MANNING PUBLICATIONS

The Art of Unit Testing

By Roy Osherove

Sooner or later, as you start writing tests for your applications, you're bound to refactor them, and create utility methods, utility classes, and many other constructs (either in the test projects or in the code under test) solely for the purpose of testability or test readability and maintenance. This article from The Art of Unit Testing how to use inheritance in your test classes, create test utility classes and methods, and make your API known to developers.

You may also be interested in…

Building a Test API for Your Application

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

Here are some things you may 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 developers.

Let's look at these in turn.

Using test class inheritance patterns

One of the most powerful arguments for object-oriented code is that you can reuse existing functionality instead of recreating it over and over again in other classes—what Andy Hunt and Dave Thomas called the DRY ("don't repeat yourself") principle in *The Pragmatic Programmer*.

Because the unit tests you write in .NET and most object-oriented languages are in an object-oriented paradigm, it's not a crime to use inheritance in the test classes themselves. In fact, I urge you to do this if you have a good reason to. Implementing a base class can help alleviate standard problems in test code by

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

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

These are the basic three patterns:

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

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

- Refactoring into a class hierarchy
- Using generics

Abstract test infrastructure class pattern

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

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

Listing 1 shows these classes:

- *The* `LogAnalyzer` *class and method*—The class and method we'd like to test
- *The* `LoggingFacility` *class*—The class that holds the logger implementation we'd like to override in our tests
- *The* `ConfigurationManager` *class*—Another user of `LoggingFacility`, which we'll test later
- *The* `LogAnalyzerTests` *class and method*—The initial test class and method we're going to write
- *The* `StubLogger` *class*—An internal class that will replace the real logger implementation
- *The* ConfigurationManagerTests *class*—A class that holds tests for `ConfigurationManager`

Listing 1 An example of not following the DRY principle in test classes

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

//another class that uses the LoggingFacility internally
public class ConfigurationManager
    {
        public bool IsConfigured(string configName)
        {
            LoggingFacility.Log("checking " + configName);
            //return result;
        }
    }

public static class LoggingFacility
    {
        public static void Log(string text)
        {
            logger.Log(text);
        }
        private static ILogger logger;

        public static ILogger Logger
        {
            get { return logger; }
            set { logger = value; }
        }
    }

   [TestFixture]
    public class LogAnalyzerTests
```

```
    {
       [SetUp]
        public void Setup()                              #1
        {
            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()                              #1
        {
            LoggingFacility.Logger = new StubLogger();
        }

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

#1 Uses Setup() method

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

There are two classes that use the `LoggingFacility` class internally, 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 (#1), which is essentially the same for both test classes. They both replace the default logger implementation.

We could refactor the test classes and create a base test class that contains the setup method. The full code for the test classes is shown in listing 2.

Listing 2 A refactored solution

```
public class BaseTestClass
    {
       [SetUp]
        public void Setup()                               #A
        {                                                 #A
            Console.WriteLine("in setup");                #A
            LoggingFacility.Logger = new StubLogger();    #A
        }                                                 #A
    }

   [TestFixture]
    public class LogAnalyzerTests : BaseTestClass |#2
    {
       [Test]
        public void Analyze_EmptyFile_ThrowsException()
```

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

   [TestFixture]
    public class ConfigurationManagerTests :BaseTestClass  #B
    {

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

#A Refactors into a common setup method
#B Inherits `Setup()` method implementation

Figure 1 shows this pattern more clearly.

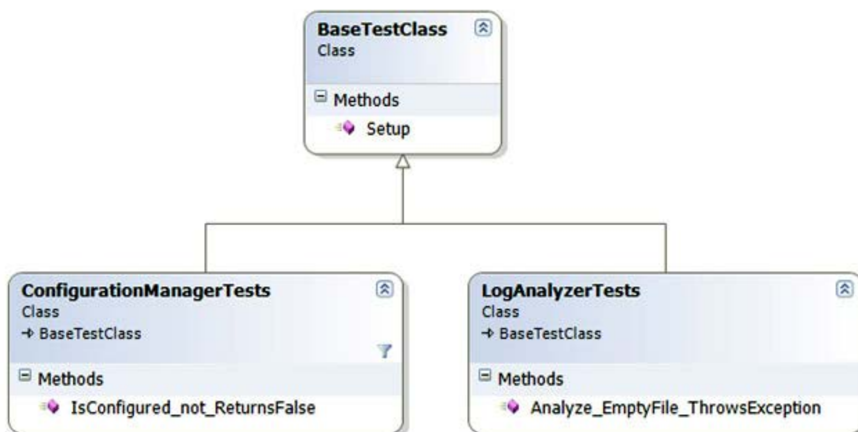

Figure 1 One base class with a common setup method, and two test classes that reuse that setup method

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 there are pros and cons in every technique. The main problem we've introduced into the derived test classes is that anyone reading the code can no longer easily understand what happens when 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.

What if you wanted to have your own derived setup in one of the derived classes? Most of the unit-testing frameworks (including NUnit) will allow you to make the setup method virtual and then override it in the derived class. Listing 3 shows how a derived class can have its own setup method but still keep the original setup method (making them work one after the other).

Listing 3 A derived test class with its own setup method

```
public class BaseTestClass
 {
    [SetUp]
     public virtual void Setup()|                          #A
     {
         Console.WriteLine("in setup");
         LoggingFacility.Logger = new StubLogger();
     }
```

```
}

[TestFixture]
public class ConfigurationManagerTests :BaseTestClass
{
    [SetUp]
    public override void Setup()                          #B
    {
        base.Setup();
        Console.WriteLine("in derived");
        LoggingFacility.Logger = new StubLogger();
    }

    //...
}
```

#A Makes `Setup()` virtual to allow overriding
#B Overrides and calls base

This style of inheritance is easier for the reader of the test class, because it specifically indicates that there's a base setup method that's called each time. You may be helping your team by requiring them to always override base methods and call their base class's implementation in the tests for the sake of readability. This approach is shown in listing 4.

Listing 4 Overriding a setup method purely for clarity

```
[TestFixture]
public class ConfigurationManagerTests :BaseTestClass
{
    [SetUp]
    public override void Setup()
    {
        base.Setup();                                      #A
    }

    //...
}
```

#A Overrides and calls base

This type of coding may feel a bit weird at first, but anyone reading the tests will thank you for making it clear what's going on.

Template test class pattern

The *template test class pattern* creates an abstract class that contains abstract test methods that derived classes will have to implement. The driving force behind this pattern is the need to be able to dictate to deriving classes which tests they should always implement. It's commonly used when there's a need to create one or more test classes for a set of classes that implement the same interface.

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

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

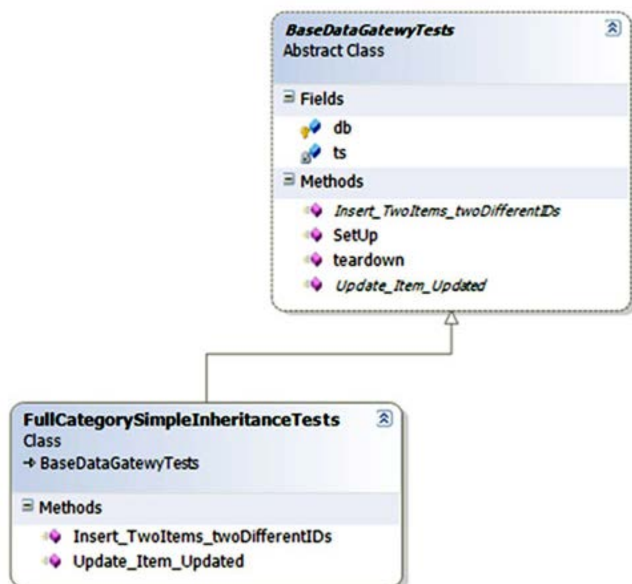

Figure 2 A template test pattern ensures that developers don't forget important tests. The base class contains abstract tests that derived classes must implement.

Figure 2 shows an example base class that helps to test data-layer CRUD (create, retrieve, update, and delete) classes.

I've found this technique useful in many situations, not only as a developer, but also as an architect. As an architect, I was able to supply a list of essential test classes for developers to implement, and to provide guidance on what kinds of tests they'd want to write next. It's essential in this situation that the naming of the tests is understandable.

But what if you were to inherit real tests from the base class, and not abstract ones?

Abstract test driver class pattern

The *abstract test driver class pattern* creates an abstract test class that contains test method implementations that all derived classes inherit by default, without needing to reimplement them. Instead of having abstract test methods, you implement real tests on the abstract class that your derived classes will inherit. It's essential that your tests don't explicitly test one class type, but instead test against an interface or base class in your production code under test.

Let's see a real scenario. Suppose you have the object model shown in figure 3 to test.

The `BaseStringParser` is an abstract class that other classes derive from to implement some functionality over different string content types.

From each string type (XML strings, IIS log strings, standard strings), we can get some sort of versioning info (metadata on the string that was put there earlier). We can get the version info from a custom header (the first few lines of the string) and check whether that header is valid for the purposes of our application. The `XMLStringParser`, `IISLogStringParser`, and `StandardStringParser` classes derive from this base class and implement the methods with logic for their specific string types.

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

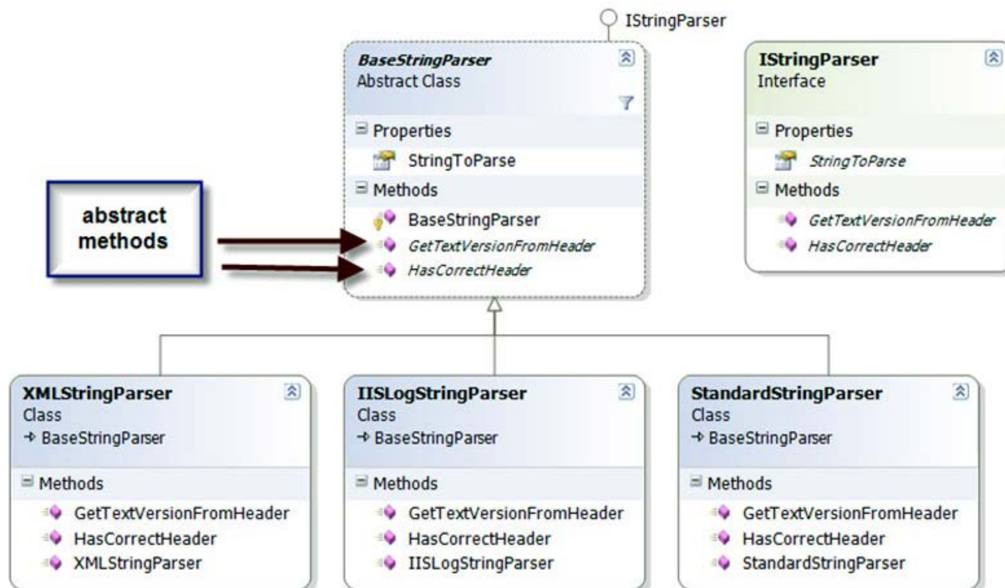

Figure 3 A typical inheritance hierarchy that we'd like to test includes an abstract class and classes that derive from it.

Listing 5 shows tests for the `StandardStringParser` that we might start out with before we refactor our test classes.

Listing 5 An outline of a test class for `StandardStringParser`

```
[TestFixture]
public class StandardStringParserTests
{
    private StandardStringParser GetParser(string input)
    {
        return new StandardStringParser(input);
    }

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

        string versionFromHeader = parser.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
    }
}
```

#1 Defines the parser factory method
#2 Uses factory method

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

When you start writing tests for the other classes in the hierarchy, you'll want to repeat the same tests that are in this specific parser class.

All the other parsers should have the same outward behavior: getting the header version and validating that the header is valid. How they do this differs, but the behavior semantics are the same. This means that, for each class that derives from `BaseStringParser`, we'd write the same basic tests, and only the type of class under test would change.

Instead of repeating all those tests manually, we can create a `ParserTestsBase` class that contains all the basic tests we'd like to perform on any class that implements the `IStringParser` interface (or any class that derives from `BaseStringParser`). Listing 6 shows an example of this base class.

Listing 6 An abstract test base class with test logic for `IStringParser` interface

```
public abstract class BaseStringParserTests
{
    protected abstract IStringParser
                         GetParser(string input);          #1

   [Test]
    public void GetStringVersionFromHeader_SingleDigit_Found()
    {
        string input = "header;version=1;\n";
        IStringParser parser = GetParser(input);                #2

        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);                #2
        //...
    }
```

```
    [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);
        //...
    }
}
```

#1 Turns `GetParser()` into an abstract method
#2 Calls abstract factory method

Several things are different from listing 5 and are important in the implementation of the base class:

- The `GetParser()` method is abstract (#1), and its return type is now `IStringParser`. This means 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 only get an `IStringParser` interface (#2) and don't know the actual class they're running against.
- A derived class can choose to add tests against a specific subclass of `IStringParser` by adding another test method in its own test class (as we'll see next).

Once we have the base class in order, we can easily add tests to the various subclasses. Listing 7 shows how we can write tests for the `StandardStringParser` by deriving from `BaseStringParserTests`.

Listing 7 A derived test class that overrides a small number of factory methods

```
[TestFixture]
public class StandardStringParserTests : BaseStringParserTests
{
    protected override IStringParser GetParser(string input)
    {
        return new StandardStringParser(input);                 #1
    }

[Test]
    public void
        GetStringVersionFromHeader_DoubleDigit_Found()          #2
    {
        //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);
    }

}
```

#1 Overrides abstract factory method
#2 Adds new test

Note that in listing 7 we only have two methods in the derived class:

- The factory method (#1) that tells the base class what instance of the class to run tests on
- A new test (#2) that may not belong in the base class, or that may be specific to the current type under test

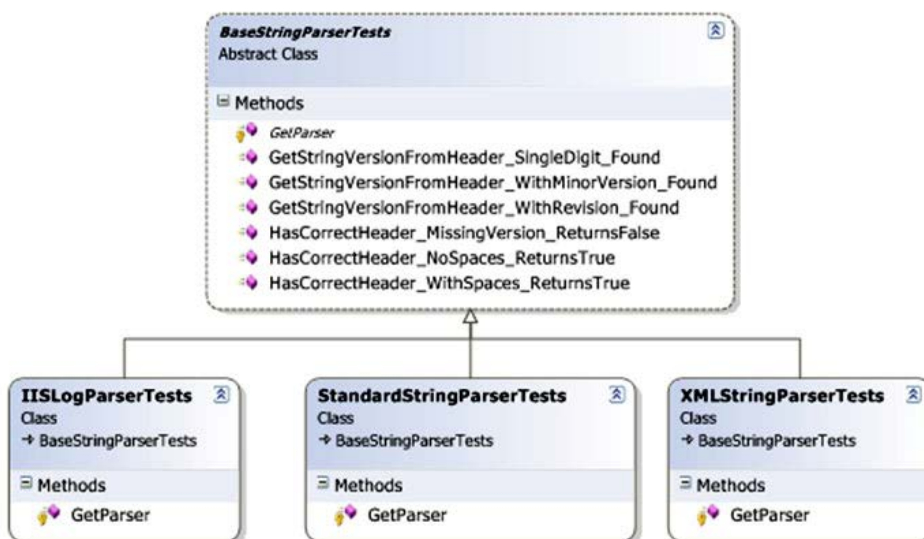

Figure 4 A standard test class hierarchy implementation. Most of the tests are in the base class, but derived classes can add their own specific tests.

Figure 4 shows the visual inheritance chain that we've just created.

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

Refactoring your test class into a test class hierarchy

Most developers don't start writing their tests with these inheritance patterns in mind. Instead, they write the tests normally, as was shown in listing 5. The steps to convert your tests into a base class are fairly easy, particularly if you have IDE refactoring tools available, like the ones found in Eclipse, IntelliJ IDEA, or Visual Studio 2008 (JetBrains' ReSharper or Refactor! from DevExpress).

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

1. Refactor: extract the superclass.
   - Create a base class (`BaseXXXTests`).
   - Move the factory methods (like `GetParser`) into the base class.
   - 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 .NET generics to create the inheritance patterns.

A variation using .NET generics to implement test hierarchy

You can use generics as part of the base test class. This way, you don't even need to override any methods in derived classes; just declare the type you're testing against. Listing 8 shows both the generic version of the test base class and a class derived from it.

Listing 8 Implementing test case inheritance with .NET generics

```
public abstract class StringParserTests<T>
    where T:IStringParser                                     #1
{
    protected T GetParser(string input)                       #2
    {
        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
    //...
}

// this is the derived test class:
[TestFixture]
 public class StandardStringParserGenericTests
              :StringParserTests<StandardStringParser>         #B
{
}
```

#1 Defines generic constraint on parameter
#2 Returns generic type
#3 Gets generic type variable instead of an interface
#A Inherits from generic base class

There are several things that change in the generic implementation of the hierarchy:

- The `GetParser` factory method (#1) no longer needs to be overridden. Create the object using `Activator.CreateInstance` (which allows creating objects without knowing their type) and send the input string arguments to the constructor.
- The tests themselves don't use the `IStringParser` interface, but instead use the T generic type (#2).
- The generic class declaration contains the `where` clause that specifies that the T type of the class must implement the `IStringParser` interface (#3).

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

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

Creating test utility classes and methods

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

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

- Factory methods for objects that are complex to create or that routinely get created by your tests.
- System initialization methods (such as methods for setting up the system state before testing, or changing logging facilities to use stub loggers).

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

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

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

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

That's why making your API known is an important next step.

Making your API known to developers

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

- Have teams of two people write tests together (at least once in a while), where one 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 document (no more than a couple of pages) or a cheat sheet that details the types of APIs out there and where to find them. You can create short documents for specific parts of your testing framework (APIs specific to the data layer, for example) or a global one for the whole application. If it's not short, no one will maintain it. One possible way to make sure it's up to date is by automating the generation process:
  - Have a known set of prefixes or postfixes on the API helpers' names (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 have some simple directives that the special tool can parse from comments you put on them.
  - Automate the generation of this document as part of the automated build process.
- Discuss changes to the APIs during team meetings—one or two sentences outlining the main changes and where to look for the significant parts. That way the team knows that this is important and it's always on people's minds.
- Go over this document with new employees during their orientation.
- Perform test reviews (as opposed to code reviews) that make sure tests are up to standards of readability, maintainability, and correctness, and ensure that the right APIs are used when needed.

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

Summary

Let's look back and see what we can draw from our discussion.

- Use a test class hierarchy to apply the same set of tests to multiple related types under test in a hierarchy, or to types that share a common interface or base class.
- Use helper classes and utility classes instead of hierarchies if the test class hierarchy makes tests less readable, especially if there's a shared setup method in the base class. Different people have different

opinions on when to use which, but readability is usually the key reason for not using hierarchies.

- Make your API known to your team. If you don’t, you’ll lose time and money as team members unknowingly reinvent many of the APIs over and over again.

Here are some other Manning titles you might be interested in:

Unit Testing in Java
Lasse Koskela

JUnit in Action, Second Edition
Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory

Test Driven
Lasse Koskela

Last updated: August 29, 2011