

The pillars of good unit tests



This chapter covers

- Writing trustworthy tests
- Writing maintainable tests
- Writing readable tests
- Exploring naming conventions for unit tests

No matter how you organize your tests, or how many you have, they're worth very little if you can't trust them, maintain them, or read them. The tests that you write should have three properties that together make them good:

- *Trustworthiness*—Developers will want to run trustworthy tests, and they'll accept the test results with confidence. Trustworthy tests don't have bugs, and they test the right things.
- *Maintainability*—Unmaintainable tests are nightmares because they can ruin project schedules, or they may be sidelined when the project is put on a more aggressive schedule. Developers will simply stop maintaining and fixing tests that take too long to change or that need to change very often on very minor production code changes.
- *Readability*—This means not just being able to read a test but also figuring out the problem if the test seems to be wrong. Without readability, the other

two pillars fall pretty quickly. Maintaining tests becomes harder, and you can't trust them anymore because you don't understand them.

This chapter presents a series of practices related to each of these pillars that you can use when doing test reviews. Together, the three pillars ensure your time is well used. Drop one of them, and you run the risk of wasting everyone's time.

8.1 **Writing trustworthy tests**

There are several indications that a test is trustworthy. If it passes, you don't say, "I'll step through the code in the debugger to make sure." You trust that it passes and that the code it tests works for that specific scenario. If the test fails, you don't tell yourself, "Oh, it's supposed to fail," or "That doesn't mean the code isn't working." You believe that there's a problem in your code and not in your test. In short, a trustworthy test is one that makes you feel you know what's going on and that you can do something about it.

In this chapter, I'll introduce guidelines and techniques to help you

- Decide when to remove or change tests
- Avoid test logic
- Test only one concern
- Separate unit from integration tests
- Push for code reviews as much as you push for code coverage

I've found that tests that follow these guidelines tend to be tests that I can trust more than others and that I feel confident will continue to find real errors in my code.

8.1.1 **Deciding when to remove or change tests**

Once you have tests in place, passing, you should generally not want to change or remove them. They are there as your safety net to let you know if anything breaks when you change your code. That said, there are times you might feel compelled to change or remove existing tests. To understand when tests might cause a problem and when it's reasonable to change or remove them, let's look at the reasons for each.

The main reason for removing a test is because it fails. A test can suddenly fail for several reasons:

- *Production bugs*—There's a bug in the production code under test.
- *Test bugs*—There's a bug in the test.
- *Semantics or API changes*—The semantics of the code under test changed but not the functionality.
- *Conflicting or invalid tests*—The production code was changed to reflect a conflicting requirement.

There are also reasons for changing or removing tests when nothing is wrong with the tests or code:

- To rename or refactor the test
- To eliminate duplicate tests

Let's see how you might deal with each of these cases.

PRODUCTION BUGS

A production bug occurs when you change the production code and an existing test breaks. If indeed this is a bug in the code under test, your test is fine, and you shouldn't need to touch the test. This is the best and most desired outcome of having tests.

Because the occurrence of production bugs is one of the main reasons you 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 BUGS

If there's a bug in the test, you need to change the test. Bugs in tests are notoriously hard to detect, because tests are assumed to be correct. (That's why I like TDD so much. It's one extra way to test the test and see it fail and pass when it should.) I've detected several stages developers go through when a test bug is encountered:

- 1 *Denial*—The developer will keep looking for a problem in the code itself, changing it, causing all the other tests to start failing. The developer introduces new bugs into production code while hunting for the bug that's actually in the test.
- 2 *Amusement*—The developer will call another developer, if possible, and they will hunt for the nonexistent bug together.
- 3 *Debuggerment*—The developer will patiently debug the test and discover that there's a problem in the test. This can take anywhere from an hour to a couple of days.
- 4 *Acceptance and slappage*—The developer will eventually realize where the bug is and will slap their forehead.

When you finally find and start fixing the bug, it's important to make sure that the bug gets fixed and that the test doesn't magically pass because you test the wrong thing. You need to do the following:

- 1 Fix the bug in your test.
- 2 Make sure the test fails when it should.
- 3 Make sure the test passes when it should.

The first step, fixing the test, is straightforward. The next two steps make sure you're still testing the correct thing and that your test can still be trusted.

Once you've 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. That could mean commenting out a line or changing a Boolean somewhere, for example. Then run the test. If the test fails, that means it's half working. The other half will be completed in step 3. If the test doesn't fail, you're most likely testing the wrong thing. (I've seen developers accidentally delete the asserts from their tests when fixing bugs in tests. You'd be surprised how often that happens and how effective step 2 is at catching these cases.)

Once you see the test fail, change your production code so that the bug no longer exists. The test should now pass. If it doesn't, you either still have a bug in your test or you're testing the wrong thing. You want to see the test fail and then pass again after you fix it so that you can be sure that it fails and passes when it should.

SEMANTICS OR API CHANGES

A test can fail when the production code under test changes so that an object being tested now needs to be used differently, even though it may still have the same end functionality.

Consider the simple test in this listing.

Listing 8.1 A simple test against the LogAnalyzer class

```
[Test]
public void SemanticsChange()
{
    LogAnalyzer logan = new LogAnalyzer();

    Assert.IsFalse(logan.IsValid("abc"));
}
```

Let's say that a semantics change has been made to the LogAnalyzer class, in the form of an Initialize method. You now 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 assert line of the test in listing 8.1 will throw an exception because Initialize was not called. The test will be broken, but it's still a valid test. The functionality it tests still works, but the semantics of using the object under test have changed.

In this case, you need to change the test to match the new semantics, as shown here.

Listing 8.2 The changed test using the new semantics of LogAnalyzer

```
[Test]
public void SemanticsChange()
{
    LogAnalyzer logan = new LogAnalyzer();
    logan.Initialize();

    Assert.IsFalse(logan.IsValid("abc"));
}
```

Changing semantics accounts for most of the bad experiences developers have with writing and maintaining unit tests because the burden of changing tests while the API of the code under test keeps changing gets bigger and bigger. The following listing shows a more maintainable version of the test in listing 8.2.

Listing 8.3 A refactored test using a factory method

```
[Test]
public void SemanticsChange()
{
```

```

    LogAnalyzer logan = MakeDefaultAnalyzer();
    Assert.IsFalse(logan.IsValid("abc"));
}

public static LogAnalyzer MakeDefaultAnalyzer()
{
    LogAnalyzer analyzer = new LogAnalyzer();
    analyzer.Initialize();
    return analyzer;
}

```

← 1 Uses factory method

In this case, the refactored test uses a utility factory method ❶. You can do the same for other tests and have them use the same utility method. Then, if the semantics of creating and initializing the object should change again, you don't need to change all the tests that create this object; you need to change only one little utility method. If you get tired of creating these factory methods, I suggest you take a look at a test helper framework called AutoFixture.

I'll go into a bit more detail on it in the appendix, but in short, AutoFixture can be used, among other things, as a smart object factory, allowing you to create the object under test without worrying too much about the structure of the constructor. To find out more about this framework, Google "String Calculator Kata with AutoFixture" or go to the AutoFixture GitHub page at <https://github.com/AutoFixture/AutoFixture>. I'm still not sure if I'd be an avid user of it (because creating a factory method is really not a big deal), but it's worth taking a look and deciding for yourself if you like it. As long as your tests are readable and maintainable while using it, I won't hold it against you.

You'll see other maintainability techniques later in this chapter.

CONFLICTING OR INVALID TESTS

A conflict problem arises when the production code introduces a new feature that's in direct conflict with a test. This means that instead of the test discovering a bug, it discovers conflicting requirements.

Let's look at a short example. Suppose the customer requests LogAnalyzer to not allow filenames shorter than four letters. The analyzer should throw an exception in that case. The feature is implemented and tests are written.

Much later on, the customer realizes that three-letter filenames do have a use and requests that they be handled in a special way. The feature is added and the production code changed. As you write new tests that the production code no longer throws an exception to and make them pass, an old test (the one with a three-letter filename) suddenly breaks. It expects an exception. Fixing the production code to make that test pass would break the new test that expects three-letter filenames to be handled in a special way.

This either/or scenario, where only one of two tests can pass, serves as a warning that these may be conflicting tests. In this case, you first need to make sure that the tests are in conflict. Once that's confirmed, you need to decide which requirement to keep. You should then remove (not comment out) the invalid requirement and its

tests. (Seriously, if I catch another person commenting out something instead of deleting it, I will write a whole book titled *Why God Invented Source Control*.)

Conflicting tests can sometimes point out problems in customer requirements, and the customer may need to decide on the validity of each requirement.

RENAMING OR REFACTORING TESTS

An unreadable test is more of a problem than a solution. It can hinder your code's readability and your understanding of any problems it finds.

If you encounter a test that has a vague or misleading name or that can be made more maintainable, change the test code (but don't change the basic functionality of the test). Listing 8.3 showed one such example of refactoring a test for maintainability, which also makes it much more readable.

ELIMINATING DUPLICATE TESTS

When dealing with a team of developers, it's common to come across multiple tests written by different developers for the same functionality. I'm not crazy about 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 see different ways or semantics of testing the same thing.

Here are some of the cons of having duplicate tests:

- It may be harder to maintain several different tests that provide the same functionality.
- Some tests may be higher quality than others, and you need to review them all for correctness.
- Multiple tests may break when a single thing doesn't work. (This may not really be undesirable.)
- Similar tests must be named differently, or the tests can be spread across different classes.
- Multiple tests may create more maintainability issues.

Here are some pros:

- Tests may have small differences and so can be thought of as testing the same things slightly differently. They may make for a larger and better picture of the object being tested.
- Some tests may be more expressive than others, so more tests may improve the chances of test readability.

Although, as I said, I'm not crazy about removing duplicate tests, I usually do so; the cons usually outweigh the pros.

8.1.2 Avoiding logic in tests

The chances of having bugs in your tests increase almost exponentially as you include more and more logic in them. I've seen plenty of tests that should have been simple become dynamically logic-changing, random-number-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 didn't consider that they might have bugs or didn't write them in a maintainable manner. Those test monsters waste more time to debug and verify than they save.

But all monsters start out small. Often, a guru in the company will look at a test and start thinking, "What if we made the method loop and create random numbers as input? We'd surely find lots more bugs that way!" And you will, especially in your tests. Test bugs are one of the most annoying things for developers, because you'll almost never search for the cause of a failing test in the test itself. I'm not saying that such tests don't have any value. In fact, I'm likely to write such tests myself. But I wouldn't call them *unit tests*. I'd call them *integration tests* because they have little control of the thing they're testing and likely can't be trusted to be truthful in their results (more on this in the section about separating unit from integration tests later in this chapter).

If you have any of the following inside a unit test, your test contains logic that shouldn't be there:

- switch, if, or else statements
- foreach, for, or while loops

A test that contains logic is usually testing more than one thing at a time, which isn't recommended, because the test is less readable and more fragile. But test logic also adds complexity that may contain a hidden bug.

A unit test should, as a general rule, be a series of method calls with assert calls, but no control flows, not even try-catch, and with assert calls. Anything more complex causes the following problems:

- The test is harder to read and understand.
- The test is hard to re-create. (Imagine a multithreaded test or a test with random numbers that suddenly fails.)
- The test is more likely to have a bug or to test the wrong thing.
- Naming the test may be harder because it does multiple things.

Generally, monster tests replace original simpler tests, and that makes it harder to find bugs in the production code. If you must create a monster test, it should be added to and not replace existing tests, and it should reside in a project explicitly titled to hold integration tests, not unit tests.

Another kind of logic that it's important to avoid in unit tests can be seen in the following:

```
[Test]
public void ProductionLogicProblem()
{
    string user = "USER";
    string greeting = "GREETING";
    string actual = MessageBuilder.Build(user, greeting);
    Assert.AreEqual(user + greeting, actual);
}
```

The problem here is that the test is dynamically defining the expected result in the assert using simple logic, but still logic. The test is very likely to repeat production code logic as well as any bugs in that logic (because the person who wrote the logic and the person writing the test could be the same person or have the same misconceptions about the code).

That means that any bugs in production could be repeated in the test, and thus, if the bug exists, the test will pass. In the example code, there's a space missing in the expected value of the assert, and it's also missing from production, so the test will pass.

It would instead be better to write the test with hardcoded values like so:

```
[Test]
public void ProductionLogicProblem()
{
    string actual = MessageBuilder.Build("user", "greeting");
    Assert.AreEqual("user greeting", actual);
}
```

Because you already know how the end result should look, nothing stops you from using it in a hardcoded way. Now you don't care how the end result was accomplished, but you find out if it didn't pass. And you have no logic in your test that might have a bug.

Logic might be found not only in tests but also in test helper methods, handwritten fakes, and test utility classes. Remember, every piece of logic you add in these places makes the code that much harder to read and increases the chances of your having a bug in a utility method that your tests use.

If you find that you need to have complicated logic in your test suite for some reason (though that's generally something I do with integration tests, not unit tests), at least make sure you have a couple of tests against the logic of your utility methods in the test project. It will save you many tears down the road.

8.1.3 **Testing only one concern**

A concern, as explained before, is a single end result from a unit of work: a return value, a change to system state, or a call to a third-party object. For example, if your unit test asserts on more than a single object, it may be testing more than one concern. Or if it tests both that the same object returns the right value and that the system state changes so that the object now behaves differently, it's likely testing more than one concern.

Testing more than one concern doesn't sound so bad until you decide to name your test or consider what happens if the asserts on the first object fail.

Naming a test may seem like a simple task, but if you're testing more than one thing, giving the test a good name that indicates what's being tested becomes almost impossible. You end up with a very generic test name that forces the reader to read the test code (more on that in the readability section in this chapter). When you test just one concern, naming the test is easy.

More disturbingly, in most unit test frameworks (NUnit included) a failed assert throws a special type of exception that's caught by the test framework runner. When the test framework catches that exception, it means the test has failed. Unfortunately, exceptions, by design, don't let the code continue. The method exits on the same line where the exception is thrown. Listing 8.4 shows an example. If the first assert fails, it will throw an exception, which means the second assert will never run, and you won't know if the object behavior differed based on its state. Each of these can and should be considered different requirements, and they can and should be implemented separately and incrementally one after the other.

Listing 8.4 A test with multiple asserts

```
[Test]
public void IsValid_WhenValid_ReturnsTrueAndRemembersItLater()
{
    LogAnalyzer logan = MakeDefaultAnalyzer();

    Assert.IsTrue(logan.IsValid("abc"));
    Assert.IsTrue(logan.WasLastCallValid);
}
```

Consider assert failures as symptoms of a disease. The more symptoms you can find, the easier the disease will be to diagnose. After a failure, subsequent asserts aren't executed, and you miss seeing other possible symptoms that could provide valuable data (symptoms) that would help you narrow your focus and discover the underlying problem.

The test in listing 8.4 should really be two separate tests, with two good names.

Here's another way to think about it: if the first assert fails, do you still care what happens to the next one? If you do, you should probably separate the test into two unit tests.

Checking multiple concerns in a single unit test adds complexity with little value. You should run additional concern checks in separate, self-contained unit tests so that you can see what really fails.

8.1.4 Separate unit from integration tests

In chapter 7, I discussed the safe green zone for tests. I'm discussing this again because it's very important. If developers don't trust your tests to run out of the box easily and consistently, they won't run them. Refactoring your tests so they're easy to run and provide consistent results will make them more trustworthy. Having a safe green zone in your tests can lead to developers having more confidence in your tests. This green zone is easily created by having a separate unit tests project in which only tests that run in memory, are consistent, and are repeatable exist.

8.1.5 Assuring code review with code coverage

What does it mean when you have 100% code coverage? Nothing, without a code review. Your CEO might have asked all employees to "get over 95% code coverage," and they might have done exactly what they were asked. Maybe those tests don't even have asserts. People tend to do what they need to do to achieve a given goal metric.

What does 100% code coverage along with tests and code reviews mean? It means the world is yours for the taking. If you did code reviews and test reviews and made sure the tests are good and they cover all the code, then you're in a golden position to have a safety net that saves you from stupid mistakes, while at the same time the team benefits from knowledge sharing and continuous learning.

When I say "code review," I don't mean that half-hearted way of using a tool from halfway around the world to comment with a text line on somebody else's code, which they'll see three hours later when you're no longer at work.

No, when I say "code review," I really mean two people sitting and talking, looking at and changing the same piece of code, live. (Hopefully, they're sitting right next to each other, but remote communication apps like Skype and TeamViewer will do fine, thank you.) I'll share more about what awesome code reviews feel like in the next chapter, but for now, just know that without continuously reviewing and pairing on code and tests, you and your peers are missing a big, juicy dimension of learning and productivity. If that's the case, you should be doing everything you can to stop denying yourself this requisite essential skill. Code review is also a technique for creating readable, high-quality code that lasts for years and being able to respect yourself in the morning.

Stop looking at me like that. Your skepticism is holding you back from making your current job your dream job.

Anyway, let's talk about code coverage.

To ensure good coverage for your new code, use one of the automated tools (for example, dotCover from JetBrains, OpenCover, NCover, or Visual Studio Pro). My personal recommendation these days is NCrunch, which gives a real-time coverage red/green view of your code that changes as you're coding. It costs money but also saves money. The point is to find a good tool and master it, use it to its fullest potential, and milk value out of it, making sure you never have low coverage.

Less than 20% coverage means you're missing a whole bunch of tests, and you never know if the next developer will try to play with your code. They may try to optimize it or wrongly delete some essential line, and if you don't have a test that will fail, the mistake may go unnoticed.

When doing code and test reviews, you can also do a manual check, which is great for ad hoc testing of a test. Try commenting out a line or doing a Boolean check. If all tests still pass, you might be missing some tests, or the current tests may not be testing the right thing.

When you add a new test that was missing, check whether you've added the correct test with these steps:

- 1 Comment out the production code you think isn't being covered.
- 2 Run all the tests.
- 3 If all the tests pass, you're missing a test or are testing the wrong thing. Otherwise, there would have been a test somewhere that was expecting that line to be called or some resulting consequence of that line of code to be true, and that missing test would now fail.

- 4 Once you've found a missing test, you'll need to add it. Keep the code commented out and write a new test that fails, proving that the code you've commented is missing.
- 5 Uncomment the code you commented before.
- 6 The test you wrote should now pass. You've detected and added a missing test!
- 7 If the test still fails, it means the test may have a bug or is testing the wrong thing. Modify the test until it passes. Now you'll want to see that the test is OK, making sure it not only passes when it should, but also fails when it should. To make sure the test fails when it should, reintroduce the bug into your code (commenting out the line of production code) and see if the test indeed fails.

As an added confidence booster, you might also try replacing various parameters or internal variables in your method under test with constants (making a `bool` always `true` to see what happens, for example).

The trick to all this testing is making sure it doesn't take up too much time to make it worth your while. That's what the next section is about: maintainability.

8.2 Writing maintainable tests

Maintainability is one of the core issues most developers face when writing unit tests. Eventually the tests seem to become harder and harder to maintain and understand, and every little change to the system seems to break one test or another, even if bugs don't exist. With all pieces of code, time adds a layer of indirection between what you think the code does and what it really does.

This section covers techniques I've learned the hard way, writing unit tests with various teams. They include testing only against public contracts, removing duplication in tests, and enforcing test isolation.

8.2.1 Testing private or protected methods

Private or protected methods are usually private for a good reason in the developer's mind. Sometimes it's to hide implementation details, so that the implementation can change later without the end functionality changing. It could also be for security-related or IP-related reasons (obfuscation, for example).

When you test a private method, you're testing against a contract internal to the system, which may well change. Internal contracts are dynamic, and they can change when you refactor the system. When they change, your test could fail because some internal work is being done differently, even though the overall functionality of the system remains the same.

For testing purposes, the public contract (the overall functionality) is all that you need to care about. Testing the functionality of private methods may lead to breaking tests, even though the overall functionality is correct.

Think of it this way: no private method exists without a reason. Somewhere down the line there's a public method that ends up invoking this method, or invokes a private method that ends up invoking the method you're interested in. That means that

any private method is usually part of a bigger unit of work, or a use case in the system, that starts out with a public API and ends with one of the three end results: return value, state change, or third-party call (or all three).

With this viewpoint, if you see a private method, find the public use case in the system that will exercise it. If you test only the private method and it works, that doesn't mean that the rest of the system is using this private method correctly or handles the results it provides correctly. You might have a system that works perfectly on the inside, but all that nice inside stuff is used horribly wrong from the public APIs.

Sometimes if a private method is worth testing, it might be worth making it public, static, or at least internal and defining a public contract against any code that uses it. In some cases, the design may be cleaner if you put the method in a different class altogether. We'll look at these approaches in a moment.

Does this mean there should eventually be no private methods in the code base? No. With TDD, you usually 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.

MAKING METHODS PUBLIC

Making a method public isn't necessarily a bad thing. It may seem to go against the object-oriented principles you were raised on, but wanting to test a method could mean that the method has a known behavior or contract against the calling code. By making it public, you're making this official. By keeping the method private, you tell all the developers who come after you that they can change the implementation of the method without worrying about unknown code that uses it, because it serves as only part of a larger group of things that together make up a contract to the calling code.

EXTRACTING METHODS TO NEW CLASSES

If your method contains a lot of logic that can stand on its own, or it uses state in the class that's relevant only 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. You can then test that class separately. Michael Feathers's *Working Effectively with Legacy Code*, has some good examples of this technique, and *Clean Code* by Robert Martin can help with figuring out when this is a good idea.

MAKING METHODS STATIC

If your method doesn't use any of its class's variables, you might want to refactor the method by making it static. That makes it much more testable but also states that this method is a sort of utility method that has a known public contract specified by its name.

MAKING METHODS INTERNAL

When all else fails, and you can't afford to expose the method in an official way, you might want to make it internal and then use the `[InternalsVisibleTo("Test-Assembly")]` attribute on the production code assembly so that tests can still call that method. This is my least favorite approach, but sometimes there's no choice (perhaps because of security reasons, lack of control over the code's design, and so on).

Making the method internal isn't a great way to make sure your tests are more maintainable, because a coder can still feel it's easier to change the method. But by exposing a method as an explicit public contract, you ensure that the coder who may change it knows that the method has a real usage contract they can't break.

If you're using a pre-2012 version of Visual Studio, you might have the option to Create Private Accessor, a wrapper class that Visual Studio generates that uses reflection to call your private method. Please don't use this tool. It creates a problematic piece of code that's hard to maintain and read over time. In fact, you should avoid anything that tells you it will generate unit tests or test-related stuff for you, unless you have absolutely no choice.

Removing the method isn't a good option because the production code uses the method too. Otherwise, there'd be no reason to write the tests in the first place.

Another way to make code more maintainable is to remove duplication in tests.

8.2.2 Removing duplication

Duplication in your unit tests can hurt you as developers just as much as (if not more than) duplication in production code. The DRY principle should be in effect in test code the same as in production code. Duplicated code means more code to change when one aspect you test against changes. Changing a constructor or changing the semantics of using a class can have a major effect on tests that have a lot of duplicated code.

To understand why, let's begin with a simple example of a test.

Listing 8.5 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 listing 8.5 seems reasonable, until you introduce another test for the same class and end up with two tests, as in the next listing.

Listing 8.6 Two tests with 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);
}
```

What's wrong with the tests in the previous listing? The main problem is that if the way you use `LogAnalyzer` changes (its semantics), the tests will have to be maintained independently of each other, leading to more maintenance work. The following listing shows an example of such a change.

Listing 8.7 `LogAnalyzer` with changed semantics that now requires initialization

```
public class LogAnalyzer
{
    private bool initialized=false;

    public bool IsValid(string fileName)
    {
        if(!initialized)
        {
            throw new 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 in listing 8.6 will both break because they both neglect to call `Initialize()` against the `LogAnalyzer` class. Because you have code duplication (both of the tests create the class within the test), you need to go into each one and change it to call `Initialize()`.

You can refactor the tests to remove the duplication by creating the `LogAnalyzer` in a `CreateDefaultAnalyzer()` method that both tests can call. You could also push the creation and initialization up into a new setup method in your test class.

REMOVING DUPLICATION USING A HELPER METHOD

Listing 8.8 shows how you could refactor the tests into a more maintainable state by introducing a shared factory method that creates a default instance of `LogAnalyzer`. Assuming all the tests were written to use this factory method, you could add a call to `Initialize()` within that factory method instead of changing all the tests to call `Initialize()`.

Listing 8.8 Adding the `Initialize()` call in 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 aren't the only way to remove duplication in tests, as the next section shows.

REMOVING DUPLICATION USING [SETUP]

You could also easily initialize `LogAnalyzer` within the `Setup` method, as shown here.

Listing 8.9 Using a setup method to remove 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, you don't even need a line that creates the analyzer object in each test; a shared class instance is initialized before each test with a new instance of `LogAnalyzer`, and then `Initialize()` is called on that instance. Beware: using a setup method to remove duplication isn't always a good idea, as I'll explain in the next section.

8.2.3 Using setup methods in a maintainable manner

The `Setup()` method is easy to use. In fact, it's almost too easy—enough so that developers tend to use it for things it wasn't meant for, and tests become less readable and maintainable as a result.

Also, setup methods have limitations, which you can get around using simple helper methods:

- Setup methods can only help when you need to initialize things.
- Setup methods aren't always the best candidates for duplication removal. Removing duplication isn't always about creating and initializing new instances of objects. Sometimes it's 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 be used as factory methods that return values. They're run before the test executes, so they must be more generic in the way they work. Tests sometimes need to request specific things or call shared code with a

parameter for the specific test (for example, retrieve an object and set its property to a specific value).

- Setup methods should only contain code that applies to all the tests in the current test class, or the method will be harder to read and understand.

Now that you know the basic limitations of setup methods, let's see how developers try to get around them in their quest to use setup methods no matter what, instead of using helper methods. Developers abuse setup methods in several ways:

- Initializing objects in the setup method that are used in only some tests in the class
- Having setup code that's lengthy and hard to understand
- Setting up mocks and fake objects within the setup method

Let's take a closer look at these.

INITIALIZING OBJECTS THAT ARE USED BY ONLY SOME OF THE TESTS

This sin is a deadly one. Once you commit it, it becomes difficult to maintain the tests or even read them, because the setup method quickly becomes loaded with objects that are specific to only some of the tests. The following listing shows what your test class would look like if you initialized a `FileInfo` object setup method but used it in only one test.

Listing 8.10 A poorly implemented `Setup()` method

```
[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()
```

← Used in only
one test

```
{
    bool valid = logan.IsValid("1234567");
    Assert.IsTrue(valid);
}

private LogAnalyzer GetNewAnalyzer()
{
    ...
}
```

Why is the setup method in the listing less maintainable? Because, to read the tests for the first time and understand why they break, you need to do the following:

- 1 Go through the setup method to understand what's being initialized.
- 2 Assume that objects in the setup method are used in all tests.
- 3 Find out later you were wrong, and read the tests again more carefully to see which test uses the objects that may be causing the problems.
- 4 Dive deeper into the test code for no good reason, taking more time and effort to understand what the code does.

Always consider the readers of your tests when writing them. Imagine this is the first time they read them. Make sure they don't get angry.

HAVING SETUP CODE THAT'S LENGTHY AND HARD TO UNDERSTAND

Because the setup method provides only one place in the test to initialize things, developers tend to initialize many things, which inevitably are 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. This means that refactoring the setup method is usually a good idea. The more readable it is, the more readable your test class will be.

But there's a fine line between over-refactoring and readability. Over-refactoring can lead to less-readable code. This is a matter of personal preference. You need to watch for when your code is becoming less readable. I recommend getting feedback from a partner during the refactoring. We all can become too enamored with code we've written, and having a second pair of eyes involved in refactoring can lead to good and objective results. Having a peer do a code review (a test review) after the fact is also good but not as productive as doing it as it happens.

SETTING UP FAKES IN THE SETUP METHOD

Please don't arrange fakes in a setup method. Doing so will make it hard to read and maintain the tests.

My preference is to have each test create its own mocks and stubs by calling helper methods within the test, so that the reader of the test knows exactly what's going on, without needing to jump from test to setup to understand the full picture.

STOP USING SETUP METHODS

I've stopped using setup methods for tests I write. They're a relic from a time when it was OK to write crappy, unreadable tests, but that time is over. Test code should be

nice and clean, just like production code. But if your production code looks horrible, please don't use that as a crutch to write unreadable tests. Just use factory and helper methods, and things will be better for everyone involved.

Another great option to replace setup methods if all your tests look the same is to use parameterized tests ([TestCase] in NUnit, [Theory] in XUnit.net, or [OopsWeStillDontHaveThatFeatureAfterFiveYears] in MSTest). OK, bad joke, but MSTest still has no simple support for this.

8.2.4 Enforcing test isolation

A lack of test isolation is the biggest cause of test blockage I've seen while 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 that cried "fail"

One project I was involved in had unit tests behaving strangely, and they got even stranger as time went on. A test would fail and then suddenly pass for a couple of days straight. A day later, it would fail, seemingly randomly, and other times it would pass even if 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, that means it passes."

It turned out that the test was calling out a different test as part of its code, and when the other test failed, it would break the first test.

It took us three days to figure this out, after spending a month living with the situation. When we finally had the test working correctly, we discovered that we had 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. The story of the boy who cried "wolf" holds true even in development.

When tests aren't isolated well, they can step on each other's toes enough to make you miserable, making you regret deciding to try unit testing on the project and promising yourself never again. I've seen this happen. Developers don't bother looking for problems in the tests, so when there's a problem with them, it can take a lot of time to find out what's wrong.

There are several test "smells" that can hint at broken test isolation:

- *Constrained test order*—Tests expecting to be run in a specific order or expecting information from other test results
- *Hidden test call*—Tests calling other tests
- *Shared-state corruption*—Tests sharing in-memory state without rolling back
- *External shared-state corruption*—Integration tests with shared resources and no rollback

Let's look at these simple antipatterns.

ANTIPATTERN: CONSTRAINED TEST ORDER

This problem arises when tests are coded to expect a specific state in memory, in an external resource, or in the current test class—a state that was created by running other tests in the same class before the current test. The problem is that most test platforms (including NUnit, JUnit, and MbUnit) don't guarantee that tests will run in a specific order, so what passes today may fail tomorrow.

The following listing shows a test against `LogAnalyzer` that expects that an earlier test has already called `Initialize()`.

Listing 8.11 Constrained test order: the second test will fail if it runs 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);
    }
}
```

Myriad problems can occur when tests don't enforce isolation:

- A test may suddenly start breaking when a new version of the test framework is introduced that runs the tests in a different order.
- Running a subset of the tests may produce different results than running all the tests or a different subset of the tests.
- Maintaining the tests is more cumbersome, because you need to worry about how other tests relate to particular tests and how each one affects state.
- Your tests may fail or pass for the wrong reasons; for example, a different test may have failed or passed before it, leaving the resources in an unknown state.
- Removing or changing some tests may affect the outcomes of others.
- It's difficult to name your tests appropriately because they test more than a single thing.

There are a couple of common patterns that lead to poor test isolation:

- *Flow testing*—A developer writes tests that must run in a specific order so that they can test flow execution, a big use case composed of many actions, or a full integration test where each test is one step in that full test.
- *Laziness in cleanup*—A developer is lazy and doesn't return any state their test may have changed back to its original form, and other developers write tests that depend on this shortcoming knowingly or unknowingly.

These problems can be solved in various manners:

- *Flow testing*—Instead of writing flow-related tests in unit tests (long-running use cases, for example), consider using some sort of integration testing framework like FIT or FitNesse or QA-related products such as AutomatedQA and WinRunner.
- *Laziness in cleanup*—If you're too lazy to clean up your database after testing, your filesystem after testing, or your memory-based objects, consider moving to a different profession. This isn't a job for you.

ANTIPATTERN: HIDDEN TEST CALL

In this antipattern, tests contain one or more direct calls to other tests in the same class or other test classes, which cause tests to depend on one another. The following listing shows the `CreateAnalyzer_GoodNameAndBadNameUsage` test calling a different test at the end, creating a dependency between the tests and breaking both as isolated units.

Listing 8.12 One test calling another breaks isolation and introduces a dependency

```
[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);
    }
}
```



1 Hidden test call

This type of dependency ❶ can cause several problems:

- Running a subset of the tests may produce different results than running all the tests or a different subset of the tests.
- Maintaining the tests is more cumbersome, because you need to worry about how other tests relate to particular tests and how and when they call each other.
- Tests may fail or pass for the wrong reasons. For example, a different test may have failed, thus failing your test or not calling it at all. Or a different test may have left shared variables in an unknown state.
- Changing some tests may affect the outcome of others.
- It's difficult to clearly name tests that call other tests.

How we got here:

- *Flow testing*—A developer writes 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 a full integration test where each test is one step in that full 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 about separating the tests*—A developer is lazy and doesn't take the time to create a separate test and refactor the code appropriately, instead taking a shortcut and calling a different test.

Here are some solutions:

- *Flow testing*—Instead of writing flow-related tests in unit tests (long-running use cases, for example), consider using an integration testing framework like FIT or FitNesse, or QA-related products such as AutomatedQA and WinRunner.
- *Trying to remove duplication*—Don't ever remove duplication by calling another test from a test. You're preventing that test from relying on the setup and tear-down methods in the class and are essentially running two tests in one (because the calling test has an assertion as does the test being called). Instead, refactor the code you don't want to write twice into a third method that both your test and the other test call.
- *Laziness about separating the tests*—If you're too lazy to separate your tests, think of all the extra work you'll have to do if you don't separate them. Try to imagine a world where the current test you're writing is the only test in the system, so it can't rely on any other test.

ANTIPATTERN: SHARED-STATE CORRUPTION

This antipattern manifests in two major ways, independent of each other:

- Tests touch shared resources (either in memory or in external resources, such as databases, filesystems, and so on) without cleaning up or rolling back any changes they make to those resources.
- Tests don't set up the initial state they need before they start running, relying on the state to be there.

Either of these situations will cause the symptoms we'll look at shortly. The problem is that tests rely on specific state to have consistent pass/fail behavior. If a test doesn't control the state it expects, or other tests corrupt that state for whatever reason, the test can't run properly or report the correct result consistently.

Assume you have a `Person` class with simple features: it has a list of phone numbers and the ability to search for a number by specifying the beginning of the number. The next listing shows a couple of tests that don't clean up or set up a `Person` object instance correctly.

Listing 8.13 Shared-state corruption by a test

```
[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);
    }
}
```

Defines shared
Person state

Changes
shared state
①

Reads
shared state

In this example, the second test (expecting a null return value) will fail because the previous test has already added a number ① to the `Person` instance.

This type of problem causes a number of symptoms:

- Running a subset of the tests may produce different results than running all the tests or a different subset of the tests.
- Maintaining the test is more cumbersome, because you may break the state for other tests, breaking those tests without realizing it.
- Your test may fail or pass for the wrong reason; a different test may have failed or passed before it, leaving the shared state in a problematic condition, or it may not have cleaned up after it ran.
- Changing some tests may affect the outcomes of other tests, seemingly randomly.

Here is how we got here:

- *Not setting up state before each test*—A developer doesn't set up the state required for the test or assumes the state was already correct.

- *Using shared state*—A developer uses shared memory or external resources for more than one test without taking precautions.
- *Using static instances in tests*—A developer sets static state that's used in other tests.

Here are some solutions:

- *Not setting up state before each test*—This is a mandatory practice when writing unit tests. Either use a setup method or call specific helper methods at the beginning of the test to ensure the state 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 of an object for each test is the safest way to go.
- *Using static instances in tests*—You need to be careful how your tests manage static state. Be sure to clean up the static state using setup or teardown methods. Sometimes it's effective to use direct helper method calls to clearly reset the static state from within the test. If you're testing singletons, it's worth adding public or internal setters so your tests can reset them to a clean object instance.

ANTIPATTERN: EXTERNAL SHARED-STATE CORRUPTION

This antipattern is similar to the in-memory, shared-state corruption pattern, but it happens in integration-style testing:

- Tests touch shared resources (either in memory or in external resources, such as databases and filesystems) without cleaning up or rolling back any changes they make to those resources.
- Tests don't set up the initial state they need before they start running, relying on the state to be there.

Now that we've looked at isolating tests, let's look at managing your asserts to make sure you get the full story when a test fails.

8.2.5 *Avoiding multiple asserts on different concerns*

To understand the problem of multiple concerns, look at the following example.

Listing 8.14 A test that contains multiple asserts

```
[Test]
public void CheckVariousSumResultsOgnoringHigherThan1001()
{
    Assert.AreEqual(3, Sum(1001,1,2));
    Assert.AreEqual(3, Sum (1,1001,2));
    Assert.AreEqual(3, Sum (1,2,1001);
}
```

There is more than one test in this test method. You might say that three different sub-features are being tested here.

The author of the test method tried to save time by including three tests as three simple asserts. What's the problem here? When asserts fail, they throw exceptions. (In NUnit's case, they throw a special `AssertException` that's caught by the NUnit test

runner, which understands this exception as a signal that the current test method has failed.) Once an assert clause throws an exception, no other line executes in the test method. This means that if the first assert in listing 8.14 failed, the other two assert clauses would never execute. So? Maybe if one fails you don't care about the others? Sometimes. In this case, each assert is testing a separate feature or end result of the application, and you do care what happens to them if one fails.

There are several ways to achieve the same goal:

- Create a separate test for each assert.
- Use parameterized tests.
- Wrap the assert call with try-catch.

Why does it matter if some 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's an assumption until you can prove it with a failing or passing assert. When people see only part of the picture, they tend to make a 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 doesn't.

This applies only when you're asserting on multiple concerns. It wouldn't hold if you were testing that you got a person with name X, age Y, and so on, because if one assert failed, you wouldn't care about the others. But this would be a concern if you're expecting an action to have multiple end results. For example, it should return 3 *and* change system state. Each one of these is a feature and should work independently of other features.

I've gone on wild goose chases hunting for bugs that weren't there because only one concern out of several failed. Had I bothered to check whether the other asserts failed or passed, I might have realized that the bug was in a different location.

Sometimes people find bugs that they think are real, but when they "fix" them, the assert that previously failed passes and the *other* asserts in that test fail (or continue to fail). Sometimes you can't see the full problem, so fixing part of it can introduce new bugs into the system, which will only be discovered after you've uncovered each assert's result.

That's why it's important that all the asserts have a chance to run in the case of multiple concerns, even if other asserts have failed before. In most cases, that means putting single asserts in tests.

USING PARAMETERIZED TESTS

Both xUnit.net and NUnit support the notion of parameterized tests using a special attribute called `[TestCase]`. The following listing shows how you can use `[TestCase]` and attributes to run the same test with different parameters in a single test method. Notice that when you use the `[TestCase]` attribute, it replaces the `[Test]` attribute in NUnit.

Listing 8.15 A refactored test class using parameterized tests

```
[TestCase(1001,1,2,3)]
[TestCase (1,1001,2,3)]
[TestCase (1,2,1001,3)]
public void Sum_HigherThan1000_Ignored(int x,int y, int z,int expected)
{
    Assert.AreEqual(expected, Sum(x, y, z));
}
```

Parameterized test methods in NUnit and xUnit.net are different from regular tests in that they can take parameters. With NUnit, they also expect at least one `[TestCase]` 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 run-time to the parameters that the test method expects in its signature.

The example in listing 8.15 expects four arguments. You call an assert method with the first three parameters and use the last one as the expected value. This gives you a declarative way of creating a single test with different inputs.

The best thing about this is that if one of the `[TestCase]` attributes fails, the other attributes are still executed by the test runner, so you see the full picture of pass/fail states in all tests.

WRAPPING WITH TRY-CATCH

Some people think it's a good idea to use a try-catch block for each assert to catch and write its exception to the console and then continue to the next statement, bypassing the problematic nature of exceptions in tests. I think using parameterized tests is a far better way of achieving the same thing. Use parameterized tests instead of try-catch around multiple asserts.

Now that you know how to avoid multiple asserts acting as multiple tests, let's look at testing multiple aspects of a single object.

8.2.6 Comparing objects

Here's another example of a test with multiple asserts, but this time it's not trying to act like multiple tests in one test; it's trying to check multiple aspects of the same state. If even one aspect fails, you need to know about it.

Listing 8.16 Testing multiple aspects of the same object in one test

```
[Test]
public void Analyze_SimpleStringLine_UsesDefaultTabDelimiterToParseFields()
{
    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]);
}
```

This example is testing that the parse output from the `LogAnalyzer` worked by testing each field in the result object separately. They should all work, or the test should fail.

MAKING TESTS MORE MAINTAINABLE

The next listing shows a way to refactor the test from listing 8.16 so that it's easier to read and maintain.

Listing 8.17 Comparing objects instead of using multiple asserts

```
[Test]
public void Analyze_SimpleStringLine_UsesDefaultTabDelimiterToParseFields2 ()
{
    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);
}
```

← Sets up an expected object

← Compares expected and actual objects

Instead of adding multiple asserts, you can create a full object to compare against, set all the properties that should be on that object, and compare the result and the expected object in one assert. The advantage of this approach is that it's much easier to understand what you're testing and to recognize that this is one logical block that should be passing, not many separate tests.

IMPORTANT Note that for this kind of testing, the objects being compared must override the `Equals()` method, or the comparison between the objects won't work. Some people find this an unacceptable compromise. I use it from time to time but am happy to go either way. Use your own discretion. Because I use ReSharper, I use Alt-Insert, then select Generate Equality Members from the menu, and BAM! I get all this code generated for me for testing equality. It's pretty neat.

OVERRIDING `ToString()`

Another approach is to override the `ToString()` method of compared objects so that if tests fail, you'll get more meaningful error messages. For example, here's the output of the test in listing 8.17 when it fails:

```
TestCase 'AOUT.CH8.LogAn.Tests.MultipleAsserts
.Analyze_SimpleStringLine_UsesDefaultTabDelimiterToParseFields2'
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.CH8.LogAn.Tests.MultipleAsserts
.Analyze_SimpleStringLine_UsesDefaultTabDelimiterToParseFields2()
```

Not very helpful, is it?

By implementing `ToString()` in both the `AnalyzedOutput` class and the `LineInfo` class (which are part of the object model being compared), you can get more readable output from the tests. The next listing shows the two implementations of the `ToString()` methods in the classes under test, followed by the resulting test output.

Listing 8.18 Implementing `ToString()` in compared classes for cleaner output

```
//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.CH8.LogAn.Tests.MultipleAsserts
.Analyze_SimpleStringLine_UsesDefaultTabDelimiterToParseFields2'
failed:
    Expected: <10:05,Open,Roy,>
    But was:  <>
    C:\GlobalShare\InSync\Book\Code\ARTOfUniTesting
\LogAn.Tests\MultipleAsserts.cs(41,0):
at AOUT.CH8.LogAn.Tests.MultipleAsserts
.Analyze_SimpleStringLine_UsesDefaultTabDelimiterToParseFields2()
```

Now the test output is much clearer, and you can understand that you got very different objects. Clearer output makes it easier to understand why the test fails and makes for easier maintenance.

Another way tests can become hard to maintain is when you make them too fragile by overspecification.

8.2.7 *Avoiding overspecification*

An overspecified test is one that contains assumptions about how a specific unit under test (production code) should implement its internal behavior, instead of only checking that the end behavior is correct.

Here are ways unit tests are often overspecified:

- A test asserts purely internal state in an object under test.
- A test uses multiple mocks.
- A test uses stubs also as mocks.
- A test assumes specific order or exact string matches when it isn't required.

TIP This topic is also discussed in *xUnit Test Patterns: Refactoring Test Code* by Gerard Meszaros.

Let's look at some examples of overspecified tests.

SPECIFYING PURELY INTERNAL BEHAVIOR

The following listing shows a test against `LogAnalyzer`'s `Initialize()` method that tests internal state and no outside functionality.

Listing 8.19 An overspecified test that tests a purely internal behavior

```
[Test]
public void Initialize_WhenCalled_SetsDefaultDelimiterIsTabDelimiter()
{
    LogAnalyzer log = new LogAnalyzer();

    Assert.AreEqual(null, log.GetInternalDefaultDelimiter());
    log.Initialize();
    Assert.AreEqual('\t', log.GetInternalDefaultDelimiter());
}
```

This test is overspecified because it only tests the internal state of the `LogAnalyzer` object. Because this state is internal, it could change later on.

Unit tests should be testing the public contract and public functionality of an object. In this example, the tested code isn't part of any public contract or interface.

USING STUBS ALSO AS MOCKS

Using mocks instead of stubs is a common overspecification. Let's look at an example.

Imagine you have a data repository that you rely on to return fake data in your tests. Now, what if you used the stub that returns the fake data and also asserted that it got called? The following listing shows this.

Listing 8.20 An overspecified test that tests a purely internal behavior

```
[Test]
public void IsLoginOK_UserDoesNotExist_ReturnsFalse()
{
    IDataRepository fakeData = A.Fake<IDataRepository>();

    A.CallTo(() => fakeData.GetUserByName(A<string>.Ignored))
        .Returns(null);

    LoginManager login = new LoginManager(fakeData);

    bool result =
        login.IsLoginOK("UserNameThatDoesNotExist", "anypassword");
}
```

```

Assert.IsFalse(result);
A.CallTo(() => fakeData.GetUserByName("UserNameThatDoesNotExist"))
    .MustHaveHappened();
}

```

← You don't need to check that the stub was called. This is overspecification.

The test is overspecified because it tests the interaction between the repository stub and `LoginManager` (using `FakeItEasy`). The test should let the method under test run its own internal algorithms and test the value results. By doing that, you would have made the test less brittle. As it is, it will break if you determine that you want to add an internal call or optimize by changing call parameters. As long as the end value still holds, your test shouldn't care that something internal was called or not called at all.

In a better-defined test, the last line of that piece of code wouldn't exist.

One more way developers tend to overspecify their tests is the overuse of assumptions.

ASSUMING AN ORDER OR EXACT MATCH WHEN IT'S NOT NEEDED

Another common pattern people tend to repeat is to have asserts against hardcoded strings in the unit's return value or properties, when only a specific part of a string is necessary. Ask yourself, "Can I use `string.Contains()` rather than `string.Equals()`?"

The same goes for collections and lists. It's much better to make sure a collection contains an expected item than to assert that the item is in a specific place in a collection (unless that's exactly what's expected).

By making these kinds of small adjustments, you can guarantee that as long as the string or collection contains what's expected, the test will pass. Even if the implementation or order of the string or collection changes, you won't have to go back and change every little character you add to a string.

Now let's examine the third and final pillar of good unit tests: readability.

8.3 Writing readable tests

Without readability the tests you write are almost meaningless. Readability is the connecting thread between the person who wrote the test and the poor soul who has to read it a few months later. Tests are stories you tell the next generation of programmers on a project. They allow a developer to see exactly what an application is made of and where it started.

This section is all about making sure the developers who come after you will be able to maintain the production code and the tests that you write, while understanding what they're doing and where they should be doing it.

There are several facets to readability:

- Naming unit tests
- Naming variables
- Creating good assert messages
- Separating asserts from actions

Let's go through these one by one.

8.3.1 Naming unit tests

Naming standards are important because they give you comfortable rules and templates that outline what you should explain about the test. The test name has three parts:

- *The name of the method being tested*—This is essential, so that you can easily see where the tested logic is. Having this as the first part of the test name allows easy navigation and as-you-type intellisense (if your IDE supports it) in the test class.
- *The scenario under which it's being tested*—This part gives you the “with” part of the name: “When I call method X *with a null value*, then it should do Y.”
- *The expected behavior when the scenario is invoked*—This part specifies in plain English what the method should do or return, or how it should behave, based on the current scenario: “When I call method X with a null value, *then it should do Y*.”

Removing even one of these parts from a test name can cause the reader of the test to wonder what's going on and to start reading the test code. Your main goal is to release the next developer from the burden of reading the test code in order to understand what the test is testing.

A common way to write these three parts of the test name is to separate them with underscores, like this: `MethodUnderTest_Scenario_Behavior()`. Here's a test that uses this naming convention.

Listing 8.21 A test with three parts in its name

```
[Test]
public void
    AnalyzeFile_FileWith3LinesAndFileProvider_ReadsFileUsingProvider()
{
    //...
}
```

The method in the listing tests the `AnalyzeFile` method, giving it a file with three 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.

8.3.2 Naming variables

How you name variables in unit tests is as important as or even more important than variable-naming conventions in production code. Apart from their chief function of testing, tests also serve as a form of documentation for an API. By giving variables good names, you can make sure that people reading your tests understand what you're trying to *prove* as quickly as possible (as opposed to understanding what you're trying to *accomplish* when writing production code).

The next listing shows an example of a poorly named and poorly written test. I call this “unreadable” in the sense that I can't figure out what this test is about.

Listing 8.22 An unreadable test name

```
[Test]
public void BadlyNamedTest()
{
    LogAnalyzer log = new LogAnalyzer();

    int result= log.GetLineCount("abc.txt");

    Assert.AreEqual(-100,result);
}
```

In this instance, the assert is using some magic number (-100, a number that represents some value the developer needs to know). Because you don't have a descriptive name for what the number is expected to be, you can only assume what it's supposed to mean. The test name should have helped you a little bit here, but the test name needs more work, to put it mildly.

Is -100 some sort of exception? Is it a valid return value? This is where you have a choice:

- You can change the design of the API to throw an exception instead of returning -100 (assuming -100 is some sort of illegal result value).
- You can compare the result to some sort of constant or aptly named variable, as shown in the following listing.

Listing 8.23 A more readable version of the test

```
[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 8.23 is much better, because you can easily understand the intent of the return value.

The last part of a test is usually the assert, and you need to make the most out of the assert message. If the assert fails, the first thing the user will see is that message.

8.3.3 *Asserting yourself with meaning*

Avoid writing your own custom assert messages. Please. This section is for those who find they absolutely have to write a custom assert message, because the test really needs it, and you can't find a way to make the test clearer without it. Writing a good assert message is much like writing a good exception message. It's easy to get it wrong without realizing it, and it makes a world of difference (and time) to the people who have to read it.

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, and possibly mention when it should have happened.

The listing that follows shows a bad example of an assert message and the output it produces.

Listing 8.24 A bad assert message that repeats what the test framework outputs

```
[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.CH8.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.CH8.LogAn.Tests.Readable.BadAssertMessage()
```

As you can see, there's a message that repeats. The assert message didn't add anything except more words to read. It would have been better to not output anything but instead have a better-named test. A clearer assert message would be something like this:

```
Calling GetLineCount() for a non-existing file should have returned a
    COULD_NOT_READ_FILE.
```

Now that your assert messages are understandable, it's time to make sure that the assert happens on a different line than the method call.

8.3.4 Separating asserts from actions

This is a short section but an important one nonetheless. For the sake of readability, avoid writing the assert line and the method call in the same statement.

The following listing shows a good example, and listing 8.26 shows a bad example.

Listing 8.25 Separating the assert from the thing asserted, improving readability

```
[Test]
public void BadAssertMessage()
{
    //some code here
    int result= log.GetLineCount("abc.txt");
    Assert.AreEqual(COULD_NOT_READ_FILE,result);
}
```

Listing 8.26 Not separating the assert from the thing asserted, making reading difficult

```
[Test]
public void BadAssertMessage()
{
    //some code here

    Assert.AreEqual(COULD_NOT_READ_FILE,log.GetLineCount("abc.txt"));
}
```

See the difference between the two examples? Listing 8.26 is much harder to read and understand in the context of a real test, because the call to the `GetLineCount()` method is inside the call to the assert message.

8.3.5 Setting up and tearing down

Setup and teardown methods in unit tests can be abused to the point where the tests or the setup and teardown methods are unreadable. Usually the situation is worse in the setup method than the teardown method.

Let's look at one possible abuse. 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 from them are in the test.

It's much more readable to initialize mock objects directly in the test itself, with all their expectations. If you're 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 will know exactly what's being set up instead of having to look in multiple places.

TIP I've several times written full test classes that didn't have a setup method, only helper methods being called from each test, for the sake of maintainability. The classes were still readable and maintainable.

8.4 Summary

Few developers write tests that they can trust when they first start out writing unit tests. It takes discipline and imagination to make sure you're doing things right. A test that you can trust is an elusive beast at first, but when you get it right, you'll feel the difference immediately.

Some ways of achieving this kind of trustworthiness involve keeping good tests alive and removing or refactoring away bad tests, and we discussed several such methods in

this chapter. The rest of the chapter was about problems that can arise inside tests, such as logic, testing multiple things, ease of running, and so on. Putting all these things together can be quite an art form.

If there's one thing to take away from this chapter, it's this: tests grow and change with the system under tests.

The topic of writing maintainable tests has gained traction in the past few years, but as I write it hasn't been covered much in the unit testing and TDD literature and with good reason. I believe that this is the next step in the learning evolution of unit testing techniques. The first step of acquiring the initial knowledge (what a unit test is and how you write one) has been covered in many places. The second step involves refining the techniques to improve all aspects of the code you write and looking into other factors, such as maintainability and readability. It's this critical step that this chapter (and most of this book) focuses on.

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 they'll also trust the tests when they pass. When this point is achieved, you're ready to handle change and to change the code when it needs changing, because you'll know when things break.

In the next chapters, we'll take a broader look at unit tests as part of a larger system: how to fit them into the organization and how they fit in with existing systems and legacy code. You'll learn what makes code testable, how to design for testability, and how to refactor existing code into a testable state.

Part 4

Design and process

These final chapters cover the problems you'll face and techniques that you'll need when introducing unit testing to an existing organization or code.

In chapter 9, we'll deal with the tough issue of implementing unit testing in an organization, and we'll cover techniques that can make your job easier. This chapter provides answers to some tough questions that are common when first implementing unit testing.

In chapter 10, we'll look at common problems associated with legacy code and examine some tools for working with it.

Chapter 11 covers a common discussion around unit testing. Should you design for testability? What is a testable design anyway?

