means the second assert will never run in that test.

Consider Assert failures as symptoms of a disease and Asserts as indication points or blood checks for the body of the software. The more symptoms you can find, the easier the disease will be to diagnose and treat. If only part of them fail, you lose sight of other possible symptoms from that line onwards. After a failure, subsequent Asserts aren't executed. These unused Asserts could provide valuable data (or symptoms) that would help you

[3] Imagine a multi threaded test, or a test with random numbers that suddenly fails.

quickly narrow your focus and discover the underlying problem. So running multiple Asserts in a single test adds complexity with little value. You should run Additional Asserts in separate, self-contained unit tests so that you have a good opportunity to see what fails.

7.1.4 Making tests easy to run

In chapter 6, I discussed the “safe green zone” for tests. If developers don’t trust your tests to be run “out of the box” easily and be consistent, they will not run them because they will not trust them to have true results. Refactoring your tests so they are easy to run and provide consistent results will make them feel more trustworthy.

7.1.5 Assuring code coverage

How do you know if you have good coverage for your new code?

There are automated tools that help you detect code coverage (NCover, Visual Studio Team Test..). Find a good one and stick with it, and make sure you never have really low coverage (less than 20% means you are missing a whole bunch of tests).

When doing code reviews and test reviews, you can do the manual option which is great for ad hoc testing of the test:

Try *commenting out* a line or a constraint check. If all tests still pass, you might be missing some tests or the current tests aren’t testing the right thing.

How do you know you’ve added the correct test that was missing? Here’s a set of steps I usually perform:

- Comment out the code you think is not being covered.
- Run all the tests.
- If all the tests pass you are missing a test or are testing the wrong thing.
- Keep the code in a comment.
- Try to write a new test that fails proving that the code you have commented is missing.
- Uncomment the code you commented before.
- The test you wrote should not pass without touching the test, just running it.
- If the test still fails modify the test until it passes (you may have a bug in your test).
- When the test passes, comment that previous piece of code before to make sure that the test fails when the code is missing, and passes when the code is there.

You never know when the next developer will try to play with your code. He may try to optimize it or wrongly delete some essential line. If you don't have a test that will fail, other developers may never know they made a mistake.

You might also want to try replacing various usages of parameters or internal variables, in your method under test with constants (making a bool always true and see what happens, for example)

Like taking care of a plant, you need to watch out for wild weeds here and there, make sure it has room to grow and that it is healthy. The trick is in making sure all this taking care of doesn’t take up too much time to make it worth your while. That’s exactly what the next section is about: *Maintainability*.

7.2 Writing maintainable tests

Maintainability is one of the core issues most developers face writing unit tests. Eventually the tests we write seem to become harder and harder to maintain and understand, and seem to break on seemingly small changes done in the codebase, without finding bugs. Every little change to the system seems to break one test or another even if bugs may not really exist. Understanding the test after some time has passed seems to be hard to do. Like any piece of code, time adds a layer of “indirection” between what you think the code does, and what it really does.

This chapter will go over some techniques I’ve gathered through my experience working and writing unit tests in various teams, and have all been learned the hard way, so you don’t have to. Some of these include the idea of removing duplication in your tests, enforcing test isolation, testing against only public contracts and others.

7.2.1 Testing private/protected methods

Private or protected methods are usually private for a good reason, in the developer's mind. Sometimes that reason is to hide implementation detail, so that the implementation may change later, without the end functionality changing. It could also be Security related, or IP related (obfuscation, for example).

Think of private/public methods like going through a grocery store. When you go to a grocery store, looking for some milk, and finding it on a specific shelf, you don't really care how that milk got on that shelf, as long as its expiration date is ok and there is enough of it for you to take home.

Changing a private method is like someone rearranging the stockroom at the grocery store so that milk is easier to put on the shelves, or that there's enough milk in the first place.

Changing a public method is like changing where the milk resides or putting a different kind of milk altogether on the shelf.

The public contract (the overall functionality) is what you care about. Testing how the stockroom is arranged may lead to breaking tests even though the milk will still end up on the correct shelf.

Our test just became more brittle for no good reason.

When you write tests against such methods, you're writing tests against code that is more prone to change than public methods, which have a defined contract against the code that uses them.

In other words, when you're testing a private method you're testing against an internal contract inside the system, which may very well change. Internal contracts inside a system are very dynamic, as you refactor the system, they can change as well. When they change, your test could fail just because some internal work is being done differently, while the eventual functionality of the system remains the same.

What do you do? If a method is worth testing, it might be worth making it *public*, *static*, or at least *internal*, making it define a public contract against any user of it. Here are the common practices you can do.

MAKING METHODS PUBLIC

Making a method public is not a bad thing by default. Yes, your animalistic basic instincts as a developer may shout at you that this goes against the OO principles you were raised on, but think about this for a minute. Wanting to test a method means that this method has a known *behavior* or *contract* against the calling code. By making it public you are, in fact, making this official. By keeping the method private, you tell every developer that comes after you and reads that code that they can change the implementation of the method without worrying too much about unknown code that uses it, because it only serves as part of a bigger scale of things that together make up a contract to the calling code.

EXTRACTING THE METHOD TO A NEW CLASS

If your method contains a lot of logic that can stand on its own, or it uses state in the class that is only relevant to the method in question, it may be a good idea to extract the method into a new class, with a specific role in the system. That class can then be tested separately. Also, see the book (Feathers n.d.) around page 250 "RuleParser" and "RuleTokenizer" for good examples of this technique.

MAKING METHODS STATIC

If your method does not use any of the class variables in the class it is written, you might want to think about refactoring the method by making it static. That makes it much more easily testable, but also states that this method is a utility method of sorts and thus has a known public contract specified by its name.

MAKING METHODS INTERNAL

When all else fails and you really can't afford to expose the method in an "official" way, you might want to make it internal, and then use the `[InternalsVisibleTo("TestAssembly")]` attribute on the production code assembly so that that tests can still call that method. It is my least favorite one but sometimes there is not choice (for example, security reasons, lack of control of the code's design).

Making it internal is still not a great way to make sure your tests are more maintainable, because a coder can still feel like it's an internal method and thus easier to change.

By exposing the methods as explicit public contracts, you force the coder who may change them to make sure they know that these methods have a real usage contract that they cannot break. Simply removing them is not a good option because someone is using them. "But only the tests use them. That's just stupid," you may be saying to yourself right now. If you are, you are wrong. The production code uses these methods too. Otherwise, there would be no reason for us to write them. You're just making them official.

Does that mean there should eventually be no private methods in the code base? No. Usually, especially with test driven development, we write tests against methods that are public, and those public methods are later refactored into calling smaller, private methods. All the while the tests against the public methods continue to pass.

The next section is just as important, if not more so, and deals with the idea of code duplication in tests, and how that might hurt us.

7.2.2 Removing duplication

Duplication in our unit tests can hurt us as developers just as much (if not more) than duplication in production code. The DRY[4] Principle should be in effect in test code as if it were production code. Duplicated code means more code to change when one particular aspect we test against may change. Changing a constructor, changing the semantics of using a class, and more, can have a large effect on tests that have a lot of duplicated code.

To understand why, let's begin with a simple example of a test, seen in listing 7.5.

Listing 7.5 A class under test and a test that uses it

```
      public class LogAnalyzer
        {
            public bool IsValid(string fileName)
            {
                if (fileName.Length < 8)
                {
                    return true;
                }
                return false;
            }
        }

        [TestFixture]
        public class LogAnalyzerTestsMaintainable
        {
            [Test]
            public void IsValid_LengthBiggerThan8_IsFalse()
            {
                LogAnalyzer logan = new LogAnalyzer();
                bool valid = logan.IsValid("123456789");
                Assert.IsFalse(valid);
            }
        }
```

The test at the bottom of the listing seems reasonable, until you introduce another test for the same class, and end up with two tests like in listing 7.6.

Listing 7.6 Can you spot the duplication?

```
    [Test]
            public void IsValid_LengthBiggerThan8_IsFalse()
            {
                LogAnalyzer logan = new LogAnalyzer();
                bool valid = logan.IsValid("123456789");
                Assert.IsFalse(valid);
            }

            [Test]
            public void IsValid_LengthSmallerThan8_IsTrue()
            {
                LogAnalyzer logan = new LogAnalyzer();
                bool valid = logan.IsValid("1234567");
                Assert.IsTrue(valid);
            }
```

[4] Don't Repeat Yourself

What's wrong with the tests in listing 8.2? The main problem is that if the way use LogAnalyzer change (its semantics), the tests will have to be maintained independently of each other, leading to more maintenance work. Here's an example of such a change (you can also see an example in chapter 7, section 7.2.3) in listing 7.8.

Listing 7.8: LogAnalyzer changed semantics and now requires Initialization

```
public class LogAnalyzer
    {
    private bool initialized=false;

        public bool IsValid(string fileName)
        {
           If(!initialized)
Throw NotInitializedException(
"The analyzer.Initialize() method should be called before any other operation!");

            if (fileName.Length < 8)
            {
                return true;
            }
            return false;
        }
        public void Initialize()
        {
           //initialization logic here
           ...
           Initialized=true;
        }
    }
```

Now, the two tests that we've written will both break, because they both neglect to call `Initialize()` against the `LogAnalyzer` class. Because we have code duplication (both of the tests create the class within the test) we need to go into each one and change it to call `initialize()` first.

We can refactor the tests to remove the duplication of creating the `LogAnalyzer` into a single `CreateDefaultAnalyzer()` method which both tests will call. We could also push the creation and initialization up into a new `[setup]` method in our test class.

Following are some other ways we can refactor our tests.

REMOVING DUPLICATION USING A HELPER METHOD

Listing 7.9 shows an example of refactoring the tests into a more maintainable state by introducing a shared factory method that creates a "default" instance of `LogAnalyzer`. We then simply add a call to `initialize()` within that factory method instead of going through all the tests.

Listing 7.9 AAdding the Initialize() call is only done in one place –the factory method.

```
        [Test]
        public void IsValid_LengthBiggerThan8_IsFalse()
        {
            LogAnalyzer logan = GetNewAnalyzer();
            bool valid = logan.IsValid("123456789");
            Assert.IsFalse(valid);
        }

        [Test]
        public void IsValid_LengthSmallerThan8_IsTrue()
        {
            LogAnalyzer logan = GetNewAnalyzer();
            bool valid = logan.IsValid("1234567");
            Assert.IsTrue(valid);
        }

        private LogAnalyzer GetNewAnalyzer()
        {
            LogAnalyzer analyzer = new LogAnalyzer();
            analyzer.Initialize();
            return analyzer;
        }
```

Factory methods are not the only way to remove duplication in tests, as the next section shows.

REMOVING DUPLICATION USING [SETUP]

We could easily have just initialized LogAnalyzer within the Setup method as noted in listing 7.10.

Listing 7.10 Using a setup method to initialize a shared variable is a simple way of removing duplication

```
        [SetUp]
        public void Setup()
        {
            logan=new LogAnalyzer();
            logan.Initialize();
        }

        private LogAnalyzer logan= null;

        [Test]
        public void IsValid_LengthBiggerThan8_IsFalse()
        {
            bool valid = logan.IsValid("123456789");
            Assert.IsFalse(valid);
        }

        [Test]
        public void IsValid_LengthSmallerThan8_IsTrue()
        {
            bool valid = logan.IsValid("1234567");
            Assert.IsTrue(valid);
        }
```

In this case, we don't even need a line that creates the analyzer object in each test, since a shared class instance is initialized before each test with a new instance of `LogAnalyzer`, and then `initialize()` is called on that instance. However, beware; using a setup method for removing duplication is not always a good idea, as I explain in the next section.

7.2.3 [Setup] methods are not always a good idea

`[Setup]` is easy to do. In fact, it's almost too easy – enough so that developers tend to use it for things it was not meant to do, to a point where the test become less readable and less maintainable. Nevertheless, before I point out those ways it's important to review the things that setup methods can't do, which you can accomplish with simple helper methods:

- Setup methods can only help when you need to initialize things
- Removing duplication isn't always just about creating and initializing new instances of objects. Sometimes it is about removing duplication in assertion logic, calling out code in a specific way.
- Setup methods can't have parameters or return values
- Setup methods can't return objects o the test. They always have to be run before the test executes, so they have to be more generic in the way they work. Tests sometimes need to request specific things or called shared code with a parameter for the specific test (for example – retrieve an object and set its property to a specific value)
- Setup methods are meant to be general for all the tests
- You only want to have things in a setup method that all the tests in the class use, or it will be hard to read and understand.

Now that we know the basic things you can't do with a setup method, let's see how developers try to get over these problems in their quest to use a setup method no matter what, instead of using helper methods.

There are several ways developers abuse setup methods; Initializing objects that they only use in some of the tests in that class

- Having setup code that is long and hard to understand
- Setting up mock objects and fake objects within the setup method.

Let’s take a closer look at these.

INITIALIZING OBJECTS THAT ARE ONLY USED BY SOME OF THE TESTS

This sin is a deadly one. Once you commit it, it becomes very hard to maintain the tests or even read them because the setup method becomes loaded quickly with objects that are only specific to some of the tests. Listing 7.11 shows what our test class would look like if we added a FileInfo object that is initialized in the setup but only used by one test in the class.

Listing 7.11 The setup method initializes a FileInfo object that is only used by the middle test, the test class harder to maintain.

```
[SetUp]
public void Setup()
{
    logan=new LogAnalyzer();
    logan.Initialize();

    fileInfo=new FileInfo("c:\\someFile.txt");
}

private FileInfo fileInfo = null;
private LogAnalyzer logan= null;

[Test]
public void IsValid_LengthBiggerThan8_IsFalse()
{
    bool valid = logan.IsValid("123456789");
    Assert.IsFalse(valid);
}

[Test]
public void IsValid_BadFileInfoInput_returnsFalse()
{
    bool valid = logan.IsValid(fileInfo);
    Assert.IsFalse(valid);
}

[Test]
public void IsValid_LengthSmallerThan8_IsTrue()
{
    bool valid = logan.IsValid("1234567");
    Assert.IsTrue(valid);
}

private LogAnalyzer GetNewAnalyzer()
{
   …
}
```

Why is it less maintainable? Because as someone who wants to read the tests for the first time and understand why they break, you need to:

- Go through the setup method to understand what is being initialized
- Assume that objects in the setup method are used in all tests
- Find out later you were wrong and read the tests again more carefully to see which test uses the specific objects that may be cause the problems
- Dive into the test code deeper than you would need for no good reason, taking more time and effort to understand what I’m doing.

HAVING A SETUP CODE THAT IS LONG AND HARD TO UNDERSTAND

Because the Setup method is providing only one place in the test to initialize things, developers tend to shove in a lot of code to initialize many things, which inevitably can get quite cumbersome to read and understand. One solution is to refactor the calls to initialize specific things into helper methods that are called from the setup method. That means that refactoring the setup method is usually a very good idea – the more readable it is the more readable your test class will be.

However, there is a fine line between over refactoring and readability –sometimes I've notices that over refactoring can lead to less readable code. This is a matter of personal preference – you just need to watch out and always be on the lookout if your code is suddenly becoming less readable. I recommend doing this with a partner to get feedback. Developers (me included) become too enamored with code they have written, and having a second pair of eyes when refactoring can lead to very good and objective results. Having a code review (a test reviews) by a peer after the fact is also good but not as productive as doing it as it happens.

SETTING UP MOCKS AND FAKES IN THE SETUP METHOD

It's not always a bad idea to use the setup method to create mock objects and fake objects, but it's important to make sure that only those mocks and fakes that are used *in all* the tests in the class are initialized in the setup method or it will become hard to read and maintain.

My own personal preference is to have each test create its own mocks and stubs by calling helper methods within the actual test, so that the reader of the test knows exactly what going on without needing to jump from test to setup to understand the full picture.

7.2.4 Enforcing Test Isolation

Test isolation is the biggest single cause of test blockage[5] I've seen consulting and working on unit tests. The basic concept is that a test should always run in its own little world, isolated from even the *knowledge* that other tests out there may do similar or different things.

The test who cried "fail"

One of the projects I was involved in was having unit tests that were behaving strangely and got even stranger as time went on. A test would begin to fail and then suddenly pass for a couple of straight days. A day later, it would fail seemingly randomly and other times it would pass even though it should have failed because code was changed to remove or change its behavior.

It got to the point where developers would tell each other:

"Ah it's OK. If it sometimes passes, it means it passes."

Turns out that test was actually calling out a different test as part of its code, which, when failed, would break the original test.

Moreover, it only took us 3 days to figure this out, after spending a month living with the situation as is. When we finally had the test working correctly, we discovered that we had created a bunch of *real* bugs in our code that we were ignoring because we were getting what we thought were "false positives" from the failing test. It seemed that "The boy who cried wolf" is a story that still holds true even in development.

When tests are not isolated well, they can step on each other's toes just enough to make your work life miserable, regretting ever deciding to try unit testing on this project, apologizing to your team lead for even trying to make things a little bit better, and promising yourself never again. Yes, the effect is that powerful, and I've seen this in real life. But why is it so powerful and destructive? It's simple. We are building tests as things we are supposed to trust to find other problems. We don't test the tests. THEREFORE, we don't even bother looking for problems in the tests. Therefore, when there is a problem in the tests (and breaking test isolation is one of the more nasty problems to find), it can take us a lot of time to find it simply because we start looking for the causes of where the test broke elsewhere, our production code, for example.

We then think we *found the problem* and as we start *fixing* the production code, we break more and more tests with the original problem in the tests still raising issues. It only goes downhill from there if you don't know where to look.

[5] When tests ruin your work life.

There are several ways to break test isolation:

- Tests expecting to be run in a specific order or expecting information from other test results
- Tests calling other tests
- Tests sharing in-memory state without rolling back
- Integration tests with shared resources and no rollback

Let's name these as simple *anti-patterns* and go through them.

ANTI-PATTERN: CONSTRAINED TEST ORDER

Tests are coded to expect a specific state in memory or in an external resource, or even in the current test class, which was created from running other tests in the same class before the current test.

THE PROBLEM:

Most test platforms (including NUnit, JUnit and MbUnit) do not guarantee specific test run order execution, so what passes today may fail tomorrow.

Example:

A test against LogAnalyzer that expects that an earlier test had already called `Initialize()` on it is seen in listing 7.12.

Listing 7.12: Constrained Test Order: the second test will fail if run first.

```
[TestFixture]
public class IsolationsAntiPatterns
{
    private LogAnalyzer logan;
    [Test]
    public void CreateAnalyzer_BadFileName_ReturnsFalse()
    {
        logan = new LogAnalyzer();
        logan.Initialize();
        bool valid = logan.IsValid("abc");
        Assert.That(valid, Is.False);
    }

    [Test]
    public void CreateAnalyzer_GoodFileName_ReturnsTrue()
    {
        bool valid = logan.IsValid("abcdefg");
        Assert.That(valid, Is.True);
    }
}
```

There is a myriad of problems than can occur when tests do not enforce isolation. Here's a short list:

- Test may suddenly start breaking when a new version of the test framework is introduced that runs the test in a different order
- Running a specific subset of the tests produces different results than running all the tests or a different subset of the tests – Inconsistent test results
- Maintaining the test becomes harder and more cumbersome because you need to worry about the other tests that relate to this test and how they affect the state it expects.
- Your test may fail or pass for the wrong reason: a different test failed/passed before it leaving the resources the current test expects in an unknown state.
- Removing or changing some tests may affect the outcomes of other tests.
- It's hard to name your test appropriately

The Causes for test isolation usually are:

- Flow testing – A developer tries to write tests that need to run in a specific order so that they can test "flow" execution, a big use case composed of many actions, or they are trying to do a full integration test where each test is one steps in that test.

- Laziness in Cleanup – A developer was lazy and not taken care of cleaning up any state their test may have changed back to its original form, which causes other developers to write tests that "depend" on this symptom knowingly or unknowingly.

These in turn can be solved in various manners:

- *Flow testing* – Instead of writing flow related tests (long running use cases for example) in unit tests, you should consider using some sort of integration testing framework like FIT/Fitnesse or more QA related products such as AutomatedQA, WinRunner and such.
- *Laziness in Cleanup* – If you're too lazy to cleanup your database after testing, or your file system after testing, or your simple memory based objects, consider moving to a different profession. This isn't a job for you.

ANTI-PATTERN: HIDDEN TEST CALL

Tests contain one or more direct calls to other tests in the same class or other test classes, which will cause tests to depend on one another. For example, Listing 7.13 shows the test "*CreateAnalyzer_GoodNameAndBadNameUsage*" calling a different test at the end, creating a dependency between the tests and breaking both of them as isolated units.

Listing 7.13 One test calling another breaks isolation and introduces a dependency between tests. [TestFixture]

```
public class HiddenTestCall
{
    private LogAnalyzer logan;
    [Test]
    public void CreateAnalyzer_GoodNAmeAndBadNameUsage()
    {
        logan = new LogAnalyzer();
        logan.Initialize();
        bool valid = logan.IsValid("abc");
        Assert.That(valid, Is.False);

        CreateAnalyzer_GoodFileName_ReturnsTrue();
    }

    [Test]
    public void CreateAnalyzer_GoodFileName_ReturnsTrue()
    {
        bool valid = logan.IsValid("abcdefg");
        Assert.That(valid, Is.True);
    }
}
```

SYMPTOMS:

- Running a specific subset of the tests produces different results than running all the tests or a different subset of the tests – Inconsistent test results
- Maintaining the test becomes harder and more cumbersome because you need to worry about the other tests that relate to this test and how and when they call it..
- Your test may fail or pass for the wrong reason: a different test failed/passed before it calling your test, or leaving some shared variables in an unknown state.
- Removing or changing some tests may affect the outcome of other tests.
- It's hard to name the tests that call other tests.

CAUSES:

- Flow testing – A developer tries to write tests that need to run in a specific order so that they can test "flow" execution, a big use case composed of many actions, or they are trying to do a full integration test where each test is one steps in that test.

- Trying to remove duplication – A developer tries to remove duplication in the tests by calling other tests (which have code they don't want the current test to repeat).
- Laziness in separating the tests– A developer was lazy and not taken the time to create a true separate test and refactor the code accordingly, and "shortcutted" their way into calling a different test.

SOLUTIONS:

- Flow testing – See the solution in section 8.3.1
- Trying to remove duplication – Don't ever remove duplication by calling another test from a test. You're violating the ability for that test to rely on the setup and teardown methods in the class and are essentially running two test in one (since that test has an assertion as well). Instead try to see what code it is you don't want to write twice in that test you're calling and refactor just that piece of code into a third method which both your test and the other test call.
- Laziness in separating the tests – If you're too lazy to separate your tests this of all the work you'll have to do extra just because you didn't. Separate the tests and treat calling another test from your test a code smell(a sign that something may be wrong). Try to imagine a world where the current test you're writing is the only test in the system so it cannot rely on any other test to help it in its path to pass/fail result.
-

ANTI-PATTERN: SHARED STATE CORRUPTION

This anti pattern manifests in two major ways, independent of each other.

- Tests touch shared resources(in memory or external resources such as databases, file system etc..) without cleaning up or rolling back any changes they have made to those resources
- Tests do not setup the initial state they need before they start running, and rely on the state to just "be there".
- It only takes one of these to points to see the symptoms I show later on.

THE PROBLEM:

Tests rely on specific state to have consistent pass/fail behavior. If that test does not control the state it expects or other tests corrupt that state for whatever reason the test is unable to run properly or report the correct pass/fail result in a consistent manner.

EXAMPLE:

Assume we have a Class Person with very simple features: it has a list of phone numbers, and the ability to Search for a number by specifying what it starts with. Listing 7.14 shows a couple of tests that do not cleanup or setup a Person object instance correctly.

Listing 7.14: The second test will fail because the first test has already "corrupted" the state of the person variable

```
[TestFixture]
public class SharedStateCorruption
{
    Person person = new Person();

    [Test]
    public void CreateAnalyzer_GoodFileName_ReturnsTrue()
    {
        person.AddNumber("055-4556684(34)");
        string found = person.FindPhoneStartingWith("055");
        Assert.AreEqual("055-4556684(34)", found);
    }

    [Test]
    public void FindPhoneStartingWith_NoNumbers_ReturnsNull()
    {
        string found = person.FindPhoneStartingWith("0");
        Assert.IsNull(found);
    }
}
```

In this example, the second test (expecting a null return value) will fail because the previous test has already added a number to the person instance.

SYMPTOMS:

- Running a specific subset of the tests produces different results than running all the tests or a different subset of the tests – Inconsistent test results
- Maintaining the test becomes harder and more cumbersome because you may break the state for other tests – breaking them without realizing it.
- Your test may fail or pass for the wrong reason: a different test failed/passed before it leaving the shared state in a problematic condition or not cleaning up after it has ran.
- Removing or changing some tests may affect the outcomes of other tests, seemingly randomly.

CAUSES:

- Not setting up state before each test – A developer did not consider setting up the state required for this test, or assumed the state was already correct.
- Using shared state – A developer used shared memory or external resources for more than one test without taking precautions.
- Using static instances in the tests – A developer sets static state that is used in other tests.

SOLUTIONS:

- Not setting up state before each test – This is a mandatory practice that any developer writing unit tests should learn. Use either setup method or call specific helper methods at the beginning of the is what you expect it to be.
- Using shared state – In many cases you don't need to share state at all. Having separate instances per test of object is the safest way to go if you can do that.
- *Using static instances in the tests* – This is problematic in that you'll need to be really careful in how your tests manage the static state. Be sure to clean up the static state using setup methods or teardown methods. Sometimes it is useful to have direct helper method calls that reset the static state from within the test to make things clearer. If you are testing singletons it is worth it to add public/internal setters so your tests can reset them to a "clean" object instance.

Now that we've looked at isolating tests, let's make sure that we can get the full story when a test fails by managing our *assert*s.

7.2.5 Multiple asserts do's and don'ts

There are several reasons that lead to people writing multiple asserts in a single test method. Some are valid, and some are just bad workarounds that cause more trouble than they are worth.

- Repeat the same test with different arguments
- Check multiple aspects of the same object as one logical unit
- Run a logical execution flow with checkpoints during execution
- Test multiple object states during a specific execution context

We'll review these and see when they are valid and when they can be replaced with better more maintainable test code, but first, there is one big important question to answer for you to have motivation at all about this section.

Why should I care if the other asserts aren't executed?

If only one assert fails, you never know if the other asserts in that same test method would have failed or not (you may *think* you know but it is an assumption until you can prove it in a failing or passing assert). When people see only part of the picture, they tend to make judgment call about the state of the system, which can turn out wrong. The more information you have about all the asserts that have failed or passed, the better equipped you are to understand where in the system a bug may lie, and where it does not.

I've seen people (I was one of them) go on a wild goose chase hunting for bugs that weren't really there just because they only say one assert out of X fail. Had they (me) bothered to see if the other asserts failed or passed, they may have realized the bug is in a different location.

Sometimes people go and "find" bugs that they think are real, but when they fix them, the assert that previously failed now passes, but the *other* asserts in that test now fail (or continue to fail, we don't know if they passed before because they never executed).

Here are several patterns that people use in their tests that end up with multiple asserts, along with how to solve the problems they create.

7.2.6 Pattern: repeating test with different arguments

To understand this situation, let's take a look at the example in listing 7.15:

Listing 7.15: A test that contains multiple asserts.

```
[Test]
public void CheckVariousSumResults()
{
    Assert.AreEqual(3, Sum(1001, 1, 2);
    Assert.AreEqual(3, Sum(1, 1001, 2);
    Assert.AreEqual(3, Sum(1, 2, 1001);
}
```

What is obvious here is that there is more than one test in this test method. The author of the test method tried to save some time by repeating three simple tests as three very simple asserts. So what's the problem here? When assertions fail, they *throw exceptions*. In NUnit's case, they throw a special *AssertException* that is caught by the NUnit test runner, which understands this exception as a signal that the current test method has failed. Another important thing to realize is that once an assert clause throws an exception, no other line executes in the test method. That means that if the first assert in listing 8.10 failed, then the other two assert clauses will never execute.

There are several ways to achieve the same goal:

- Create a separate test for each assert
- Use Parameterized Tests
- Wrap with try-catch

Refactoring into multiple tests

Multiple tests are what you're doing anyway, but you're not getting the benefit of test isolation (a failing test causes the other asserts/tests to not execute). Instead, create separate test methods with meaningful names that represent each test case. Listing 8.11 shows an example refactoring from the code in listing 8.10:

Listing 7.16: A refactored test class with three different tests

```
[Test]
public void Sum_1001AsFirstParam_Returns3()
{
    Assert.AreEqual(3, Sum(1001, 1, 2);
}

[Test]
public void Sum_1001AsMiddleParam_Returns3()
{
    Assert.AreEqual(3, Sum(1, 1001, 2);
}

[Test]
public void Sum_1001AsThirdParam_Returns3()
{
    Assert.AreEqual(3, Sum(1, 2, 1001);
}
```

As you can see, the outcome of the refactoring gives us three separate tests, each one with a slightly different meaning of how we're testing the unit under test. The benefit? If one of those tests fails, the others will still run. The problem? This is *too verbose* for most people to handle and most developers won't want to do such a refactoring because they would feel it is overkill for this benefit. While I disagree that it is overkill (about 20 seconds of work to get the benefit), I agree that the verbosity is an issue. It's an issue because if people won't like to do it, they won't do it and we'll end up with many tests that have multiple "mini tests" inside them that keep hiding other "mini tests" when they fail.

That's why in many unit testing frameworks including MbUnit and NUnit, there is a custom attribute you can use that provides the same goal with much more concise syntax, as the next section shows.

USING PARAMETERIZED TESTS

Both MbUnit and NUnit support the notion of Parameterized tests using a special attribute called [RowTest][6]. Listing 7.17 shows how you can use the [RowTest] attribute to achieve running the same test with different parameters, using a single test method:

Listing 7.17: A refactored test class with three different tests

```
[RowTest(1001,1,2,3)]
[RowTest(1,1001,2,3)]
[RowTest(1,2,1001,3)]
  public void SumTests(int x,int y, int z,int expected)
  {
      Assert.AreEqual(expected, Sum(x, y, z);
  }
```

Parameterized test methods in NUnit and MbUnit are different from regular tests in that they can take parameters. They also expect at least one `[RowTest]` attribute to be placed on top of the current method instead of a regular `[Test]` attribute. The attribute takes any number of parameters which are then mapped at runtime to the parameters that the test method expects in its signature (in our case we expect 4 arguments). In our test we call an assert method with the first three parameters and use the last one as the expected value. This allows the test writer a declarative way of creating the same test with different inputs.

The best thing about it is that if one of the `[RowTest]` attributes fails, the other attributes are still executed by the test runner, so the test runner shows the full picture of pass-fail state in all tests.

If you'd like to use parameterized tests with NUnit, the attribute that does this is located in a separate DLL named `nunit.framework.extensions.dll`. The attribute comes built into MbUnit.

WRAPPING WITH TRY-CATCH

Some people think it's a good idea to do try-catch on each assert to catch its exception and write it to the console and then continue to the next statement, "bypassing" the problematic nature of exceptions in tests. I am not one of those people. The reason is that using Parameterized tests is a far better way of achieving the same thing. This section is written here to tell you one thing: *don't do it*. Use parameterized tests instead of try-catch around multiple asserts.

We just finished talking about ways to refactor and understand multiple asserts as a way to create multiple tests. Next, we look at writing multiple asserts to test one objects from multiple aspects.

7.2.7 Pattern: testing multiple aspects of the same object

Let's take another example of a test with multiple asserts, only this time it's not trying to be multiple tests in one test, it's trying to check multiple aspects of the same state as one atomic unit. If even one aspect fails, we need to know about it. Listing 7.18 shows such a test.

Listing 7.18: Testing multiple things on the same object as one test.

```
[Test]
```

[6] For NUnit you would need to download a special NUnit Addin that provides this ability

```
            public void Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFields()
            {
                LogAnalyzer log = new LogAnalyzer();
                AnalyzedOutput output = 
                    log.Analyze("10:05\tOpen\tRoy");

                Assert.AreEqual(1,output.LineCount);
                Assert.AreEqual("10:05",output.GetLine(1)[0]);
                Assert.AreEqual("Open",output.GetLine(1)[1]);
                Assert.AreEqual("Roy",output.GetLine(1)[2]);
            }
```

We are testing that the parse output from out log analyzer eventually worked by testing each field in the result object separately. They either all work or the test should fail.

MAKING TESTS MORE MAINTAINABLE

Listing 7.19 shows a way to refactor the test so that it is easier to read and maintain:

Listing 7.19: An object assert is much more concise and easy to read.

```
             [Test]
            public void Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFields2()
            {
                LogAnalyzer log = new LogAnalyzer();
                AnalyzedOutput expected = new AnalyzedOutput();
                expected.AddLine("10:05", "Open", "Roy");

                AnalyzedOutput output = 
                    log.Analyze("10:05\tOpen\tRoy");

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

Instead of adding multiple asserts, we create a full object that we will compare against, we set all the properties we think should be on that objects, and then compare the result and the expected object in one single assert. The advantage here is that it is much easier to understand what you are testing, as well as to figure out that all of this is one logical block that should be passing, not many separate tests.

It's important to understand that for this kind of testing, the objects being compares should override the `Equals()` method or the comparison between the objects will not work.

DON'T FORGET TO OVERRIDE TOSTRING()

Another thing you might want to do is override the `ToString()` method of compared objects so that if tests fail, you will get more meaningful error messages. For example, here's the output of the test in listing 8.14 if it failed without us overriding the `ToString()` method on `AnalyzedOutput`.

```
    TestCase
    'AOUT.CH7.LogAn.Tests.MultipleAsserts.Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseF
    ields2'
    failed: 
      Expected: <AOUT.CH789.LogAn.AnalyzedOutput>
      But was:  <AOUT.CH789.LogAn.AnalyzedOutput>
        C:\GlobalShare\InSync\Book\Code\ARtOfUniTesting\LogAn.Tests\MultipleAsserts.cs(41,0): at 
    AOUT.CH7.LogAn.Tests.MultipleAsserts.Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFi
    elds2()
```

By implementing ToString() n both the AnalyzedOuput class and the LineInfo class (which are part of the object model being compared) I am able to get a more coherent output from the tests. Listing 7.20 shows the two implementations of the ToString() methods in the classes under test and the resulting test output.

Listing 7.20 Implementing ToString() in compared classes so that test output is clearer.

```
    ///Overriding ToString inside The AnalyzedOutput Object//////////////
            public override string ToString()
            {
                StringBuilder sb = new StringBuilder();
                foreach (LineInfo line in lines)
                {
                    sb.Append(line.ToString());
                }
                return sb.ToString();
            }

    ///Overriding ToString inside each LineInfo Object//////////////
            public override string ToString()
            {
                StringBuilder sb = new StringBuilder();
                for (int i = 0; i < this.fields.Length; i++)
                {
                    sb.Append(this[i]);
                    sb.Append(",");
                }
                return sb.ToString();
            }

    ///TEST OUTPUT//////////////
    ------ Test started: Assembly: er.dll ------

    TestCase
    'AOUT.CH7.LogAn.Tests.MultipleAsserts.Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseF
    ields2'
    failed:
      Expected: <10:05,Open,Roy,>
      But was:  <>
        C:\GlobalShare\InSync\Book\Code\ARtOfUniTesting\LogAn.Tests\MultipleAsserts.cs(41,0): at
    AOUT.CH7.LogAn.Tests.MultipleAsserts.Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFi
    elds2()
```

Now the test output is much clearer and we can understand that we got very different objects. If the output was not clear enough, understanding why the test fails and maintaining it would be much harder.

Let's look at another way tests can become hard to maintain: making them too fragile by over specification.

7.2.8 Overspecification in tests

An overspecified test is a test that contains assumptions over how a specific unit under test *should implement* its behavior, instead of only checking that the end behavior is correct[7].

Here are some patterns of over specifying unit tests:

- A test that specifies purely internal behavior of an object under test
- A test that uses mocks when using stubs should be enough
- A test that assumes specific order or exact string matches when it is not required

Let's look at some examples of overspecified tests.

SPECIFYING PURELY INTERNAL BEHAVIOR

Listing 7.21 shows a test against LogAnalyzer's `Initialize()` method that tests internal state, and no outside functionality.

Listing 7.21: An overspecified test that tests a purely internal behavior, not the external behavior of a unit.

```
        [Test]
        public void Initialize_WhenCalled_SetsDefaultDelimiterIsTabDelimiter()
        {
            LogAnalyzer log = new LogAnalyzer();

            Assert.AreEqual(null,log.GetInternalDefaultDelimiter());
            log.Initialize();
            Assert.AreEqual('\t', log.GetInternalDefaultDelimiter());
        }
```

Why is the test in listing 8.16 overspecified? First, it only tests the internal state of the LogAnalyzer object. Because it is internal, it could change later on. What unit tests should be testing is the *public contract* and public functionality of an object. The tested code is not part of any public contract or interface (and in fact, it was exposed specifically for the purposes of the test – which could be considered a code smell).

[7] Also see the section titled "Overspecified software" in (Meszaros 2007)

USING MOCKS INSTEAD OF STUBS(OR MOCK OVER-USE)

The idea of using mocks instead of stubs is a very common mistake. In the next example, we use mocks to assert the “interaction” between our log analyzer and a provider it uses to read a text file.
We are wrongly checking that it calls the provider correctly to read the file’s text (an implementation detail which could change later on and break our test).
Instead, we could use a stub to return the fake results from the text file, and assert against the public output of the LogAnalyzer’s method, which makes for a more robust, less brittle test.

Listing 7.22 shows the method we want to test, followed by an overspecified test for that code.

Listing 7.22 An overspecified test that uses mocks when stubs would do just fine.

```
       public AnalyzeResults AnalyzeFile(string fileName)
            {
                int lineCount = logReader.GetLineCount();
                string text = "";
                for (int i = 0; i < lineCount; i++)
                {
                    text += logReader.GetText(fileName, i, i);
                }
                return new AnalyzeResults(text);
            }
    /////////////////////////the test////////////////
            [Test]
            public void AnalyzeFile_FileWith3Lines_CallsLogProvider3Times()
            {
                MockRepository mocks = new MockRepository();
                ILogProvider mockLog = mocks.CreateMock<ILogProvider>();
                LogAnalyzer log = new LogAnalyzer(mockLog);
                using(mocks.Record())
                {
                    mockLog.GetLineCount();
                    LastCall.Return(3);

                    mockLog.GetText("someFile.txt", 1, 1);
                    LastCall.Return("a");

                    mockLog.GetText("someFile.txt", 2, 2);
                    LastCall.Return("b");

                    mockLog.GetText("someFile.txt", 3, 3);
                    LastCall.Return("c");
                }
                AnalyzeResults results = log.AnalyzeFile("someFile.txt");
                mocks.VerifyAll();
            }
```

Why is the test in listing 8.17 overspecified? It seems to be testing the interaction between the interface of some *LogReader* (something that can read text files) and the LogAnalzyer object.

Unfortunately, we are testing the underlying reading algorithm inside the method under test, instead of testing for an expected output result from it.

To understand why this is problematic, think of a business manager. The business manager has an employee that trades stock for him. Eventually, we can measure trading stock by only one thing – do I have more money in the end than I had in the beginning? But if we were to follow the example in listing 8.17, it would be equivalent to the business manager saying he doesn’t care what the outcome of the trading actions are, as long as they seem smart on paper.

In reality, the manager should let the trader run his own internal algorithm, and only judge based on results, which is exactly what we’d like to do in our test. By doing that we make the test less brittle because we don’t care about “how” the code goes about analyzing the file, as long as it gives the expected results.

Listing 7.23 shows a modified test that only checks on the outcome of the operation without relying on its internal implementation:

Listing 7.23: Replacing mocks with stubs and checking outputs instead of interactions

```
     [Test]
```

```
public void AnalyzeFile_FileWith3Lines_CallsLogProvider3TimesLessBrittle()
{
    MockRepository mocks = new MockRepository();
    ILogProvider stubLog = mocks.Stub<ILogProvider>();
    using(mocks.Record())
    {
        SetupResult.For(stubLog.GetText("", 1, 1))
            .IgnoreArguments()
            .Repeat.Any()
            .Return("a");

        SetupResult.For(stubLog.GetLineCount()).Return(3);
    }
    using(mocks.Playback())
    {
        LogAnalyzer log = new LogAnalyzer(stubLog);
       AnalyzeResults results = log.AnalyzeFile("someFile.txt");

        Assert.That(results.Text,Is.EqualTo("aaa"));
    }
}
```

A NOTE ON REFACTORING FOR TESTABILITY CODE SMELLS:

When you refactor internal state to be visible to an outside test, could it be considered a code smell? It is not a code smell when you are refactoring to expose collaborators. It is a code smell if you are refactoring and there are no collaborators (so you don't need to stub or mock anything)

ASSERT.THAT

I am using NUnit's Assert.That syntax instead of Assert.AreEqual, because the fluent nature of the new syntax is much cleaner and nicer to work with. Give it a try!

Here's another way developers tend to make their tests over specified: over-assumptions.

Assuming order or exact match when it is not needed

Another common pattern people tend to repeat is to have asserts against hard coded strings in the unit's return value or properties, when in fact only a specific part of a string is necessary. The main rule should be "*can I use string.Contains() rather than string.Equals()?*" . The same goes for collections and lists. It is much better to make sure a collection contains an expected item than to assert that not only it contains it, but that the item is in a specific place in a collection (unless that is specifically part of what is expected). By doing these kinds of small adjustments, you can guarantee that implementation of the strings or collections' order may change, but as long as they *contain* what is expected, the test will pass and you won't have to go back and change it on every little space character you add to a string.

In the next section, we cover the third and final pillar of good unit tests: Readability. Together with maintainability and trust-worthiness, you can be sure to have all your bases covered.

7.3 Writing readable tests

Readability is so important that without it, the tests we write are almost meaningless. From the names of the tests, to having good assert messages, readability is the connecting thread between the person who wrote the test, to the poor soul who has to read it a few months later.

Tests are stories we tell each generation of new programmers on a project. They are the past, present and the future of the project's code, and like rings in a tree trunk, they allow a developer to see exactly what an application is made of and where it started.

This chapter is all about making sure the developers who come after you in the project will be able to maintain the production code and the tests that your write while understanding what they are doing and where they should be doing it.

There are several facets to readability, and this chapter focuses on the following:

- Naming standards for unit tests

- Naming variables
- Separating Assert from action
- Good assert messages

Let's go through these one by one.

7.3.1 Naming standards for unit tests

Naming standards are important because they give us comfortable rules and templates that signal what we should explain what we want to express in the test. The naming has three parts:

- The name of the method being tested
- The scenario under which it is being tested
- The expected behavior when the scenario is invoked

Removing even one of these parts from a test name can cause the reader of the test to wonder what exactly is going on, and to start reading the test code. To make it clear, our main goal is to release the reader of the test from the burden of reading the test code in order to understand what the test is trying to do.

The name of the tested method is essential so that you can easily see where the tested logic is. Having this as the first part of the test name also allows easy navigation and as-you-type intellisense(if your IDE supports it) in the test class.

The scenario under which it is being tested – this part gives us the "with" part of the sentence ("when I call method X **[with]** a null value…").

The Expected behavior – this part specifies in plain English what the method should do, return, or how it should behave based on the current scenario ("when I call method X with a null value **[then it should]** …")

A common way to write these three parts is to separate them by underlines like this:

```
    Public void MethodUnderTest_Scenario_Behavior()
{
}
```

As an example, here is a test , in listing 9.1 that uses this same naming convention:

Listing 7.24: A test with three parts in its name is easy to understand without reading its code.

```
[Test]
public void AnalyzeFile_FileWith3LinesAndGivenFileProvider_ReadsFileUsingProvider()
{
   //...
}
```

The method in listing 9.1 tests the method AnalyzeFile, giving it a file with 3 lines and a file reading provider, and expects it to use the provider to read the file.

If developers stick to this naming convention it will be easy for other developers to jump in and understand tests.

7.3.2 Naming variables

Variable naming in unit tests is just as important as, or even more important than, the run of the mill production code variable naming convention. The main reason is that tests apart from their chief function also serve as a form of *documentation* for an API. By giving variables good names we can make sure that people reading our tests understand *what* we're trying to prove (as opposed to "what we're trying to *accomplish*" when writing production code) as quickly as possible.

Listing 7.25 shows an example of an unreadable test. Can you tell why it's unreadable? (No looking at the listing text!)

Listing 7.25: An unreadable test. What does -100 mean? Did you peek??

```
[Test]
public void BadlyNamedTest()
{
```

```
            LogAnalyzer log = new LogAnalyzer();
            int result= log.GetLineCount("abc.txt");
            Assert.AreEqual(-100,result);
        }
```

In this particular instance, the API returns some “magic number” (a number that represents some magic value the developer needs to know). Because we don’t have a descriptive name for what the number is expected to be, we can only *assume* what it is supposed to mean. The test *name* should have helped us a little bit here, but the test name, as many test names are, needs more work, to put it mildly.

Is –100 some sort of exception? Is it a valid return value in a parallel universe?

This is where we have a choice:

- We can detect a design smell in the test and perhaps change the design of the API to perhaps throw an exception instead of returning -100 (assuming -100 is some sort of an illegal result value)
- We can compare the result to some sort of constant or aptly named variable as shown in listing 7.26:

Listing 7.26: a more readable version of the test with a const to describe intent.

```
        [Test]
        public void BadlyNamedTest()
        {
            LogAnalyzer log = new LogAnalyzer();
            int result= log.GetLineCount("abc.txt");
            const int COULD_NOT_READ_FILE = -100;
            Assert.AreEqual(COULD_NOT_READ_FILE,result);
        }
```

The code in listing 7.26 is much better because you can easily understand what the intent of the return value is.

The last part of a test is usually the assert. If the assert fails, the first thing the user will see is the assert message. The next section deals with making the most out of the assert message, if you’re going to use it.

7.3.3 Asserting yourself with meaning

Writing a good assert message is much like writing a good exception message – it is deceptively easy to get it wrong without realizing it, and for the people who have to read it, it makes a world of difference (and time).

There are several key points to remember when writing a message for an assert clause:

- Don’t repeat what the built in test framework outputs to the console.
- Don’t repeat what the test name explains.
- If you don’t have anything good to say, don’t say anything.
- Write what should have happened or what failed to happen.
- Possibly add when it should have happened.

Listing 7.27 shows a bad example of an assert message:

Listing 7.27: A bad assert message that repeats what the test framework writes by default.

```
        [Test]
        public void BadAssertMessage()
        {
            LogAnalyzer log = new LogAnalyzer();
            int result= log.GetLineCount("abc.txt");
            const int COULD_NOT_READ_FILE = -100;
            Assert.AreEqual(COULD_NOT_READ_FILE,result,"result was {0} instead of
{1}",result,COULD_NOT_READ_FILE);
        }


//Running this would produce:
TestCase 'AOUT.CH7.LogAn.Tests.Readable.BadAssertMessage'
failed:
  result was -1 instead of -100
  Expected: -100
  But was:  -1
```

```
    C:\GlobalShare\InSync\Book\Code\ARtOfUniTesting\LogAn.Tests\Readable.cs(23,0): at
AOUT.CH7.LogAn.Tests.Readable.BadAssertMessage()
```

As you can tell if you read the output written just below the test code, there is a message that repeats twice. Our assert message didn't add anything to the conversation, except more words to read. Here are a couple of better alternatives for that assert message:

- Don't write anything but instead have a better named test.
- "Calling GetLineCount() for a non existing file should have returned a COULD_NOT_READ_FILE"

Now that the assert message is understandable, it's time to make sure that the assert happens on a different line than the method call. To understand why, read the next section.

7.3.4 Separate Assert from Action

This is going to be a short section, but an important one nonetheless. The main point: try to avoid writing the assert line and the method call in the same statement. Here it is, in the plainest terms I can come up with in listing 7.28:

Listing 7.28: Separating the assert from the thing asserted on is crucial to readability.

DO:

```
        [Test]
        public void BadAssertMessage()
        {
           //some code here
            int result= log.GetLineCount("abc.txt");
            Assert.AreEqual(COULD_NOT_READ_FILE,result);
        }
```

DO NOT:

```
        [Test]
        public void BadAssertMessage()
        {
           //some code here
            Assert.AreEqual(COULD_NOT_READ_FILE,log.GetLinweCount("abc.txt"));
        }
```

Get the difference? The second example is much harder to read and understand in the context of a real test, because the actual call toe the GetLineCount() method is inside the call to the assert message.

7.3.5 Setting up and tearing down

Setup and teardown methods in unit tests can be abused to the point of almost no readability in the tests or in the setup and teardown methods. Usually the situation in the Setup method is worse than teardown.

For a full discussion on abusing setup methods read section 8.2.3 ([Setup] methods are not always a good idea) in chapter 8.

The next section details one possible abuse.

SETTING UP MOCK OBJECTS IN THE SETUP METHOD

If you have mocks and stubs being set up in a setup method, that means they *don't* get set up in the actual test. That, in turn, means that whoever is reading your test may not even realize that there are mock objects in use or what the expectations are from them in the test.

DO YOU EVEN *NEED* A SETUP METHOD?

I've found myself at several instances writing full test classes that didn't even have a setup method, only helper methods being called from each test for the sake of maintainability. The class was still readable *and* maintainable.

It is much more readable to initialize the mock objects directly in the test itself with all their expectations. If you are worried about readability, you can refactor the creation of the mocks into a helper method, which each test calls. That way whoever is reading the test know exactly what is being setup instead of having to look in multiple places.

7.4 Summary

Tests that you can trust are something most developers don't achieve easily when they first start out writing unit tests. It takes some discipline as well as some imaginative ways to make sure you're doing the right thing. A test that you can trust can be an elusive beast to capture at first, but when you get it, you'll feel the difference immediately.

Some of the ways to achieve this kind of trustworthiness has to do with keeping our good tests alive and our bad tests removed or refactored away, which a lot of what this chapter was about. The rest was about what may be a problem inside a test, such as logic, testing multiple things, ease of run ability and more. Putting all these things together can be quite an art form.

If there's one thing to take away from this chapter, it is this: tests grow and change with the system under tests.

The idea of writing tests that are *maintainable* has not been covered much by mainstream unit testing and TDD related literature, but at the writing of this book, is just starting to appear online in many blogs and forums. I believe that this is the next step in the *evolution* of unit testing techniques. The first step of *acquiring the initial knowledge* (how do you write a unit test? What is a unit test?) has been covered in many places. The second stage is the one that happens *after* initial adoption (which is where we are today) and involves *refining the techniques* to improve all the aspects of the code that we write, and looking into other variables other than just writing a test (such as maintainability, readability and such). It is this critical step that this chapter focuses on (and most of this book).

In the end it's simple: Readability goes hand in hand with maintainability and trustworthiness. People who can read your tests can understand them and maintain them, and will also trust them when they pass. When all these conditions are met, your team is ready to handle change, and change the code when it needs to change, knowing when things break.

In the next chapters, we'll take a broader look at what makes code testable, how to design for testability and how to refactor existing code into a testable state.

8
Integrating unit testing into the organization

As a consultant, I had the distinct pleasure of helping several companies, big and small, to integrate Test Driven Development and Unit Testing as part of their organizational culture. Some have failed. Those who have succeeded had several things in common. This chapter combines many of the stories from both camps.

As with any type of change in an organization, changing people's habits is more psychological than it is technical. People don't like change, and change is usually accompanied with plenty of FUD *(Fear, Uncertainty, Doubt)* to go around. If you plan to introduce this kind of change into your organization, there are some things you need to consider no matter where you work.

This chapter is made of four main parts:

- *Becoming the agent of change* – Initial steps before you start mixing things up
- *Ways to succeed* – from my experience, things contributed to success in making changes to a process
- *Ways to fail* – also from my experience, things which can destroy what you are trying to do
- *Tough questions and answers* – most frequently asked questions when introducing unit testing to a team.

The first thing you need to realize is that this won't be a walk in the park for most people, as you'll see in the next section.

8.1 Steps to becoming an agent of change

If you are going to be the "agent of change" in your organization, you should first accept that role upon yourself. Whether you plan to or not, people will view *you* as the person responsible for what's happening and there is no use in hiding behind anything. In fact, hiding can help make sure that things go awfully wrong.

As you start to implement things, people will start asking the *tough questions* that they care about. How much time will this "waste"? What does this mean for me as a QA? How do we know it works? Be prepared to answer these. The end of this chapter features some answers to the most common questions.

You'll find *that convincing other people inside the organization* before you start your move will help you immensely when there comes a time to make tough decisions and answer those questions. Finally, *someone will have to stay at the helm*, making sure the change doesn't "die" for lack of momentum. That's *you*. There are ways to keep things alive, as you'll see in the next sections.

8.1.1 Be prepared for tough questions

Do the research and read up on the answers at the end of this chapter as well as the related resources. Read forums, mailing lists and blogs, and consult with peers. If you can answer *your* own tough questions there is a bigger chance you can answer someone else's tough questions.

8.1.2 Convince insiders: champions and blockers

Loneliness is a terrible thing. Not many things make you feel more alone in an organization than going against the current. If you're the only one who thinks what you're doing is a good idea, there is little cause for anyone to actually try and implement what you are advocating.

Champions

As you start pushing for change, identify the people that you think are most likely to help you in your quest. They will be your champions. They are the ones who are usually early adopters, or people who have an open enough mind to try some of the things you are advocating. They may already be half convinced but are looking for an excuse to start the move. They may have even tried it once and failed on their own.

As you start the move, approach them before anyone else and ask for their opinions on what you're about to do. You may find that they will tell you some things that could help you but you have not considered: teams that might be good candidates to start with or places where people are more accepting for such changes. They may even tell you what to watch out for from their own personal experience.

By approaching them, you are helping to ensure they are part of the process. People who feel "part" of the process will usually try to help make it work. Make them your champions: Ask them if they can help you and be the ones that people can come to and ask questions. Prepare them for such events.

Blockers

Next, you'll have to identify "blockers". Those people in the organization are most likely to resist the changes you're making for some reason. For example, a manager might object to adding unit tests claiming that they will add too much time to the development effort, and increase the amount of code that needs to be maintained. The answer to this and many other questions is in the last part of this chapter (8.4) .The main idea is to make them part of the process instead of resistors to it, by having them take an active role in the process (those who are willing and able to).

The reasons why people would resist the change vary. You will find some helpful hints that people could resist the process by looking at some of the questions and answers at the end of this chapter. Some are afraid for their jobs, and some are afraid of change by default (they feel comfortable with the way things are).

Going to people and detailing all the things they could have done better is often a non-constructive activity, as I've found out the hard way. People do *not* like to be told what they do not do well.

Ask those people to help you in the process by being in charge of defining coding standards for unit tests, for example, or by doing code and test reviews to peers every other day. You will have given them a new responsibility that will help them feel that they are still relevant in the organization and that they are relied upon. Have them help be part of the team that chooses the course materials or the outside consultants. Something. Anything. They *need* to be part of this or they will almost certainly take part in a mini *rebellion* against this change.

8.1.3 Identify possible entry points

Identify where in the organization you can start implementing the process. Most successful implementations have taken a very steady route: start with a "pilot" project in a small team and see what happens. If all goes well, move on to other teams and other projects.

Smaller and Simpler

Identifying possible teams to start with is usually an easy matter. It is usually a smaller team compared to the others, working on a lower profile project with low risks. As long as the risk is minimal, it is easier to convince people to try it. One caveat is that the team needs to have members who are open-minded to changing the way they work and learning new skills. Ironically, more often than not, the people with less experience on a team are most likely to be open to change, and people with more experience tend to be more entrenched in their way of doing things. If you can find a team with an experienced lead who's open to change, that also has less experienced developers, there is a big chance that team will offer less resistance. You can go to the team and ask them their opinion directly on holding a pilot such as this. They will tell you exactly if this is the right place for you or not.

Sub team and feature

Another possible candidate for a team would be to form a *sub team* in an existing team working on some product *feature* in an existing project. A couple of places I've been to were inclined to find a project with a very problematic codebase, focusing on one of the components in that project and the features associated with it. It's easy to find "black sheep" components in teams. Almost every team will have its own little "black hole" component that they need to maintain, one that does many things but also has many bugs. Adding features for such a component is a tough task and sometimes it is that kind of pain point that drives people to try a pilot projects for change.

MAKE SURE IT IS FEASIBLE.

For a pilot project make sure you're not biting off more than you can chew. Sometimes it takes more experience to do more difficult projects so you might want to have at least two options: One for a complicated project and one for an easier project, so that you can pick and choose between them.

Now that you are mentally prepared for the task at hand, it's time to look at some of the things you can do to make sure things go smoothly (or at all).

8.2 Ways to succeed

There are two main ways an organization or team start changing a process. Bottom-up or top-down. The two ways are *very* different in nature, as you'll see, and each of them could be the right way for your team or company. There is no one right way.

8.2.1 Guerilla implementation (bottom up)

Guerilla style implementation is all about starting out with a team, getting results, and only then convincing other people that the practices are worthwhile. Usually the drivers for guerilla implementation are the actual team that is tired of doing things the "prescribed" way and so they set out to do things differently. They will study things on their own and will make things happen.

Later when the team shows results, other people in the organization may decide to start implementing things on their own teams.

Some people refer to guerilla style implementation as a process adopted first by developers and only then by management. Other people see it as a process *advocated* first by developers and only then by management. The difference is that you can accomplish the first in a covert manner without the higher powers knowing about it. The latter is done in conjunction with management.

It is up to you to figure out which will work best for you. This may sound harsh, but sometimes the only way to change things it by covert operations. Avoid it if you can but if there is no other way and you're really sure the change is needed, nothing is stopping you from implementing things in your own team the Nike way: "Just Do It".

Don't take this as a recommendation for a career limiting move. Developers do things they didn't ask "permission" for all the time. No one gives permission to a developer to Debug their code 5 times more than the other developer next to them. The same for reading email, writing code comments creating flow diagrams and so on. These are all tasks developers do between themselves. The same goes for unit testing. Most developers already write tests of some sort (automated or not). The idea is to redirect those loads of time into something that will also benefit in the long term.

8.2.2 Convincing management (top down)

This type of move usually starts in one of two ways: A manager or a developer will start the process and trigger the rest of the organization to move in that direction, piece by piece.

THE MANAGER

A midlevel manager somewhere in the organization may see a presentation somewhere, read a book (such as this one), or talk to a colleague about the benefits of specific changes to the way they work. Such managers would usually make some sort of initiation—give a presentation to other people in other teams, or even use their authority to make the change happen.

8.2.3 Get an outside champion

I'd highly recommend to get an outside person to help with the change. An outside consultant coming in to help with unit testing and related matters has some advantages over someone who works in the company. First, he can say things that people inside the company may not be willing to hear from someone who works inside ("the code quality is bad", "your tests are unreadable"). Second, they will have more experience with dealing with resistance from the inside, coming up with good answers to tough questions, and knowing which buttons to push to get things going. Another advantage is the fact that *this is their job.* Unlike other employees in the company, who have "better" things to do than to push change (like, ehm, writing software), the consultant does this full time and is dedicated to this purpose. I've often seen how a change breaks down simply due to an overworked champion who does not have the time to dedicate to the process.

8.2.4 Big Visible Progress

It is always important to keep the progress and status of the change visible. Hang whiteboards, or A3 or A2 posters up on walls in corridors on in the food related areas where people congregate. The data should be related to the goals you are trying to achieve. For example, show the number of passing or failing tests in the last nightly build. Keep a chart of which teams are already running an automated build process. Put up a Scrum burn down chart of iteration progress (as seen in figure 8.1) if that's what your team is practicing. Put up contact details of all the champions and yourself and how to catch them if any question arises.

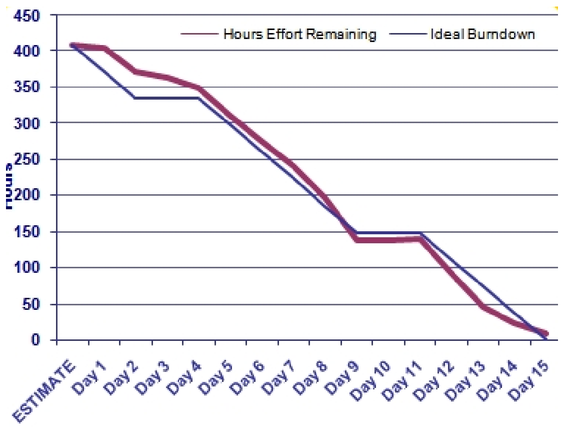


Figure 8.1: An example of a Scrum Burn down chart

You are aiming to "talk" to two groups with these charts: The first group is the one that is undergoing the change. They will get a better feeling of accomplishment and pride as the charts (that are open to everyone) are updated. They will also feel more compelled to do the process since it is more visible to others. They will also be able to keep track of how they are doing compared to other groups. For example, they may push harder knowing that another group has gone faster with implementation of specific practices.

The other group you're aiming for is those in the organization who are not part of the process. You are raising their interest and curiosity, triggering conversations and buzz, and creating a "current" that they can join if they choose to.

8.2.5 Aim for specific goals

Without goals, the change will be hard to measure, hard to communicate to others, and generally be a vague "something" that can be easily shut down at the first sight of trouble.

Possible goals you might want to consider:

- Increase amount of test code coverage

  A study by Boris Beizer showed that developers who write tests and don't use code coverage tools or other techniques to test code coverage will be naively optimistic about the actual coverage they gained from the tests (Johnson 1994). Another study suggests that testing without code coverage tools may only cover about 50-60% of the code (Wiegers 2002) . (There are many anecdotal evidence that using TDD one can get up to 95-100% code coverage for logical code)

  A very simple goal to measure is the percentage of the code covered by our tests. The more coverage the better chance we have of finding bugs. It's not a silver bullet, though. One could easily have close to 100% code coverage with tests so bad that they don't mean anything. A *low* coverage is a sign. *High* coverage is a *possible* sign that things are better.

- Increase amount of test code coverage relative to the amount of code churn.

  Some production systems will allow you to measure the amount of "code churn" – how much code was changed between builds (lines of code). The less lines of code changed, the fewer bugs you're supposed to introduce into a system. This isn't always practical, especially in systems where you do a lot of code generation as part of the build process. However, this also be solved by ignoring those places in the codebase. One of the systems that allows to measure code churn is Microsoft Team System [8]

- Reduce amount of bug-re opening

  Bugs that were closed and need to be reopened are usually easier to follow in systems with a low amount of tests. It's very easy to fix something and then mistakenly break something else. The less this happens it's a sign that we are able to fix things and maintain the system without breaking previous assumptions.

- Reduce average bug fixing time (time from bug opened to bug closed)

  A system with good test and coverage will usually allow you to fix things more quickly (assuming the tests are written in a maintainable manner). That in turn means better turnaround times and release cycles that are less stressed.

- In his book *Code Complete* (2nd Edition) Steve McConnell outlines several metrics you can use to test progress (McConnell 2004) . Assuming you can collect such numbers from your code, they include number of defects found per class, by priority, number of defects per routine number of testing hours per bug found, avg. number of defects per Testcase and more. I highly recommend reading chapter 22 of that book, which deals with Developer Testing.

8.2.6 Realize that there will be hurdles

There always are. Most of the hurdles will be from within the organizational structure. Some of them will be technical. The technical ones are easier to fix, since it's just a matter of finding the right solution. The organizational ones need care and attention and a psychological eye for the dev guy.

It's important not to back step yourself when something goes wrong. An iteration goes bad, tests go way slower than expected and so on. It is sometimes hard to get going, but you will need to persist with the efforts for at least a couple of months or more in order to start feeling comfortable in the new process and iron out all the kinks. Have management commit to continuing for at least 3 months even if things don't go as planned. It's important to have their backs up front. You don't want to run around trying to convince people in the middle of a stressful first month.

Now that we've seen what *can go right*, here are things which can lead to failure, that I've personally seen or experienced in companies I've worked with.

8.3 Ways to fail

In the beginning of chapter 1 I told you about one project I was involved with that failed, partly due to unit testing not being implemented correctly. That's just one way you can fail a project. I've listed the others here along with the one that cost me that project, along with things that can be done about it.

8.3.1 Lack of driving force

Of all the places where I have seen change fail, the lack of a driving force was the most consistent in making sure this happened. Consistency has its price – it will take time away from your "normal" job and you need to be willing to let it go of find a way to make it work, or it won't. Having an outside person as mentioned in 8.2.3 will help you in your quest for consistency.

8.3.2 Lack of political support

If your boss explicitly tells you to "not do it" then there isn't a whole lot you can do, besides trying to convince them to see what you see. Sometimes it's much more subtle than that, and the trick is to realize that you are facing an opposition. For example, you may get a "sure, go ahead and implement those tests. We are adding 10%

[8] Find out more at http://tinyurl.com/4fco7q

to your time to do this". Hint – 10% is not realistic for beginning a unit testing effort. Anything below 30% is a wishful thinking.

So that is one way of a manager to "break" a trend by "chocking" it out of existence. Again, you'll have to realize this is what you are facing. It's easy to realize. When you tell them that it's not realistic and you get back a "so don't do it".

8.3.3 Bad implementation and first impressions

If you're planning to implement unit testing without prior knowledge, do yourself one big favor: start doing it with someone who has experience, as well as use some best practices (such as those outlines in this book). I've been to that place where you just jump into the deep water without proper understanding of what to do or where to start, and it's not a good place to be. Not only will it take you a *huge* amount of time to get up to speed with how to do things that are acceptable for your situation, you'll also lose a lot of credibility along the way for starting out with a bad implementation of things, which can eventually lead to the shutting down of the pilot project.

If you go back to the first page of this book, this is what happened to me on that project, as I depicted it. You only have a couple of months to get things up to speed and convince the higher ups that you're not screwing everything up. Make them count, and remove any type of risk that you can get rid of. Don't know how to write good tests? Read a book. Get a consultant. Don't know how to make your code testable? Do the same. Don't go over paths other people have been to if you don't have to.

8.3.4 Lack of team support

If your team does not support your efforts, it will be near impossible to succeed, since you will have a hard time consolidating your extra work on the new efforts with "real" work. You should strive to have your team be part of the new process, or at least not stop it. Getting team members one by one is a good way to start. Sometimes talking to them as a group about your efforts and answering hard questions can prove valuable. Whatever you do, don't take the team's support for granted. Make sure you know what you're getting into; these are the people you have to work with on a daily basis.

The next section has answers to questions I wished someone gave me when I first started to do unit testing. It can help you as you come across them or if they are being asked by people who can make or break your change agenda.

8.4 Tough questions and answers

Here are some questions I've come across in various places. They usually hold underneath them the premise that implementing unit testing can hurt someone personally—a manager looking out for his deadlines, a QA looking out for her relevancy. Once you realize where the question is coming from, it's important to address that issue (directly or indirectly). Otherwise, there will always be subtle resistance.

8.4.1 How much time will this add to the current process?

Let's begin with some facts. Studies have shown that raising the code quality overall in the project can increase productivity and shorten schedules (Capers, Programming Productivity 1986) (Capers, Software assesments, benchmarks and best practices 2000). How does this sit with the fact that writing tests can take longer to code? Maintainability and ease of fixing bugs, mostly.

This question is usually asked by the people who are in the front lines in terms of timing. Team leads, project managers and clients. There is a difference in views here. A team lead may really be asking "so what should I tell my project manager when we go way past our due date?" but they may think the process is useful, they are just looking for ammo in the upcoming uphill battle. They may also be asking the question not in terms of the whole product, but in terms of specific feature sets or functionality. A project manager or customer, on the other hand, will usually be talking in terms of full product releases.

The fact that different people may care about different scopes is relevant because the answer for specific scopes may be different. For example, unit testing can even double the time it takes to implement a specific feature in the next month, but the overall bottom line release date for the product in the next 5 months may actually be sooner. To understand this, let's look at a real example from a company I was involved with.

A TALE OF TWO FEATURES

One of the larger companies I was consulting with was starting to implement the idea of unit testing into their process. Before they went all in they went on a pilot project. The pilot project consisted of a group of developer tasked with adding a new feature to an existing large application. The company's main livelihood was in creating this large application and then customizing parts of it for various clients. It was a billing style application and the company had thousands of developers around the world.

The following measures were taken to test the success of the pilot:

- Test the time it took to go through each of the development stages for the team
- Test the overall time it took for the project to be released to the client
- Test the amount of bugs found by the client after release

The pilot was measured against a similar feature made by a different team for the same client, with almost the exact same feature size (these were both customization efforts of the product for two different clients – one was with unit tests, the other was without).

Table 8.1 shows the differences in time:

Table 8.1: Team Progress and output measured with and without tests

Stage	Team without tests	Team with tests
Implementation (coding)	7 days	14 days
Integration	7 days	2 days
Testing + bug fixing	Testing – 3d + Fixing - 3d + Testing – 3d + Fixing - 2d + Testing – 1d = 12 days	Testing – 3d + Fixing - 1d + Testing – 1d + Fixing - 1d + Testing – 1d = 8 days
Overall release time	26 days	24 days
Bugs found in production	11	71

Overall, the time to release the same feature with tests was less than it was without tests. The teams were roughly at the same skill and experience level and the features were roughly the same to implement.

Still, the managers on the team with the unit tests didn't initially believe the pilot would be a success. The reason is that they only looked at the *first row* as the criteria for success in the project instead of the bottom line. In the first line, it takes *twice* the amount of time to code the same feature (unit tests cause you to write more code, naturally. It doesn't matter that the time "wasted" more than made up for itself when the QA team got a hold of the product and found less bugs to deal with.

That's why it's important to emphasize the unit testing can increase the amount of time it takes to implement a feature but that time balances out based on quality and maintainability over the product's release cycle.

8.4.2 Will my QA job be at risk because of this?

No. As a QA you will first be able to receive the application with full unit test suites, so that you can make sure all the unit tests pass before you start your own testing process. Having unit tests in place will actually make your job more interesting. Instead of doing "UI Debugging" (where every second button click results in an exception of some sort) your job will be able to focus on finding more logical (applicative) bugs in real world scenarios. Unit tests provide the first layer of defense against bugs. QA provides the second layer, the user's acceptance layer.

Just like security, you always need to have more than one layer of protection. Having a QA that can focus on the larger issues can produce better applications.

Another possible aspect I've seen is QAs who can code, who help write unit tests for the application. That happens in conjunction with the work of the application developers and not *instead* of them writing tests.

Also see 8.4.5 for additional QA related questions.

8.4.3 How do we know this is actually working?

Create a metric of some sort. See section 8.2.5. If you can measure it, you'll have a way to know. Plus, you'll *feel* it.

Figure 8.2 shows a sample test code coverage report (coverage per build). Adding a report like this by running tools like NCover for .NET automatically during the build process can help see progress in one aspect of development.

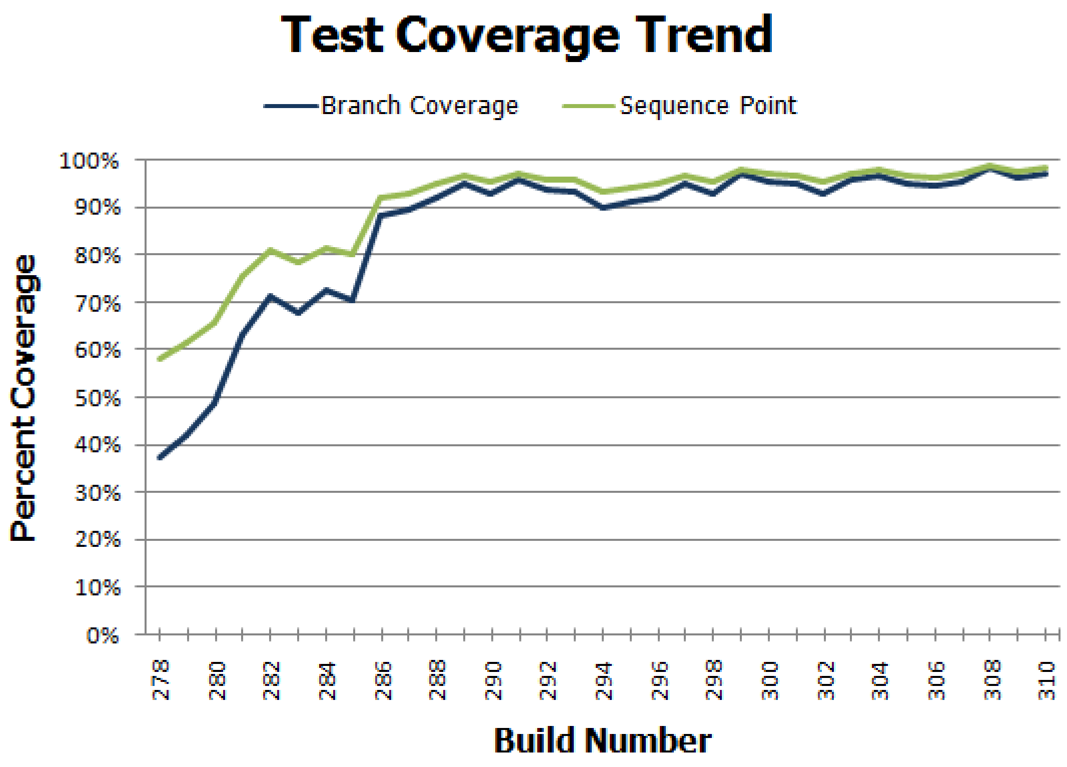


Figure 8.2: An example test code coverage trend report

Coverage is a good starting point if you're looking for signs you're missing plenty of unit tests (getting coverage of 5% on your code is a sure sign of that.

8.4.4 Is there proof that unit testing helps?

There aren't any specific studies that have been done. Most studies that do exist talk about adopting agile specific methods, with unit testing being just one of them. There is empirical evidence all over the world of companies and colleagues having great results, never wanting to go back to a codebase without tests. [TODO: Quote studies]

8.4.5 How come QA is still finding bugs?

A QA's job is to find bugs at many different levels, attaching the application from many different corners. Usually a QA will perform integration style testing, which can find problems that unit tests cannot. For example, how different components work together in production may prove to have bugs even though they pass unit tests (they work great in isolation). In addition, a QA may test things in terms of use cases or full long scenarios that unit tests usually won't cover. That can lead to logical bugs or acceptance related bugs and is a great help to ensuring better project quality.

A study held by Glenford Myre showed that developers writing tests were not really looking for bugs, and so found only part (half to two thirds) of actual bugs in an application (A controlled experiment in program testing and code walkthroughs/inspections 1978). Broadly, that means you can count on it that QA will always have a job, not matter what.

8.4.6 We have lots of code without tests already. Where do we start?

A study conducted in the 70s and 80s showed that typically, 80% of the bugs are found in 20% of the code (Endres 1975) (Gremillion 1984) (Boehm n.d.) (Shull 2002). The trick is to find the code that has the most problems. More often than not any team can tell you without even looking which components are the most problematic. Start there. You can always add some metrics as listed in 8.2.5 relating to mount bugs per class.
Testing legacy code requires different starting rules than writing new code with tests. See *chapter 10: Testing legacy code* for more details.

8.4.7 We work in several languages. Is unit testing feasible in that situation?

Sometimes tests can be written in one language that still test things written in other languages, especially if this is a .NET mix of languages. You can write tests in C# that test code written in VB.NET for example. Sometimes each team that works using a different language will have to learn to write tests in that language. C# developers can write tests in C# using NUnit or MbUnit, and C++ developers can write tests using one of the C++ oriented frameworks such as CPPUnit. I've also seen solutions where people who wrote C++ code would write managed C++ wrappers around it and then write tests in C# against those managed C++ wrappers, so that things would be easier to write and maintain.

8.4.8 We have a combination of software and hardware. Is that feasible?

You need to write tests for the software part. Chances are you most likely already have some sort of hardware simulator. The tests you write can take advantage of this. It may be a little more work but it is definitely possible and companies do this all the time.

8.4.9 How do you know you don't have bugs in your tests?

You need to make sure your tests fail when they should, and pass when they should. Test Driven Development is a great way to make sure you don't forget to check those things. See chapter 1 for a short walkthrough of TDD.

8.4.10 I don't need to write tests. My code works fine and I can see so in my debugger.

That's great for you. However, what about other people's code? How do you know *it* works? How do they know *your* code works and that they haven't broken anything when they make changes? Remember that coding is just the first part in the life of the code. Most of its time, the code will spend in maintenance mode. You need to make sure it will tell people when it breaks, using unit tests.

A study held by Curtis, Cresner and Iscoe showed that most defects arrive into this world not by the code itself, but by the virtues of miscommunication between people, requirements that keep changing, and lack of application domain knowledge (Bill, Krasner and Iscoe 1988) . Even if you're the world's greatest coder, chances are if someone tells you to code the wrong thing, you'll do it. And when you need to change it because it was all wrong in the first place, you'll be glad you had tests for everything else to make sure you didn't break it.

8.4.11 Are we forced to do TDD style coding?

If the reason people are asking this is fear of "too much change" at once, learning can be broken up into several intermediate steps:

- Learn Unit Testing (from books such as this) and use tools such as Typemock Isolator or JMockit so that you won't have to worry about design aspects while testing.
- Learn good design techniques such as S.O.L.I.D (see Appendix I on this technique)
- Learn to do Test Driven Development (a good book is "Test Driven Development with Microsoft .NET" by Jim Newkirk)

This way the learning is easier and you can get started more quickly with less loss of time to the project.

In summary, the answer is no. You can write the tests after the code. TDD is a style choice, which many people find very productive and beneficial. Many other people find that writing the tests after the code is good enough for them. Choose according to what you see fit. I Personally see a lot of value in TDD, but I recognize that other people may find that just writing unit tests in a good first step.

8.5 Summary

Implementing unit testing in the organization is something that many readers of this book will have to face at one time or another. Be prepared. Make sure you have good answers to questions. Make sure that you are not alienating the people who can help you. Make sure you are personally ready for what could be an uphill battle.

In the next chapter, we take a look at one aspect of that battle: the idea of designing code for testability, and why it may be important for you. We'll also talk about why you may never need to do it.

8.6 Works Cited

A controlled experiment in program testing and code walkthroughs/inspections. Communications of the ACM 21 No.9 (September) 760-68, 1978.

Andy Hunt, Dave Thomas. The Pragmatic Programmer. Addison Wesley, 1999.

Bill, Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. Communications of the ACM 31, no.11 (November): 1268-87, 1988.

Boehm, Barry W. Industrial software metrics top 10 list. IEEE software no. 9 (September): 84-85.

Capers, Jones. Programming Productivity. New York, NY: McGraw-Hill, 1986.

—. Software assesments, benchmarks and best practices. Reading, MA: Adison Wesley, 2000.

Endres, Albert. An analysis of errors and their causes in system programs. IEEE transactions on software engineering no.2 (June): 140-49, 1975.

Gremillion, Lee L. Determinants of program repair maintenance requirements. Communications of the ACM 27 no. 8 (August): 826-32, 1984.

Johnson, Mark. *Dr. Boris Beizer on software testing: an interview part 1.* The software QA quarterly, Summer 41-45, 1994.

McConnell, Steve. *Code Complete 2nd Edition.* MS Press, 2004.

Meszaros, Gerard. *xUnit Test Patterns, Refactoring Test Code.* Addison Wesley, 2007.

Shull, et al. *What we have learned about fighting defects.* Proceedings, Metrics IEEE; 249-258, 2002.

Wiegers, Karl. *Oeer reviews in software: A practical guide.* Boston, MA.: Addison-Wesley, 2002.

9 Working with legacy code

A few years back I was consulting for a (very) large development shop that did Billing software. They had over 10,000 developers with .NET, Java and C++ mixed products, sub products and intertwined projects. The software had existed in one form or another for over 5 years and most developers were tasked with maintaining and building on top of existing functionality.

My job was to try to help several divisions (of all languages) to learn Test Driven Development techniques. For about 90% of the developers I worked with, this never became a reality. There were several reasons:

- Once I left it was really easy to go back to the “old” mode of work
- There was too much pressure to do other things than to devote time to this learning task
- It was proving really hard to write tests against existing code
- It was next to impossible to refactor the existing code (or not enough time to do it)
- The learning curve was too high for most people
- Some people didn’t want to change their design
- People did not see clear and quick results from doing this
- Tooling was getting in the way (or lack thereof)
- We didn’t know where to begin.

How do we approach existing code? What issues are there to solve? Where do we start? These are tough questions that most of us have to deal with. This chapter will try to tackle some of the problems associated with approaching legacy code bases to begin with, listing techniques, references and tools that might help.

9.1 What is the big problem?

As anyone who’s ever tried to add tests to an existing system, 99.9% of the time the system is almost impossible to write tests for, because it has been written without proper places in the software that allow extensions or replacements to existing components (*seams*).

The problems that need to be addressed are:

- There is so much work, where do I start? Where should I focus my efforts?
- How can I make the code more testable?
- How can safely I refactor my code if it has no tests to begin with?
- Resources

We begin at the start – where *do* you start?

9.2 Where do you start adding tests?

Assuming you have a bunch existing code inside components, you'll need to find a way to create a priority list of components where testing them would make the most sense. There are several factors to consider for each component, that can affect its priority. We can number these factors from 1 to 10 (where 10 means more).

- Amount of logic in the component (Logical Complexity)
- Amount of Dependencies in the component (Dependency Level)
- General Priority in the project

The most obvious way to approach testing legacy code components is to think of them in at least two ways:

- How much Logic do I have in there (The Logical Complexity (LO) factor)

  For example, nested ifs, switch cases or recursion. Tools for checking cyclomatic complexity can also be used for this purpose.

- How many dependencies do I have to break in order to bring this class under test?(Dependency Level-DL)

  Does it communicate with an outside Email Component perhaps? Does it call a Static "Log" method somewhere?

Table 9.1 shows a short example of a list of classes with these parameters. I call this "test feasibility table":

Table 9.1 A simple test feasibility table

Component	Logical Complexity	Dependency level	Priority	Notes
Utils	6	1	5	This utility class has little dependencies, but contains a lot of logic. It will be easy to test it, and it provides lots of value.
Person	2	1	1	This is just a data holder class. Little logic and no dependencies. Some (small) real value in testing this.
TextParser	8	4	6	This class has lots of logic and lots of dependencies. To top it off it is part of a high priority task in the project. Testing this will provide lots of value but will also be hard and time consuming to bring under test.
ConfigManager	1	6	1	This class holds configuration data and reads files from disk. It has little logic but many dependencies. Testing it may provide little value to the project and will also be hard and time consuming to do.

From the data in table 9.1, we can create the diagram shown in figure 9.1, which divides our components by amount of value to the project, and by amount of dependencies.

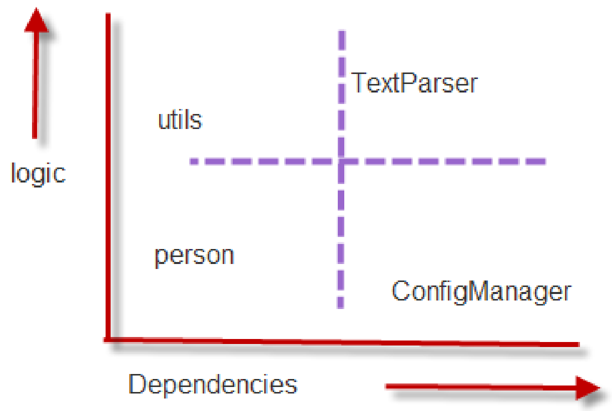


Figure 9.1 Mapping components to test feasibility

We can safely ignore items that are below our designated threshold of logic (I usually set it at 2 or 3) so Person and ConfigManager can be ignored from our list of candidates. We are left with only the top two components.

There are two basic ways to look at the data and decide based on the results what you'd like to test first.

- The more complex it is, and the *easier* it is to test, the more I'd like to test it; or,
- The more complex it is, and t*he harder* it is to test, the more I'd like to test it

That is, the items in the top left will be easier to test and provide value, and the items on the right will be harder to test but provide value, as figure 9.2 shows.

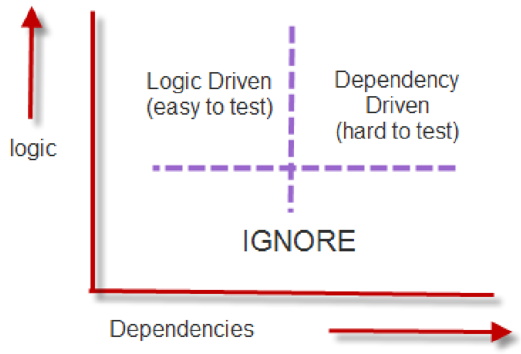


Figure 9.2 Easy, hard, and irrelevant component mapping based on logic and dependencies

The question now is what path do you take? Start with the easy stuff or the hard stuff?

9.3 Choosing a selection strategy for legacy components

As I have shown, there are two possible ways to go: start with the components that are easy to tests or start with the ones that are hard to test (many dependencies). Each strategy presents different challenges, pros and cons.

9.3.1 Pros and cons of "easy first"

Starting out with the components that have less dependencies will make writing the tests initially much quicker and easier. However, there's a catch, as figure 9.3 shows.

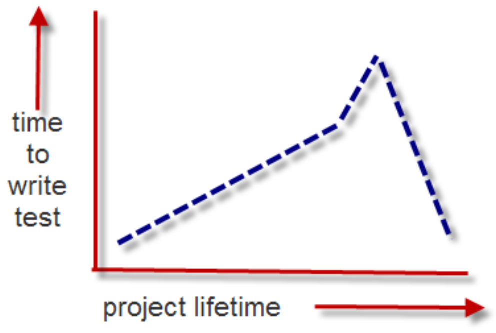


Figure 9.3 Time to test during the lifetime of the project when starting with "easy" components.

Figure 9.3 shows how long it takes to bring components under test during the lifetime of the project. Initially it is really easy to write tests, but as time goes by we are left with components that are increasingly harder and harder to test, with the really tough ones waiting got us at the end of the project cycle, just when everyone is stressed out in pushing a product out the door.

WHEN WOULD YOU WANT TO CHOOSE THE "EASY FIRST" STRATEGY?

If your team is relatively new to unit testing techniques, it is worthwhile to start with the easy stuff. As time goes by the team learns the techniques needed to deal with the more complex components and dependencies. Most likely, the really complex stuff will be delayed as your team learns its way around unit testing, leaving the more complex challenges to a different, more comfortable time. It may be wise to de-select all components over a specific threshold of dependencies (4 is a good start) for such a team.

9.3.2 Pros and cons of the "hard first" strategy

Starting with the "hard" components may seem like a losing proposition to begin with, but it has an upside as well, as long as your team has experience with unit testing techniques.

Figure 9.4 shows the time to test a component over the lifetime of the project, if you start testing the components with the most dependencies first.

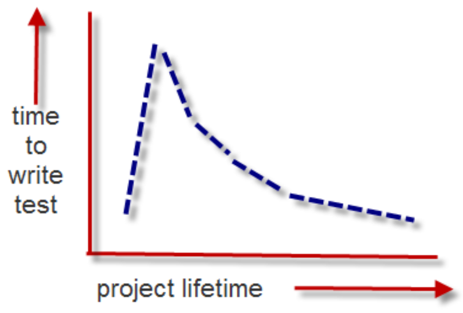


Figure 9.4 Time to test over the lifetime of the project with "hard first" strategy.

With this strategy, you could be spending a day or more to get even the simplest tests going on the more complex components (tools like Typemock Isolator for .NET and JMockit for Java can ease the burden of legacy code testing significantly, as I discuss at the end of this chapter).

Notice the quick decline in "time-to-test" relative the slow incline in figure 9.3. Every time you bring a component under test, and you refactor it to make it more testable, you may also be solving more testability issues for the dependencies it uses, or for other components. Specifically *because* that component has lots of dependencies, refactoring it can make a difference for the better on other parts of the system. That is why that quick decline appears. As you refactor, *other components* become easier and easier to test as well.

The "Hard First" strategy is only possible if your team has experience in unit testing techniques, since it is harder to implement. If your team *does* have experience, use the "priority" aspect of components (they are part of features in the project that have priorities) to choose whether to start with the "hard" or "easy" components first.

You might want to make a mix of them, but it's important that you know in advance what effort you are getting into and what the possible consequences are for such an effort.

9.4 Writing integration tests before refactoring

If you do plan to refactor your code for testability (so you can write unit tests) a good practical way to make sure you don't break anything during the refactoring phase is to write integration style tests against your production system.

I was once consulting for a large project, sitting with a developer that needed to some XML-configuration manager in that legacy project. The project had not tests, and was hardly testable. It was also a C++ project so we couldn't use a tool such as Typemock Isolator to isolate components without refactoring the code.

That developer needed to add another value attribute into the XML file and be able to read and change it through the existing configuration component. We ended up writing a couple of integration tests that use the real system to save and load configuration and asserted on the values that the configuration component was getting and writing to the file.

Those tests set the "original" behavior as our base of work. Next thing we wrote an integration test that shows that once the component was reading the file, it contained no attribute in memory with the name we were trying to add. We "proved" that the feature was missing, and we now had a test that will pass once we add the new attribute to the XML file and write to it correctly from the component.

Once we wrote the actual code that saves and loads the extra attribute, we ran all the three integration tests (two tests for the original, base implementation, and the new one that tries to read the new attribute). All three passed so we knew that we didn't break existing functionality while adding new functionality into an existing system.

The process is relatively simple:

- Add one or more integration tests (no mocks or stubs) to the system to prove the original system works as needed.
- Refactor or add a failing test for the feature you are trying to add to the system.
- Refactor and change the system in small chunks and run the integration tests as often as you can to know if you broke something.

Sometimes, integration tests may seem *easier* to write than unit tests, because you don't need to mess with dependency injection and such. However, making those tests run on your local system may prove a bit annoying or time consuming because every little thing that the system needs has to be in place and you're going to have to do it.

As you continue adding more and more tests, you can refactor the system and add more unit tests to it, growing it into a more maintainable and testable system. These processes take time (sometimes months and months) but they are worth it.

The trick is to work on the parts of the system that you need to fix or add features to. Not focus on other parts. That way the system grows in the right places, leaving other bridges to be crossed when you actually get to them.

Did I mention you need to have good tools? Coming up is a list of my favorites.

9.5 Important tools for legacy code unit testing

Here are a few tools that, if you are going to be doing any testing on existing code in .Net (mostly in .NET) may help you get a head start on your effort.

- Isolating Dependencies Easily with Typemock Isolator
- Finding testability problems with Depender
- Use JMockit for Java Legacy code
- Java: Use “Vide” while refactoring your code
- Fitnesse for integration/acceptance tests before you refactor
- Book: Working Effectively with legacy code (Michael Feathers)
- Use NDepends to investigate your production code
- Use Resherper to navigate and refactor your production code more easily
- Detect duplicate code (and bugs) with Simian
- Detect threading issues with Typemock Racer

Typemock Isolator has already been mentioned at the beginning of this book as the only commercial grade isolation framework (allows stubs and mocks) of the ones currently on the market. It is also very different from the other frameworks in that it is the only one that allows creating stubs and mocks of dependencies in production code without needing to refactor it *at all*, saving valuable time in bringing a component under test.

9.5.1 Isolating dependencies easily with Typemock Isolator

Full disclosure: while writing this book I’ve also been working as a developer at Typemock on a different product. I’ve also helped to design the API in Isolator 5.0.

In its latest version (5.0 at the time of writing this book) it uses the term “Fake” and removes completely the words “mock” and stub from the API. Using the new framework you can “fake” interfaces, classes that are sealed, static, non-virtual methods, and static methods. This allows not worrying about changing the design (you may not have the time or event *want* to change the design for testability reasons) and just start testing almost immediately.

Typemock Isolator also contains a *free* version for open source projects, so you can feel free to download this product and try it on your personal machine.

Here are a couple of examples of unit the new Isolator API to fake instances of classes, in listing 9.1.

Listing 9.1 Faking static methods and creating fake classes with Isolator.

```
[Test]
Public void FakeAStaticMethod()
{
   Isolate.WhenCalled(delegate{
                  MyClass.SomeStaticMethod()
    }).WillThrowException(new Exception());

//code that uses MyClass.SomeStaticMethod will get an exception automatically.
}

[Test]
Public void FakeAPrivateMethodOnAClassWithAPrivateConstructor()
{
   ClassWithPrivateConstructor c = Isolate.Fake.Instance<ClassWithPrivateConstructor>();
   Isolate.NonPublic.WhenCalled(c,”SomePrivateMethod”).WillReturn(3);

//you can now pass the ‘c’ instance around and the private method inside that class will
always return 3
}
```

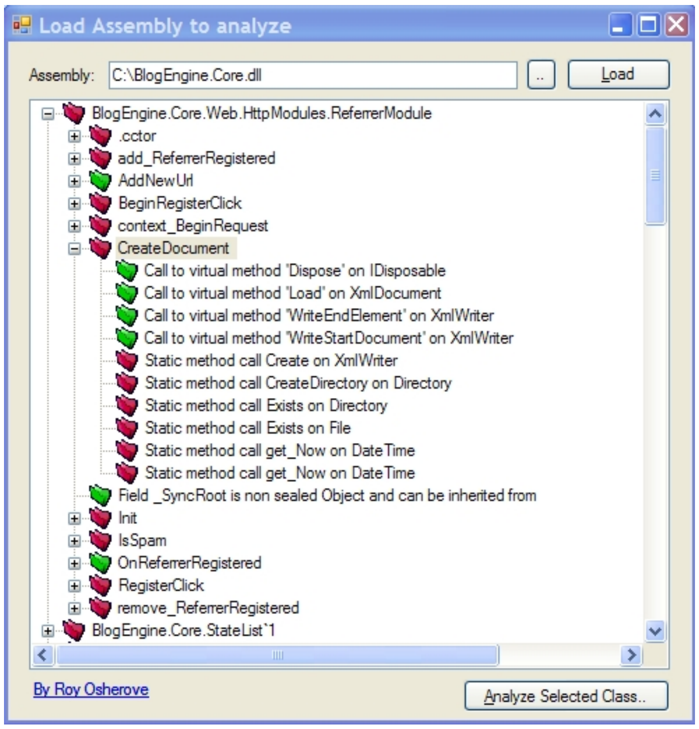
As you can see, the API is simple and clear, and uses generics and delegates (you can also use lambda syntax if you’re comfortable with it) to return fake values. You will not need to change anything in the design of your class to make these tests work, because Isolator uses the specialized “extended reflection” (or “profiler APIs”) of the CLR to perform its actions, which gives it much more power than the other frameworks.

It is a great framework for places who want to start testing, have existing code bases and want a bridge that can help them cross the hard stuff to be more agile. You can find more examples and downloads at www.Typemock.com

9.5.2 Finding testability problems with Depender

Depender is a free tool that I've created that can analyze .NET assemblies for their types and methods, and help determine possible testability problems (static methods, for example) in methods. It shows a graphic UI with a simple report for an assembly or a method.

Figure 9.1 shows the UIs for an assembly and a method report:



C:\BlogEngine.Core.dll

```
BlogEngine.Core.Web.HttpModules.ReferrerModule
    .cctor
    add_ReferrerRegistered
    AddNewUrl
    BeginRegisterClick
    context_BeginRequest
    CreateDocument
        Call to virtual method 'Dispose' on IDisposable
        Call to virtual method 'Load' on XmlDocument
        Call to virtual method 'WriteEndElement' on XmlWriter
        Call to virtual method 'WriteStartDocument' on XmlWriter
        Static method call Create on XmlWriter
        Static method call CreateDirectory on Directory
        Static method call Exists on Directory
        Static method call Exists on File
        Static method call get_Now on DateTime
        Static method call get_Now on DateTime
    Field _SyncRoot is non sealed Object and can be inherited from
    Init
    IsSpam
    OnReferrerRegistered
    RegisterClick
    remove_ReferrerRegistered
BlogEngine.Core.StateList`1
```

Figure 9.1: Depender UI: simple and easy to use.

You can download Depender from my blog:
http://weblogs.asp.net/rosherove/archive/2008/07/05/introducing-depender-testability-problem-finder.aspx

9.5.3 Use JMockit for java Legacy code

Jmockit is an open source project that uses the java instrumentation apis to do some of the same things that Typemock Isolator does in .NET. You don't need to change the design of your existing project to isolate your components from their dependencies.

The approach it uses is a "swap" approach. First, you can create a manually coded class that will replace the class that acts as a dependency to your component under test (say, a Database class, which you want to replace with your own FakeDatabase class) . Then you use Jmockit to "swap" calls from the original class to your own fake class. You can also swap "redefine" class to anonymous methods that you create at runtime.

Listing 9.6 shows a sample of a test that uses JMockit

Listing 9.2 using JMockit to swap class implementations

```
public class ServiceATest extends TestCase   {
      private boolean serviceMethodCalled;

      public static class MockDatabase      {
         static int findMethodCallCount;
```

```
            static int saveMethodCallCount;

            public static void save(Object o)    {
               assertNotNull(o);
               saveMethodCallCount++;
            }

            public static List find(String ql, Object arg1)   {
               assertNotNull(ql);
               assertNotNull(arg1);
               findMethodCallCount++;
               return Collections.EMPTY_LIST;
            }
         }

         protected void setUp() throws Exception  {
            super.setUp();
            MockDatabase.findMethodCallCount = 0;
            MockDatabase.saveMethodCallCount = 0;
            Mockit.redefineMethods(Database.class, MockDatabase.class);
         }

         public void testDoBusinessOperationXyz() throws Exception  {
            final BigDecimal total = new BigDecimal("125.40");

            Mockit.redefineMethods(ServiceB.class, new Object()
              {
               public BigDecimal computeTotal(List items)
               {
                  assertNotNull(items);
                  serviceMethodCalled = true;
                  return total;
               }
            });

            EntityX data = new EntityX(5, "abc", "5453-1");
            new ServiceA().doBusinessOperationXyz(data);

            assertEquals(total, data.getTotal());
            assertTrue(serviceMethodCalled);
            assertEquals(1, MockDatabase.findMethodCallCount);
            assertEquals(1, MockDatabase.saveMethodCallCount);
         }
   }
   }
```

Jmockit is a good start for java legacy code testing.

9.5.4 Java: Use "Vise" while refactoring your code

Michael Feathers wrote an interesting tool for java that allows you to verify that you aren't messing up the values that may change in your method while refactoring it. For example, if your method changes an array of values and you want to make sure that as you refactor you don't screw up some value in the array, as you refactor your method. Listing 9.3 shows an example of using the Vise.grip() method for just such a purpose.

Listing 9.3 Using Vise in java code to verify values to no change while refactoring

```
import vise.tool.*;

public class RPRequest {
    ...
    public int process(int level, RPPacket packet) {
        if  (...) {
            if (...) {
                ...
            } else {
                ...
                bar_args[1] += list.size();
                Vise.grip(bar_args[1]);
```

```
                    packet.add(new Subpacket(list, arrivalTime));
                    if (packet.calcSize() > 2)
                        bar_args[1] += 2;
                    Vise.grip(bar_args[1]);
                }
            } else {
                int reqLine = -1;
                bar_args[0] = packet.calcSize(reqLine);
                Vise.grip(bar_args[0]);
                ...
        }
        }
    }
```

Vise forces you to add lines to your production code, and is there to support refactoring of the code. There is no such tool for .NET though I imagine it should be pretty easy to write one. Every time you call the Vise.grip() method, it will check if the value of the passed in variable is still what it is supposed to be. It's like adding an internal assert to your code, with a simple syntax. Vise can also report on all "gripped" items and their current value.

You can read about and download Vise free here:

http://www.artima.com/weblogs/viewpost.jsp?thread=171323

9.5.5 Fitnesse for integration\acceptance tests before you refactor

It's a good idea to add integration tests to your code before you start refactoring it. Here is one tool that helps create a suite of integration\acceptance style tests.

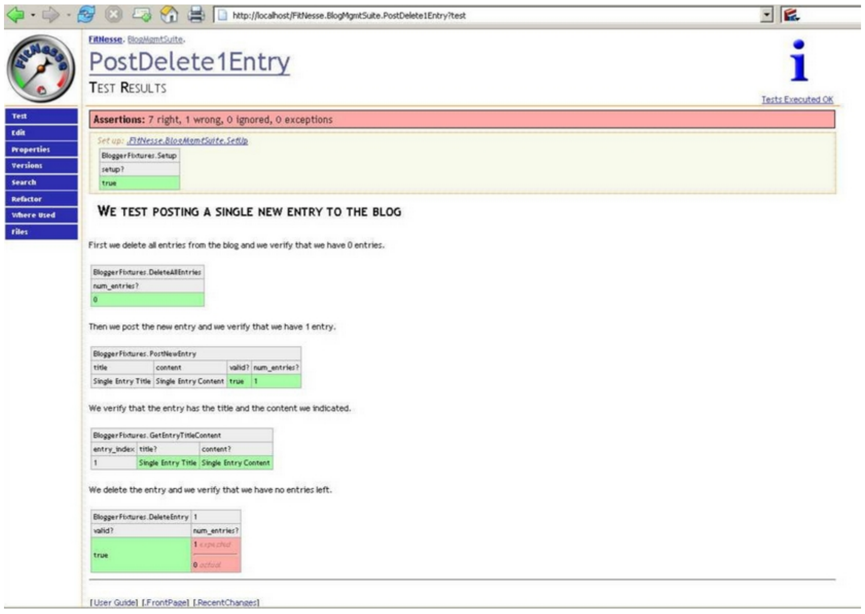
Fitnesse is a tool that allows writing integration style tests (in java or .NET) against your application, and then change or add to them easily without needing to write code. Using the framework comprises of two steps:

8. Create code "adapter" classes that can wrap your production code, and represent actions that a use might take against it (if it were a banking application, you may have a bankingAdapter class that has a "withdraw" and "deposit" methods.

9. Create specialized "tables" using a special syntax that the fitnesse engine recognizes and parses. You write those tables inside pages located in a specialized "wiki" website that runs the fitnesses engine underneath. That means that your test suite is represented to the outside world using a specialized website.

Each page with such a "table" (which you can see in any internet browser) is editable like a regular Wiki page. Each page also has a special "execute tests" button. That button will invoke the engine underneath with the parameters inside the table (values in columns, basically. Eventually the engine calls your specialized "wrapper" classes that invoke the target application and asserts on return values from your wrapper classes.

Figure 9.5 shows an example table in a browser using fitnesse.



PostDelete1Entry

TEST RESULTS

Assertions: 7 right, 1 wrong, 0 ignored, 0 exceptions

Set up: .FitNesse.BlogMgmtSuite.SetUp

BloggerFixtures.Setup
setup?
true

WE TEST POSTING A SINGLE NEW ENTRY TO THE BLOG

First we delete all entries from the blog and we verify that we have 0 entries.

BloggerFixtures.DeleteAllEntries
num_entries?
0

Then we post the new entry and we verify that we have 1 entry.

BloggerFixtures.PostNewEntry			
title	content	valid?	num_entries?
Single Entry Title	Single Entry Content	true	1

We verify that the entry has the title and the content we indicated.

BloggerFixtures.GetEntryTitleContent		
entry_index	title?	content?
1	Single Entry Title	Single Entry Content

We delete the entry and we verify that we have no entries left.

BloggerFixtures.DeleteEntry	1
valid?	num_entries?
true	1 *expected* 0 *actual*

[User Guide] [.FrontPage] [.RecentChanges]

Figure 9.5 Using fitnesse for integration testing

You can learn more about fitnesse at http://fitnesse.org/. For .NET integration with Fitnesse go to http://fitnesse.org/FitNesse.DotNet.

9.5.6 Book: Working effectively with legacy code (Michael Feathers)

"Working Effectively with legacy code" is the only book I've known to deal with the issues that you'll encounter other than this chapter. And it's worth its weight in gold. Go get it.

It shows many refactoring techniques and gotchas to an extent that this book does not attempt to cover. Get it from http://www.amazon.com/Working-Effectively-Legacy-Robert-Martin/dp/0131177052.

9.5.7 Use NDepends to investigate your production code

NDepends is a relatively new commercial .NET related analyzer tool for .NET that can create visual representations of many aspects of your compiled assemblies. Anything from showing dependency graphs to problematic code complexities. The breadth of the possibilities of this tool is huge and I'd recommend any developer to learn how to use it.

Its most powerful feature is a special query language you can use to query against the structure of your code (called CQL) to find out various metrics of components that answer to a specific criteria. For example, you could create a query that shows a report of all components that have a private constructor very easily.

You can get this tool from http://www.ndepend.com.

9.5.8 Use Resharper to navigate and refactor your production code more easily

Resharper is one of the best productivity related plugins for VS.NET.other than powerful automated refactoring abilities (much more powerful than the ones built into visual studio 2008), it's also known for its navigation features.

When jumping into an existing project, it has unmatched abilities to easily navigate the code base, with shortcuts that allow you to jump from any point in the solution to any other point that might be related to it. Some examples of possible navigations:

- When in a class declaration or a method declaration you can jump to any inheritors of that class or method, or jump "up" to the base implementation of the current member or class if such exists.
- Find all usages of a given variable (highlighted in the current editor).
- Find all usages of a common interface or a class that implements it.

All these and many more make it a much less painful experience to navigate and understand the structure of existing code.

It works both on VB.NET and C# code. In addition, you can download a trial version at www.jetbrains.com.

9.5.9 Detect duplicate code (and bugs) with Simian

Let's say you found a bug in your code. Now you want to make sure that bug was not duplicated somewhere else. With Simian, it's easier to track down code duplication so you know more than you did about how much work you have ahead of you. Simian is a commercial product that works on .NET, Java, C++ and more languages.

You can get Simian here: http://www.redhillconsulting.com.au/products/simian/

9.5.10 Detect threading issues with Typemock Racer

If your production code uses multiple threads Racer may help you discover common but hard to detect threading problems in your existing code. It is a relatively new product that aims to find deadlocks and race conditions in your existing production code without needing to change it. You can find out more about it at www.Typemock.com.

9.6 Summary

In this chapter, we talked about how to approach legacy code for the first time. It's important to map out the various components based on amount of dependencies, amount of logic and project priority. Once you have that information, you can start to choose the components to work with based either on how easy or how hard it will be to get them under test.

If your team has little or no experience in unit testing, it's a good idea to start with the easy ones and let the team grow confidence as they add more and more tests to the system. If your team is experiences, getting the hard ones under test first can get you through the rest of the system quicker.

If your team does not want to start refactoring code for testability, but only start with unit testing out of the box for good measure, a tool like Typemock Isolator would prove helpful, because it allows isolating dependencies without needing to change the existing code's design. Consider this tool whenever dealing with legacy .NET code. If you do Java, consider JMockit for the same reasons.

Last, I cover a list of tools that will prove helpful in your journey to better code quality on existing code. Each one can be used in different stages of the project, but it's up to your team to choose when to use which tool (if at all).

Also, as a friend once said, "a good bottle of vodka never hurts when dealing with legacy code".

Appendix A
Design and testability

Let’s touch on a subject matter that some developers find to be a very controversial issue: changing the design of your code so that it is more easily testable. The appendix consists of two main parts:

- First, assuming that you would *like* to do this, it teaches the basic concepts and techniques for doing so.
- The second part discusses pros and cons of this method and when it is appropriate to use it.

Why would you need to design for testability in the first place, thought?

A.1 why should I care about testability in my design?

The question is a very legitimate one. When we design our software today, we are taught to think about what the software should accomplish, and what the end product properties will be for the end user of the system. We seldom stop and think about the fact that tests against our software server are yet *another* type of user. That user has strict demands of our design, but they all stem from one mechanical request: Testability. In turn, that mechanical request can influence the design of the software in various ways, mostly for the better.

A.1.1 What is testability?

Testability: For each logical piece of code in the system (loops, *if*s, switches and such) it should be *easy and quick* to write a unit test against, that demonstrates the following *FICC* properties:

- Runs **F**ast
- **I**solated:
  - Can run independently or as part of a group of tests
  - Can run before or after any other test
- Requires no external **C**onfiguration
- Provides a **C**onsistent pass\fail result during all of the above

FICC rules: **F**ast, **I**solated, **C**onfiguration-less, **C**onsistent. If it is hard to write that test, or takes a long time to write it the system is not testable.

A.1.2 Tests as a system user

If you think of tests as a user of the system, then design for testability becomes engrained in your brain cells. If you were doing test driven development, you'd have no choice but to write a testable system, because in TDD, the tests come first and largely determine the API design of the system under test, forcing it to be something that they tests can work against.
So, What is testable design?

A.2 Design goals

There are several design points that make code much more testable. Robert C. Martin has a nice reference of design goals for object oriented systems that largely form the basis for the designs shown in this chapter. To read them go to http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod .

Most of advice I show them is about allowing your code to have *seams* – places in code where you can inject other code or replace behavior without needing to change the original class (also called the "open closed principle"). For example, in a method that calls a *web service*, the web service API can hide behind a *web service interface*, allowing us to replace the *real* web service with a *fake* one that we can use to return whatever values we want(stub), or to use it as a mock object. For a full introduction to fakes, mocks and stubs, see chapters 3, 4, and 5.

Table A.1 shows some basic pointers that we'll discuss in more detail later on in the appendix.

Table A.1 Test design guidelines and benefits

Design guideline	Benefit(s)
Make methods virtual by default	Allows overriding the methods in a derived class for testing. Overriding allows changing behavior or breaking a call to an external dependency.
Interface based designs	Allow you to use polymorphism to replace dependencies in the system with your own stubs or mocks
Make classes non sealed by default	You can't override anything virtual if the class is sealed ('final' in java)
Avoid "new" inside methods. Get instances of classes from helper methods, factories, IoC containers or other places, but not directly create them.	You will be able to serve up your own fake instances of classes to methods that require them instead of being tied down to working with an internal production instance of a class.
Avoid direct calls to static methods. Prefer calls to instance methods that later on call statics.	Will allow you to break calls to static methods by overriding instance methods (you won't be able to override static methods)
Avoid Constructors and static constructors that do logic	Overriding constructors is very hard to implement so keeping constructors simple will simplify the job of inheriting from a class in your tests.
If you have a singleton, have a way to replace its instance	Allowing you to inject a stub singleton or "reset" it.

Let's go through these guidelines one by one, see examples of breaking them and how to refactor them to make them more testable.

A.1.1 Make Methods virtual by default

Java has this feature built into the language. .NET developers aren't so lucky. In .NET, to be able to replace a method's behavior, you need to explicitly set it as virtual so you can override it in a default class. By doing this you can use the "Extract & override" method that I discussed in chapter 3 on manual stubs and mocks.

An alternative to this method is that instead of having a virtual method, you would have a custom delegate that the class invokes . You can replace this delegate from the outside by setting a property or sending in a parameter to a constructor or a method. This is not a typical approach but some system designs find this to be suitable to their needs. Listing A.1 shows an example of a class that has a delegate that can be replaced by a test.

Listing A.1: A class that invokes a delegate that can be replaced by a test.

```
public class MyOverridableClass
{
    public Func<int,int> calculateMethod=delegate(int i)
                                          {
                                              return i*2;
                                          };
    public void DoSomeAction(int input)
    {
        int result = calculateMethod(input);
        if (result==-1)
        {
            throw new Exception("input was invalid");
        }
        //do some other work
    }
}

    [Test]
    [ExpectedException(typeof(Exception))]
    public void DoSomething_GivenInvalidInput_ThrowsException()
    {
        MyOverridableClass c = new MyOverridableClass();
        int SOME_NUMBER=1;

        //stub the calculation method to return "invalid"
        c.calculateMethod = delegate(int i) { return -1; };

        c.DoSomeAction(SOME_NUMBER);
    }
```

Virtual methods are handy, but interface based designs are also a good choice, as the next section explains.

A.1.2 Interface based designs

Identifying "roles" in the application and abstracting them under interfaces is an important part of the design process. An abstract class should have no calls to "concrete" classes and concrete classes should also have no calls to concrete classes unless they may be data-objects (just holding data, with no behavior). This allows having multiple seams in the application where one could intervene and provide their own implementation. For examples on interface based replacements, see chapters 3, 4, and 5.

A.1.3 Non sealed by default

Some people have a hard time with this rule of thumb, because they like to have full control over who inherits from what in the application. The problem is that if you can't inherit from a class you can't override any virtual methods in it. Sometimes you can't keep this rule because of security concerns and such, but overall, it should be the default, not the exception.

A.1.4 Avoid instantiating concrete classes inside methods with logic

This one is a little more tricky to handle because we are sometimes *so used* to doing it.

If your method relies on a Logger, for example, don't "new" it up inside the method. Get it from a simple factory method at least. Make that factory method virtual so that you can override it later and then you can easily control what logger your method works against. Or you can use constructor injection instead of a virtual method. These and more injection methods appear in chapter 3.

A.1.5 Avoid direct calls to static methods (or statics all together)

Much like the 9.4, actively try to abstract any direct dependencies that would be hard to replace at runtime. In most cases, replacing a static method's behavior is something that is hard or very cumbersome to accomplish in a static language like VB.NET or C#. Abstracting it away using "Extract & Override" as shown in chapter 3 (under "Use a local factory method") is one way to deal with static methods.

A more extreme approach could be to try and not use any static methods whatsoever. That way everything is on an instance of a class that makes it more replaceable. That is one of the reasons some people who do unit testing or TDD dislike singletons – they act as a public shared resource that is static and cannot it's hard to override them.

Avoiding statics altogether may be a long reach, but trying to actively minimize the amount of singletons or static methods in your application is good enough to start making things easier for you while testing.

A.1.6 From statics and singletons to IoC Containers

Things like Configuration based classes are often made statics or singletons since so many parts of the application use them. One way to solve this is to start using some form of Inversion of control containers (such as Microsoft Unity, AutoFac, NInject, StructureMap, Spring.NET or Castle Windsor – all open source frameworks for .NET).

These containers can do many things, but one of the basic things they do is provide a common "smart factory" of sorts, that allows you to get instances of objects without requiring to know whether the instance is a singleton, or what the underlying implementation of that instance really is. You just ask for an interface (usually in the constructor) and an object that matches that type will be provided for you automatically as your class is being created.

When you use an IoC Container (also known as *Dependency Injection Containers*) you abstract away the lifetime management of an object type and also make it easier to create an object model that is largely based on interfaces, because filling up all the dependencies in a class is done automatically for you.

Discussing containers is outside the scope of this book. For more information, you can find a comprehensive list and starting points at Scott Hanselman's blog: http://www.hanselman.com/blog/ListOfNETDependencyInjectionContainersIOC.aspx

A.1.7 Separate Singletons and singleton holders

If you are planning to use a singleton in your design, separate the logic of the singleton class from the logic that makes it a "singleton" (the part that initializes a static variable, for example) into two separate classes. That way you can keep the SRP (Single Responsibility Principle) and also have a way to override singleton logic. Listing A.2 shows an example of this.

Listing A.2: Refactoring a singleton design into a testable design.

```
public class MySingleton
    {
        private static MySingleton _instance;
        public static MySingleton Instance
        {
            get
            {
                if (_instance == null)
                {
                    _instance = new MySingleton();
                }

                return _instance;
            }
        }

///this is the logic after refactoring
  public class RealSingletonLogic
    {
        public void Foo()
        {
            //lots of logic here
        }
    }

//this is the class that holds the logic
public class MySingletonBHolder
    {
        private static RealSingletonLogic _instance;
        public static RealSingletonLogic Instance
        {
            get
            {
                if (_instance == null)
                {
                    _instance = new RealSingletonLogic();
                }

                return _instance;
            }
        }
    }
```

Now that we've gone over some possible techniques, let's get back to the larger picture. Should you do it at all? Are there consequences for doing this?

A.2 Pros and cons of design for testability

Design for testability is a loaded subject for many people, who believe that testability in a design should be one of the default traits of designs vs. those who believe that the design shouldn't "suffer" just because someone would need to test it.

The thing to realize here is that testability is not an end goal in itself, but it is merely a bi-product of a specific school of design - a design that uses the SOLID principles laid out by Robert C. Martin (see beginning of this chapter for a reference). In a design that favors class extension-ability and abstractions, it is easy to find *seams* all over the place for testability related action. Indeed, the principles I laid out in the first part of this chapter are very orthogonal to the SOLID principles Robert Martin has discussed.

The questions remain – is this the best way to do things? What are the cons of such a method? What happens when you have legacy code? And many other questions that need to be answered.

A.2.1 Amount of work

Most likely, it is more work to design for testability than it is to not do it since it usually means writing more code. The argument can be made that the testability aspects that drive the extra design work are simply showing you things you hadn't considered that you may have been expected to do in your design anyway (separation of concerns, Single Responsibility principle etc..)

On the other hand, assuming you are happy with your design as is, many people find it problematic to change it for something that isn't part of "production". Again, you could argue that test code is as important as production code since it exposes the API usage characteristics of your domain model and forces you to look at how someone will use your code. From this point on any discussions in this matter rarely become productive, unfortunately.

Let's just agree to say that it is indeed more code\work when testability is involved, but it makes you think about the user of your API more, which is a good thing.

A.2.2 Complexity

Designing for testability can sometimes feel a little (or a lot) like it's overcomplicating things. Adding interfaces where it doesn't feel natural to use interfaces, or exposing class behavior semantics that you haven't considered before.

Most notoriously, when many things have interfaces and are abstracted away, navigating the codebase to find the *real* implementation of some method can become harder and more annoying. You *could* argue that using a tool such as resharper makes this argument obsolete since navigation with resharper becomes much easier. I agree that it eases most of the navigational pains. A tool for the right job *can help a lot*;.

A.2.3 Exposing sensitive IP

Many projects have sensitive Intellectual property that they would not be happy to expose, but which design for testability forces to expose. It could be security issues, licensing, or just plain algorithms under patent. There are workarounds for this (keeping things internal and using the [InternalsVisibleTo] attribute, but the essentially defy the whole notion of changing the design. You *are* changing the design but still keep it hidden. Big deal.

This is where I feel design for testability starts to melt down a bit. Sometimes you can't work around security or patent issues. You have to change what you do or compromise on the way you do it.

A.2.3 Sometimes you can’t

Sometimes the design has to look a specific way due to political or other reasons, and you really can’t change it, or refactor it. Sometimes you don’t have the time to look at your design and take the time to refactor it, or the design is too fragile to refactor at all.

This is another case where design for testability breaks down – when you can’t or won’t do it.

Now that we’ve gone through some pros and cons, it’s time to consider some alternatives.

A.3 Alternatives to design for testability

It is interesting to look outside the box and see other languages at work to see what other ways of working there are. Looking at dynamic languages such as Ruby or SmallTalk, the language is inherently testable because you can replace anything and everything dynamically at runtime. In such a language you can choose to design the way you want, without having to worry about testability. You don’t need an interface to replace something you’d like to replace, and you don’t need to make something public in order to override it. You can even change core types’ behavior in the system dynamically and no one would yell at you or tell you that you can’t compile.

So (imagine a movie announcer saying the following),

In a world where everything is testable, do you still design for testability?

The answer is of course – no. In that sort of world, you are free to choose your own design. Here’s a more .NET related example:

In a world where memory is managed for you, do you still design for memory management?

Mostly no, would be the answer.

By following SOLID design principles, you might *get* testable designs as a bit product, but testability should not be a goal in your design.

The main problems with “non testable” designs is the lack of ability to replace dependencies at runtime. That’s why we need to create interfaces, make methods virtual and many other things. There are tools on the market that can help replace dependencies in .NET code without needing to refactor it for testability. One such tool is Typemock Isolator (www.Typemock.com). It is a commercial tool with an open source alternative.

A.4 Summary

In this short appendix, we discussed the idea of design for testability: what it means in terms of design techniques, its pros and cons, and alternatives to it. There are no easy answers, but the questions are interesting.

The future of the world of unit testing will change significantly based on how people approach such issues, and based on what tooling is available as alternatives.

Testable designs usually only matter in static languages such as C# or VB.NET, where the testability aspect comes from our proactive design choices that allow things to be replaced. It matters less in more dynamic languages where things are much more…“agile,” shall we say? Where most things are easily replaceable as a language feature.

Testable designs mean having virtual methods, non sealed classes, interfaces, clear separation of concerns. Less statics, and much more. In fact, what Testable design is is what S.O.L.I.D design principles have stood for a long time now. Perhaps it’s time that testability will not be an end goal, but good design should be instead.

Appendix B
Extra tools and frameworks

This book would not be complete with an overview of some of the tools you can use while writing code inside today's unit test frameworks. From database testing to UI testing and ASP.NET – here is a non-binding list of tools to look at if you're looking at these tasks. Some of them are used for integration testing, and some allow unit testing. They are arranged under easy to find categories with a short description of each one. I'll also mention ones that I think are good for beginners. Here are the categories.

- Mock frameworks
- Test frameworks
- Dependency injection and Ioc containers
- Database testing
- Web Testing
- UI testing
- Threading related testing
- Acceptance testing

So, let's begin.

B.1 Mock Frameworks

Mock frameworks are the bread and butter of advanced unit testing scenarios. There are many to choose from, and that's a great thing.

- Moq (recommended, free)
- Rhino Mocks
- Typemock Isolator (recommended, commercial)
- NMock
- NUnit.Mocks

Here is a short description of each framework.

B.1.1 Moq

Moq is an open source newcomer to the world of Mocking, and has adopted an API that tries to be both very simple to learn, and very easy to use. The API also follows the AAA (Arrange Act Assert) style (vs. the Record-

Replay style) and relied heavily on .NET 3.5 features such as lambdas and extension methods. If you're planning working with .NET 3.5 exclusively this is a relatively pain free learning curve (but you need to feel really comfortable with using lambdas).

Feature wise is has less features than most other mock frameworks, which also means its simpler to learn.

You can get mock at http://code.google.com/p/moq/

B.1.2 Rhino Mocks

Rhino Mocks is a widely used open source framework for mocks and stubs. It's also the framework used throughout this book for examples. You can find out more about it in chapter 5. It is loaded with features and has recently moved to using the new AAA syntax (used to have record-replay syntax).

You can get Rhino Mocks at http://ayende.com/projects/rhino-mocks.aspx

B.1.3 Typemock Isolator

Typemock Isolator is a powerful commercial Isolation framework which tries to remove the terms "mocks" and "stubs" from its vocabulary in favor of a more simple and terse API. It differs from the other frameworks by allowing to "isolate" components from their dependencies regardless of how the system is designed (although it supports all the other features the other frameworks have). This makes it ideal for people who are getting into unit testing and want a more incremental approach to learning. Because it does not force design for testability, you can learn to write tests correctly and then move on to learning better design, without having the two being mixed together as a necessity.(disclosure: while writing this book I've also been working at Typemock). You can get Typemock Isolator at www.Typemock.com.

B.1.4 NMock

NMock is an open-source mocking framework which started out as a direct port of *JMock* from Java. It used to be the de facto mocking framework until Rhino took its place in the open source world. The main reason it was dethroned is that it did not offer strong typing of method names (you had to use strings to define expectations on methods). Lately it has been getting back into development after sitting quietly for a while, and its new released (2.0) marks a nice change in the API. It remains to be seen how well it will take vs. the current competition. You can get NMock at http://www.NMock.org/.

B.1.5 NUnit.Mocks

NUnit.Mocks is an ultra small open-source mocking framework that comes built in with Nunit as a separate dll. It initially started its life as an aid for testing NUnit internally and NUnit's authors still discourage using it in your own projects since it may "disappear" in the coming future. It is still one of the simplest frameworks to learn with, but I don't recommend using it for anything other than a learning exercise.

You can get NUnit.Mocks as part of the installation of NUnit at www.Nunit.org

B.2 Test Frameworks

The Test Frameworks are the basics from which we start writing our tests. Like mock frameworks, there are many to choose from. Competition is great. As you'll see, it brings lots of innovation with it. Here are the frameworks I will cover:

- MS Test
- NUnit (recommended)
- MbUnit
- Gallio
- XUnit
- Pex

Here is a short description of each framework.

B.2.1 Microsoft Unit Testing Framework

The Microsoft Unit Testing Framework (also known as “MS Test”) comes bundled in with any version of visual studio .NET 2008 professional or above. It allows basic features similar to NUnit but runs a little slower. The upcoming versions of Visual Studio (2010) will add a lot of power to this framework, but you can use it today just as easily as you can with NUnit.

One big con in this framework is that it’s not easily extensible as the other test frameworks. To see how cumbersome it is to add a simple attribute to it, see http://code.msdn.microsoft.com/yunit .

One big plus about it is that it integrates well as part of the Visual Studio Team System tool suite and provides good reporting, coverage, and build automation out of the box. If your company uses team system, I highly suggest using MS Test as your Test Framework of choice for the good integration possibilities.

You can get MS Test built in with Visual Studio.

B.2.2 NUnit

NUnit is currently the “De-facto” test framework for unit test developers in .NET. It’s open source and in almost ubiquitous use among those who unit test. I cover Nunit deeply in chapter 2. NUnit is easily extensible and contains a large user base and forums. As such, I’d recommend it to anyone starting out with unit testing in .NET. I still personally use it today. You can get Nunit at www.Nunit.org.

B.2.3 MbUnit

The “mb” in MbUnit stands for “Model-based” testing. It is fully open source. It started out as a competitor to Nunit and soon zoomed past NUnit in terms of features and abilities. It has its own UI and console runners that also support running tests written in NUnit. mbUnit is easily extensible and supports lots of interesting test attributes such as “Repeat” or “Timeout”. If you are looking for something more in your current test framework, MbUnit is a good way up from NUnit. Note that I personally almost never have to use such features, but if you are doing part-integration testing and part unit testing with the same framework, mbUnit is a good fit. You can get MbUnit at www.mbunit.com.

B.2.4 Gallio

The Gallio Automation Platform is an open, extensible, and neutral system for .NET that provides a common object model, runtime services and tools (such as test runners) that may be leveraged by any number of test frameworks.

Gallio seeks to facilitate the creation of a rich ecosystem of interoperable testing tools produced by and for the community at large to address a far wider range of testing needs than any previous independent testing framework has done before. Unit tests, integration tests, performance tests, and even semi-automated test harnesses should all be able to leverage common facilities where appropriate. Moreover, they should present consistent interfaces to the world so that they can easily be integrated into the systems and processes of the enterprise.

At present Gallio can run tests from MbUnit versions 2 and 3, MSTest, NBehave, NUnit, and xUnit.Net. Gallio provides tool support and integration with CCNet, MSBuild, NAnt, NCover, Pex, Powershell, Resharper, TestDriven.Net, TypeMock, and Visual Studio Team System.

Gallio also includes its own command-line runner, Echo, and a Windows GUI, Icarus. Additional runners are planned or under development. Third parties are also encouraged to leverage the Gallio platform as part of their own applications. You can get gallio at www.gallio.org.

B.2.5 XUnit

XUnit is an open source test framework, developed in cooperation with one of the original authors of Nunit, Jim Newkirk. It is a minimalistic and elegant test framework that tries to “get back to basics” by having less features, not more, than the other frameworks, and by supporting different names on its attributes.

Because it reads so radically different from the other frameworks, it takes a while to get used to if you have experience with say, NUnit or MbUnit. If you’ve never used any test framework before, xunit is very easy to grasp and use, and is robust enough to be used in a real project.

What is so radically different about it? It has no Setup or TearDown methods, for example (you have to use the constructor and a “dispose” method on the test class. Another big difference is in how easy it is to extend it.

For more information and download, see http://www.codeplex.com/xunit.

B.2.6 Pex

Pex (Program EXploration) is an intelligent assistant to the programmer. From a parameterized unit test, it automatically produces a traditional unit test suite with high code coverage. In addition, it suggests to the programmer how to fix the bugs.

With Pex you can create special tests that have parameters in them, and put special attributes on those tests. The Pex engine will *generate* new tests that you can later run as part of your test suite. It is great for finding corner cases and edge conditions that are not handled properly in your code.

You should use pex in addition to a regular test framework, such as NUnit or MbUnit. You can get it at http://research.microsoft.com/projects/pex/.

B.3 Dependency injection and Ioc Containers

Dependency Injection frameworks can be used to improve the architectural qualities of an object oriented system by reducing the mechanical costs of good design techniques (such as using constructor parameters and managing object lifetimes).

DI frameworks can enable looser coupling between classes and their dependencies, improve the testability of a class structure, and provide generic flexibility mechanisms. Used judiciously, DI frameworks can greatly enhance the opportunities for code reuse by minimizing direct coupling between classes and configuration mechanisms (such as using interfaces).

The following tools are covered:

- StructureMap
- Microsoft Unity
- Castle Windsor
- Common IServiceLocator Framework
- Managed Extensibility Framework
- Spring.NET
- Autofac
- Ninject

Here is a short description of each framework.

B.3.1 StructureMap

StuctureMap is an open source DI framework that has one clear differentiator from the others. It's API is *very* fluent and tries to mimic natural language and generic constructs as much as possible. It has a relatively small user base and the current documentation on it is lacking at best, but it contains some very powerful features such as a built in auto-mocking container (a container that can created stubs automatically when requesting an object), powerful lifetime management, XML-less configuration, integration with ASP.NET and more. You can get it at http://structuremap.sourceforge.net.

B.3.2 Microsoft Unity (Recommended)

Unity is a latecomer to the DI Container field, but provides a very simple and minimalistic approach that can be easily learned and used by beginners. Advanced users may find it lacking but it certainly answers my 80-20 rule. It provides 80% of the features you look for most of the time.

It's also open source by Microsoft and has good documentation. I'd recommend it to start with a DI container. You can get it at http://www.codeplex.com/unity.

B.3.3 Castle Windsor

The Castle project is a large open source project that covers a lot of areas. One of those areas is called "Windsor" and it provides a mature and powerful implementation of a DI container. It contains most of the features you'd ever want in a container and more, but has a relatively high learning curve due to its high amount of features.

You can learn about Castle Windsor at http://www.castleproject.org/container/ and download the castle project at http://www.castleproject.org.

B.3.4 Autofac (Auto Factory)

Autofac is a fresh approach to IoC in .NET that fits well with C# 3.0 syntax. It was designed with modern .NET features and obsessive object-orientation in mind. The API is radically different from the other frameworks, and required a bit of getting used to. It also required .NET 3.5 to work and a good knowledge of lambda syntax. It's hard to explain, so you'll have to go to the site and see how different it is. I recommend using it for people who already have experience with other DI frameworks. You can get it at http://code.google.com/p/autofac/ .

B.3.5 Common Service Locator Library

The Common Service Locator library contains a shared interface for service location that application and framework developers can reference. The library provides an abstraction over IoC containers and service locators. Using the library allows an application to indirectly access the capabilities without relying on hard references. The hope is that using this library, third-party applications and frameworks can begin to leverage IoC/Service Location without tying themselves down to a specific implementation.

The project contains a full test suite that validates a particular implementation of the locator meets the functional requirements. Additionally, several adapters for working with popular IoC containers will be included within the project at a future date.

You can get the Common Service Locator library at http://www.codeplex.com/CommonServiceLocator.

B.3.6 Spring.NET

Spring.NET is an open source DI framework. The design of Spring.NET is based on the Java version of the Spring Framework, which has shown real-world benefits and is used in thousands of enterprise applications worldwide. As such, it is one of the oldest DI frameworks for .NET, with the most “baggage” but with loads of features. Spring .NET is not a quick port from the Java version, but rather a “spiritual port” based on following proven architectural and design patterns in that are not tied to a particular platform.

You can get Spring.NET at http://www.springframework.net

B.3.7 Microsoft Managed Extensibility Framework

The Managed Extensibility Framework (MEF) enables greater reuse of applications and components. Using MEF, .NET applications can make the shift from being statically compiled to dynamically composed. If you are building extensible applications, extensible frameworks and application extensions, then MEF is for you.

MEF includes a lightweight DI framework so you can easily inject dependencies into various places in your code by using special attributes.

It does require a bit of a learning curve, and I wouldn't recommend using it solely as a DI framework. However, if you do use it for extensibility features in your application, it can also be used as a DI container. You can get MEF at http://www.codeplex.com/MEF.

B.3.8 Ninject

Ninject is a latecomer to the DI field but has a very simple syntax to learn, with very good usability. There isn't much else to day about it except that I'd highly recommend taking a look at it at http://ninject.org/.

B.4 Database testing

Database testing is always a burning question for those who are just starting out. “How do I test the DB?” “Should I stub out the database in my tests?” and many other questions arise. This section tries to deal with this issue.
First, let's talk about doing integration tests against the database.

B.4.1 Use integration tests for your data layer

Lots of people have asked me how I think they should test their data layer. Should they abstract away the database interfaces? Should they use the real database?

My answer is that I usually write integration style test for my data layer (they later in my app that talks directly to the database). The reason for this is that data logic is almost always divided between the application logic and the database itself (triggers, security rules, referential integrity) unless we can test the database logic in complete isolation. (I've found no really good framework for this purpose, only some really bad ones.) The only way to make sure it works in our tests is to couple testing the data layer logic to the real database.

By testing the data layer and the database together, we leave little surprises for later on in the project in those places.

Testing against the database, however, has its problems. It's main problem is that you are testing against shared state by many tests. Insert a line to the DB in one test, the other test can see that line as well.

We need a way to roll back the changes we made to the database, and thankfully, there is good support for this in the current test tools and the .NET framework.

B.4.2 Using Rollback attributes

The three major frameworks, MbUnit, NUnit and XUnit support the special [Rollback] attribute that you can put on top of your test method. When used, the attribute creates a special database transaction that the test code runs in. when the test is finished, the DB transaction is rolled back automatically and the changes to the DB vanish.
To learn more about how this works visit an MSDN article I wrote a while back called "Simplify Data Layer Unit Testing using Enterprise Services" at http://msdn.microsoft.com/en-us/magazine/cc163772.aspx.

If you are not interested in using the Rollback attributes the frameworks provide, or are testing in other means, you can use the very simple class introduced in .NET 2.0 called TransactionScope.

B.4.3 Using TransactionScope to Rollback

For examples on how to use this class in your setup and teardown code see this blog article: http://www.bbits.co.uk/blog/archive/2006/07/31/12741.aspx called "Even simpler Database Unit Testing with TransactionScope"

B.5 Web Testing

"How do I test my web pages" is another question that comes up a lot. I'll list some tools that can help you in this quest. Here they are:

- Ivonna
- Team System Web Test
- NUnitASP
- Watin
- Watir
- Selenium

Following is a short description of each tool.

B.5.1 Ivonna

Ivonna is a commercial Asp.Net testing tool that allows writing unit tests for your WebForms applications. What makes it different is that Ivonna runs in the same process as the test runner.

Unlike other testing tools, Ivonna is not a "client-side" tool. It doesn't just test the HTML output (although such a possibility exists), it creates and runs an Asp.Net request and allows you to examine the intrinsic objects, such as the Page object. This opens many new possibilities, such as examining properties of page controls and other objects (such as the HttpContext object), Faking external dependencies in order to isolate the page behavior, Injecting setup code and assertions into the page's lifecycle event handlers and more.

Ivonna is being developed in partnership with TypeMock and runs as an add-on to the TypeMock Isolator framework.

B.5.2 Team System Web Test

Visual Studio Team Test includes the powerful ability to record and replay web requests for pages, and verify various things during these runs. This is strictly integration testing, but it is really powerful. The latest versions also support recording ajax actions on the page, and make things much easier to test.

B.5.3 NUnitASP

NUnitAsp is no longer supported but still has a wide user base. It is a class library for use within your NUnit tests. It provides NUnit with the ability to download, parse, and manipulate ASP.NET web pages.

With NUnitAsp, your tests don't need to know how ASP.NET renders controls into HTML. Instead, you can rely on the NUnitASP library to do this for you, keeping your test code simple and clean. For example, your tests don't need to know that a DataGrid control renders as an HTML table. You can rely on NUnitAsp to handle the details. This gives you the freedom to focus on functionality questions, like whether the DataGrid holds the expected values.

Most people end up using it for acceptance testing. In those cases, it would be better to use frameworks such as Watin and Watir, described later. You can get it at http://nunitasp.sourceforge.net/

B.5.4 Watir

Watir(pronounced "water") stands for "Web application testing in Ruby". It is a simple open-source library for automating web browsers written in Ruby. It allows you to write tests that are easy to read and easy to maintain. It is optimized for simplicity and flexibility.

Watir drives browsers the same way people do. It clicks links, fills in forms, and presses buttons. Watir also checks results, such as whether expected text appears on the page.

Watir works with Internet Explorer on Window and is currently being ported to support Firefox and Safari.

The big problem is that it has a learning curve: you need to learn Ruby to use it. Other than that it is quite a lovely framework for integration testing the browser. You can get it at http://wtr.rubyforge.org/.

B.5.5 Watin

Inspired by Watir, development of WatiN(pronounced "what-in") started in December 2005 to make a similar kind of Web Application Testing possible for the .Net languages. Since then WatiN has grown into an easy to use, feature rich and stable framework. WatiN is developed in C# and aims to bring you an easy way to automate tests with Internet Explorer.

If you know .NET you can learn to use this framework, and it is quite powerful. You can get it at http://watin.sourceforge.net/.

B.5.6 Selenium

Selenium is a suite of tools to automate web app testing across many platforms. It has existed longer than all the other frameworks in this list, and also has an API wrapper for .NET. It is an integration testing framework in very wide use, and is a good place to start. Beware – it has many features and the learning curve is high.
You can get it at http://selenium.openqa.org/.

B.6 UI Testing (Winform)

UI testing is always a difficult task. I'm not a great believer in writing unit tests or integration tests for UI because I think the ROI on such tests is to low compared to the amount of time you invest in writing them. UI changes too much to be testable in a consistent manner, as far as I'm concerned. That's why I usually try to separate all the logic from the UI into a lower layer that I can test separately with standard unit testing techniques.

Nevertheless, several tools try to make the UI testing job easier:

- NUnitForms
- Project White
- Team System UI Tests

Here is a short rundown of each tool.

B.6.1 NUnitForms

NUnitForms is an NUnit extension for unit and acceptance testing of Windows Forms applications. It aims to make it easy to write automated tests for your Windows Forms (UI) classes. NUnitForms tests can open a window and interact with the controls. The tests will automatically manipulate and verify the properties of the gui.

NUnitForms takes care of cleaning up your forms between tests, detecting and handling modal dialog boxes, and verifying that your expectations for the test are fulfilled. You can get it at http://nunitforms.sourceforge.net/.

B.6.2 Project White

Project White is, in a way, a successor to NUnitForms, in that it supports a richer set of application types (WPF, WinForm, Win32 and Java JWT), and sports a newer API with better usability. Unline NUnitForms, it is more of an integration test framework, since it allows spawning separate processes that it tests against. It uses the UI Automation API (part of windows) to do its bidding, which gives it much more power. This of it as Selenium or Watin for Winforms. You can get it at http://www.codeplex.com/white.

B.6.3 Team system UI tests

The upcoming version of Team System will support a new kind of test – UI Test. You will be able to record actions on UI windows and then play them back and verify assertions during test runs. As will all team system tools, its main advantage will be its integration with other team system tools, reports, source control and servers.

B.7 Threading-related testing

Threads have always been the bane of unit testing. They are simply untestable. That's why new frameworks that let you test that thread logic does not do any hard (deadlock and race conditions) are emerging.

- Typemock Racer
- Microsoft Chess
- Osherove.ThreadTester

Here is a brief rundown of each tool.

B.7.1 Typemock Racer

Typemock Racer is a managed framework for multi-threaded code testing, that helps visualize, detect and resolve deadlocks and race conditions in managed code. You use it by simply putting an attribute on top of an existing test method and the engine does all the work. It also allows full thread visualization during debugging of a threaded test.

It is a commercial product that works with all flavors of Visual Studio (including Express) and all test frameworks. You can get it at www.Typemock.com.

(Disclosure: During the writing of this book, I joined the developer team at Typemock.)

B.7.2 Microsoft Chess

CHESS is an automated tool for finding errors in multithreaded software by systematic exploration of thread schedules. It finds errors, such as data-races, deadlocks, hangs, and data-corruption induced access violations, that are extremely hard to find with current testing tools. Once CHESS locates an error, it provides a fully repeatable execution of the program leading to the error, thus greatly aiding the debugging process. In addition, CHESS provides a valuable and novel notion of test coverage suitable for multithreaded programs. CHESS can use existing concurrent test cases and is therefore easy to deploy. Both developers and testers should find CHESS useful.

It currently supports native code, but a managed .NET version should be coming out soon as part of Visual Studio Team System offering (commercial product). You can get it at http://research.microsoft.com/projects/CHESS/.

B.7.3 Osherove.ThreadTester

This is a little open source framework I developed a while back. It allows running multiple threads during one test to see if anything weird happens to your code (deadlocks, for example). It is not feature complete, but it is a good attempt at a multithreaded test (rather than a test for multi threaded code).

You can get it at http://weblogs.asp.net/rosherove/archive/2007/06/22/multi-threaded-unit-tests-with-osherove-threadtester.aspx.

B.8 Acceptance testing

Acceptance tests enhance collaboration between customers and developers in software development. They enables customers, testers, and programmers to learn what their software should do, and automatically compare that to what it actually does do. They compares customers' expectations to actual results. It's a great way to collaborate on complicated problems (and get them right) early in development.

Unfortunately, there are not a lot of frameworks for automated acceptance testing (just one that works these days!). I'm hoping this will change soon. Here are the tools I mention:

- Fitnesse
- StoryTeller

Let's take a closer look.

B.8.1 Fitnesse

Fitnesse is a lightweight, open-source framework that makes it easy for software teams to define Acceptance Tests – web pages containing simple tables of inputs and expected outputs, and then run those tests and see the results.
Fitnesse is quite buggy but has been in use in many places with varying degrees of success.
You can learn more about Fitnesse at www.fitnesse.org.

B.8.2 StoryTeller

StoryTeller is a new tool for efficient creation and management of automated testing of .Net code with the NFit/FitNesse engine. StoryTeller is specifically created to support an Acceptance Test Driven Development strategy. All existing .Net FitNesse tests will run under StoryTeller. Features will include editing, tagging, and integration with source control, CruiseControl.Net, NAnt and/or MSBuild, and support for application versioning.
It is still under development. The project page is at http://storyteller.tigris.org/ but most of the real details are found on the author's blog: http://codebetter.com/blogs/jeremy.miller/archive/tags/StoryTeller/default.aspx.