

# An Introduction To Test Driven Development

There are *a lot* of introductory articles for Test Driven Development (TDD). So why, might you ask, am I writing yet another? The simple answer is because I was asked to by Allan Kelly who wanted a piece for a small book he gives to his clients. I was happy to oblige, but writing about TDD is difficult. In fact if Allan hadn't wanted an original piece he could print as part of his book I would have suggested he just get a copy of Test Driven Development by Kent Beck. The main difficulty is coming up with a suitably concise, yet meaningful, example and Beck has already done this.

Allan was also quite keen for me to publish elsewhere, so I chatted the idea over with Steve Love, the editor of the ACCU's CVu magazine to see if he thought the readers would be interested in yet another TDD article. He said they probably would be as long as I thought carefully about how I wrote it. I thought this over for a long while. The majority of introductory TDD articles, at least the ones I have read, focus on unit testing. A recently completed ACCU Mentored Developers project read through Growing Object Orientated Software Guided (known as GOOS) by Tests by Freeman & Pryce [Freeman]. They focus on starting a simple application with acceptance tests and only writing unit and integration tests when the level of detail requires it or the acceptance tests become too cumbersome. However, it is a big book, so I decided to try and condense what I saw as the most important points into an introductory article and this is what you see before you. I hope you find it as useful and fun to read as I did to write.

## What?

Test Driven Development is a way of developing software by writing tests *before* you write any production code. Hence it is sometimes referred to as Test First Development. The first and definitive book on the subject is Test Driven Development by Kent Beck [Beck], which focuses on Unit Testing. However, as Steve Freeman and Nat Pryce tell us in GOOS, TDD can and should be employed at all levels, including integration and acceptance tests.

## Unit Tests

Unit tests are automated tests that test a single unit, such as a class or method. Usually the class under test is instantiated with any dependent collaborators mocked out (I'll

describe mock objects later) and then tests are run against it. In some ways it is easier to describe what a unit test is not and Michael Feathers has done this very well:

*A test is not a unit test if:*

- *It talks to the database*
- *It communicates across the network*
- *It touches the file system*
- *It can't run at the same time as any of your other unit tests*
- *You have to do special things to your environment (such as editing config files) to run it.*

## **Integration Tests**

Integration tests are automated tests that test the interaction between at least two different parts of a system. For example a test which checks that a data access layer correctly loads data from a database is an integration test.

## **Acceptance Tests**

Rachel Davies describes acceptance tests as "...scripts - manual or automated - that detail specific steps to test a feature." [Davies] Generally acceptance tests run against a *deployed* system to check that it behaves in a way that will be accepted by as a complete or partially complete feature.

## **Why?**

In *Working Effectively With Legacy Code*, Michael Feathers [Feathers] tells us that:

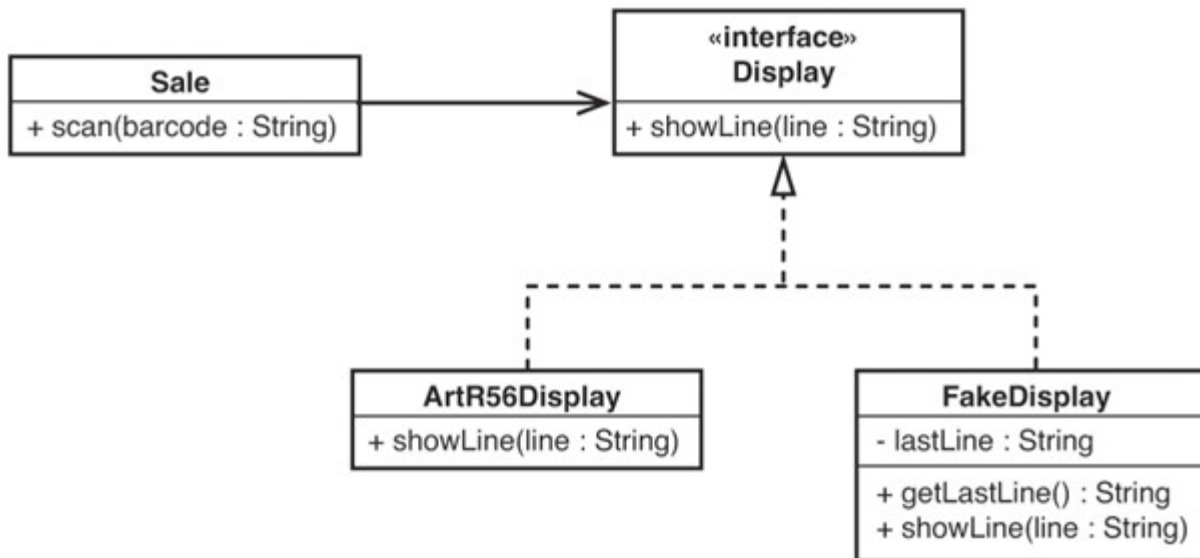
*Code without tests is bad code. It doesn't matter how well written it is; how pretty or object orientated or well encapsulated it is. With tests we can change the behaviour of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.*

There are many reasons why you should develop code with TDD, but this is the most important. The way I like to describe it is with the analogy of nailing down a floor in the dark. You've done one end of a plank and now you're nailing down the other end. As it's dark you can't see that the first end is slowly being prised up as you bang in the nails on the second end. Code without tests is the same. Without tests you cannot see how a

new change to the code is affecting existing code. As soon as a test passes it instantly becomes a regression test that guards against breaking changes. In our analogy it's like turning the light on or getting a friend to watch the other end of plank: you know when it starts to prize up.

Another advantage of test driven development, especially with unit testing, is that you tend to develop highly decoupled code. When writing units test you generally want to exercise a very small section of code, often a single class or single method, to run the tests against. It becomes very difficult if you have to instantiate a whole slew of other classes or access a file system or database that your class under test relies on (this is slightly different when writing integration tests that often do talk to a file system or database). There are a number of techniques for reducing dependencies on collaborators, mostly including the use of interfaces and Mock Objects. Mock objects are implementations of interfaces for testing purposes whose behaviour is under control of the test, so it can focus solely on the behaviour of the code under test. Writing against interfaces promotes decoupled code for just this reason, collaborators can be mocked out when a class that uses them is tested.

In Working Effectively with Legacy code Michael Feathers has an excellent example of mocking a collaborator.



**Figure 1: Mock Collaborators**

Figure 1 shows a point-of-sale class, called `Sale`, that has a single method, `scan`, which accepts a barcode. Whenever `scan` is called the item relating to the barcode must be displayed, along with its price on a cash register display. If the API for displaying the information is buried deep in the `Sale` class, it is going to be very difficult to test, as we'd have to find some way of hooking into the API at runtime. Luckily the `Sale` class takes a `Display` interface which it uses to display the information. In production an implementation of the interface that talks to the cash register API is used. When testing, a `Fake` object (I'll talk more about fake objects later) is used to sense what the `Sale` object is passing to the display interface.

If you write a test for production code *before* the production code itself, stop developing once the test passes and then refactor to remove any duplication, you end up with just enough code to implement the required feature and no more. It is very important to take time to refactor, thus ensuring that complexity and repetition is reduced. Again if you come to change code in the future and it has been refactored to remove duplication, the chances are you will only have to change the code in one place, rather than several. Code that is developed with TDD, including refactoring, generally yields a better design which stands up well in the face of change.

There have been a number of case studies measuring the benefits of TDD, including "Realizing quality improvement through test driven development" by Nagappan et al [Nagappan]. The key point is that following a study of two teams at Microsoft, one using TDD and one not, it was shown that the team following TDD achieved 75% fewer defects for only 15% more effort. It is true that TDD can take a little longer initially, but generally time is saved by the next release as there are fewer bugs to fix and those that do creep in are easier to find and resolve. In *Software Engineering Economics* [Boehm] Barry Boehm tells us that the cost of fixing a defect increases very quickly the later it is found. This often leads to the paradoxical observation that going "faster" takes longer to get to release as there's a much longer spell of fixing the bugs. By avoiding them being there in the first place you save that effort with a very quick payback. Developers who use TDD will also find that they spend significantly less time in the debugger as their tests highlight where the bugs are.

The best way to find out why TDD is so good is to try it.

## How?

The fundamental steps in test driven development are:

0. Decide what to test
1. Write a test
2. Run all the tests and see the new one fail
3. Write some code
4. Run all the tests and see the new one pass
5. Refactor
6. Run all the tests and see them pass

Repeat.

The first step isn't strictly a TDD step, but it's very important to know what you are going to test before write a test. For acceptance tests this will come from the user story or specification that you have. For integration tests you'll be looking at how two things, such as your data access layer and a database, interact with each other. For unit tests you want to test that the interface does what is expected.

The first real step is to write a test for the new feature you want to implement. Initially the test will not pass. In a lot of cases it wont even compile as you haven't written the code it is going to test yet.

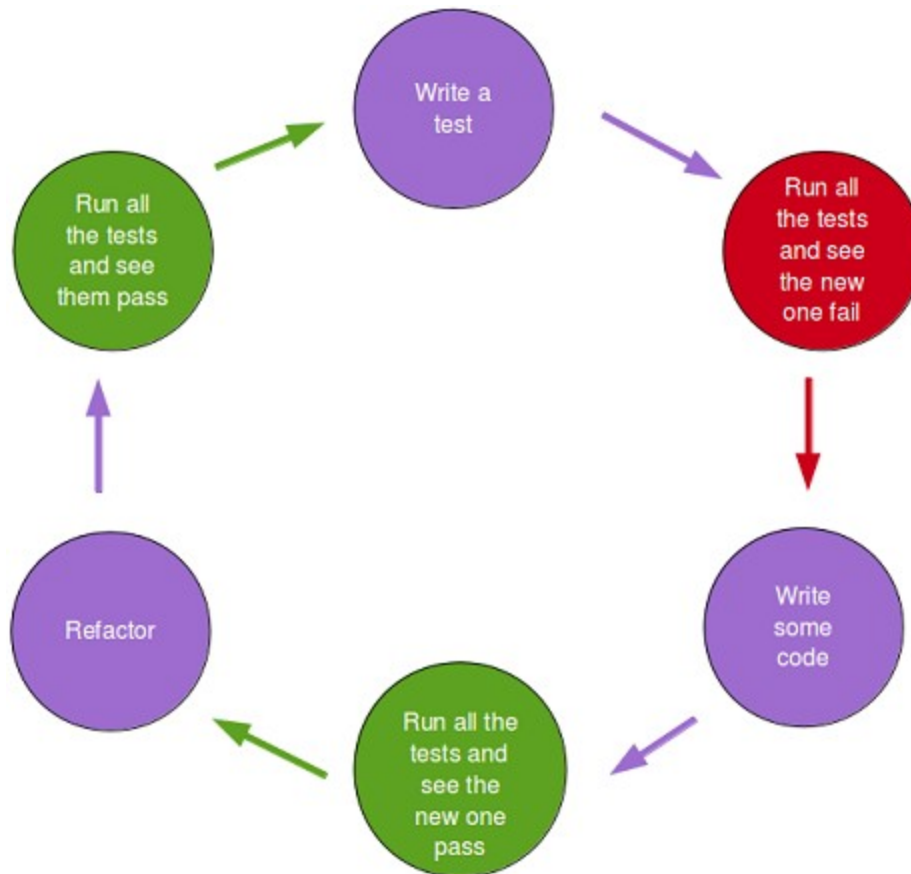
The next step is to write the code to make the test compile, run all the tests and see the new one fail. Then write the code to make the test pass. Write the simplest and most straight forward code you can to make the test pass. Write no more code than it takes to make the test pass. If you write more code than is required you will have code that can change without breaking a test, so you wouldn't know when a change, or in a lot of cases a bug, is introduced, which defeats the goal of TDD. Run the tests again and see them all pass.

The next step is to examine both the test and the production code for duplication and remove it. The final step is to make sure all the tests still pass. Refactoring is about removing duplication and should not change what the code does. Then go back to step 1 and *repeat*. This is also known as the Red/Green/Refactor cycle in reference to the red (tests failed) and green (tests pass) bar that is a feature of many testing frameworks.

Many people think that TDD is only applicable to greenfield projects. In his blog post Implications of the Power Law [Kelly] Allen Kelly tells us that this is not the case:

*The power law [PowerLaw] explains this: because most changes happen in a small amount of code, putting tests around that code now will benefit you very soon. Most of the system is not changing so it doesn't need tests. Of the 1 million lines in a project perhaps only 10,000 changes regularly. As soon as you have 1,000 covered you see the pay back.*

Michael Feathers has written a whole book (the already mentioned Working Effectively with Legacy Code), and it's a big book, of techniques for getting existing code under test. Again, try it for yourself and you will soon see the benefits.



**Figure 2: Test Driven Development Steps**

## An Example

I am going to use a simple program, called DirList, that generates a formatted directory listing for an example of how to develop a project test first, using TDD. One of the prime reasons for choosing it is to show how to deal with programs that rely on external resources and have external effects and so are traditionally hard to test. Here the external resource is the filesystem and the external effect is writing to standard out.

Here's a simple set of stories to describe how the application works:

- When DirList is executed with no arguments, the current directory will be the first message displayed in the output, preceded by "Directory: ".
- When DirList is executed and a path is specified as an argument, it will be the first message displayed in the output, preceded by "Directory: ".
- When DirList is executed without any arguments it will list all the files in the current directory.
- When DirList is executed and a path is specified as an argument it will list all the files in the specified directory.
- When DirList is executed each file will be listed on a separate line along with size, date created and time created.
  - The file name must be left justified and the other details right justified with a single space in between each detail.
  - If the file name would push the whole display line to greater than 40 characters, it must be truncated leaving a single space before the first detail.
  - The date format is dd/mm/yyyy
  - The time format is HH:MM

This is a good example project as it is relatively easy to write acceptance tests that invoke an executable with varying arguments and to examine the output. There is also scope for unit and integration tests, as well as the use of mock objects.

Every new project should start with acceptance tests. The easiest way to start is with a small simple acceptance test that exercises the “system”. In DirList’s case the system is an executable.

### Step 1: Write A Test

Here is a test for the first story, written in Python<sup>1</sup>:

```
class DirListAcceptanceTests(unittest.TestCase):
    def testThatCurrentDirectoryIsFirstMessageInOutputIfNoArgumentsSupplied(self):
        output = Execute(DIRLIST_EXE_PATH, '')
        self.assertEqual(0, output.find("Directory: " + os.getcwd() + "\n" ), output)
```

In this test `DIRLIST_EXE_PATH` is a variable holding the path to the DirList executable and `Execute` is a method that calls the executable with the command line arguments passed in as the second parameter and returns the output. The single assert checks that the output *begins* with “Directory: “ followed by the current working directory and a line feed. The third parameter, `output`, is there so that if the test fails the actual output from the executable is displayed for debugging.

### Step 2: Run All The Tests And See The New One Fail

`Execute` throws an exception if the executable does not exist, so although this test compiles, it fails at runtime as the DirList executable hasn’t been built yet.

### Step 3: Write Some Code

DirList as is a normal C# command line application:

```
namespace DirList
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Building the code into an executable and setting `DIRLIST_EXE_PATH` to its path (e.g. `<bin dir>/DirList.exe`) will get the test a little further. It is important to rerun tests

---

<sup>1</sup> If you’re not familiar with unit testing in Python, Dive Into Python by Mark Pilgrim [Pilgrim] provides an excellent introduction.

after every change, no matter how small. `Execute` can now find the executable and execute it, but there is no output so the assert fails. The simplest way to get the test to pass is to send the current working directory to standard out with "Directory: " prepended and a carriage return appended:

```
static void Main(string[] args)
{
    Console.WriteLine(
        string.Format("Directory: {0}", Directory.GetCurrentDirectory()));
}
```

#### **Step 4: Run All The Tests And See The New One Pass**

Now when the executable is built and the test run, it passes and we have our first passing acceptance test that tests our little system from end-to-end.

#### **Step 5: Refactor**

Normally at this point you would go back and examine the test and the production code and remove any duplication. However, at this early stage there isn't any.

#### **Step 6: Run all the tests and see them pass**

As we haven't had to refactor, there's no need to run the tests again. However, it's always good to get that green bar feeling, so you can run them again anyway if you want to.

#### **Repeat...**

#### **Step 1: Write A Test**

So we move on to write the next test:

```
def testThatSuppliedDirectoryIsFirstMessageInOutputIfPathSupplied(self):
    dir = tempfile.mkdtemp()
    try:
        output = Execute(DIRLIST_EXE_PATH, dir)
        self.assertEqual(0, output.find("Directory: " + dir + "\n" ), output)
    finally:
        os.rmdir(dir)
```

This test is for the second story. It creates a temporary directory, passes its path as a command line argument to `Execute` and then asserts that the directory path is the first message in the output. The temporary directory must be cleaned up by the test so, as

asserts throw when the condition they test for fails, the statement to remove it goes in a finally block. If the temporary directory wasn't cleaned up, running the tests would eventually start to fill your disk up. Creating and cleaning up a specific directory for each test, as opposed to using the same directory each time, means that the tests could be easily run in parallel.

### Step 2: Run All The Tests And See The new One Fail

Of course, this new test fails as DirList sends the current directory to standard output. So we must look at the DirList `Main` method again to find the smallest and simplest change we can make to get the test to pass without breaking the existing tests.

### Step 3: Write Some Code

```
static void Main(string[] args)
{
    if (args.Length == 1)
        Console.WriteLine(string.Format("Directory: {0}", args[0]));
    else
        Console.WriteLine(
            string.Format("Directory: {0}", Directory.GetCurrentDirectory()));
}
```

Here the number of arguments passed in is examined and if there is only one it is assumed to be the path to use, otherwise the current directory is used. This really is the simplest implementation to get the test to pass. We're not taking any error handling into account or allowing for the first argument not to be the path. How DirList behaves under error conditions might be included in future stories.

### Step 4: Run All The Tests And See The New One Pass

After DirList is compiled, the tests must be run and they all pass.

### Step 5: Refactor

Stop! We're not finished yet. Is there some duplication we can refactor away? Yes, there is. The format of the output is repeated in two places. That's easily refactored:

```
private static string getDirToList(string[] args)
{
    if (args.Length == 1)
        return args[0];
    else
```

```
        return Directory.GetCurrentDirectory();
    }

    static void Main(string[] args)
    {
        var dirToList = getDirToList(args);
        Console.WriteLine(string.Format("Directory: {0}", dirToList));
    }
}
```

The duplication is refactored by moving the code that determines which path to use to a new method and formatting the result.

### Step 6: Run all the tests and see them pass

The tests will tell us if a mistake has been made during the refactor, that's what they are for, and running them again reassures us that nothing has broken.

This is an example of the complete TDD cycle. We added a test, watched it fail, wrote some code, watched the test pass, and then refactored to remove duplication.

### Repeat...

The third story is concerned with making sure all the files in the directory being listed are present in the output from DirList. What we need is another test:

#### Step 1: Write A Test

```
def testThatAllFilesInCurrentDirectoryAreIncludedInOutputIfNoArgumentsSupplied(self) :
    output = Execute(DIRLIST_EXE_PATH, '')
    dirList = os.listdir(os.getcwd())
    for fname in dirList:
        if os.path.isfile(os.path.join(os.getcwd(), fname)):
            self.assertTrue(fname in output, "Missing file: " + fname)
```

Again, DirList is invoked without any command line arguments. Then a list of all the directories and files in the current directory is iterated through, the files are identified and their presence checked for in the output.

#### Step 2: Run All The Tests And See The New One Fail

Naturally, the new test fails. To make it pass we need to modify `Main` again.

#### Step 3: Write Some Code

```
static void Main(string[] args)
{
    var dirToList = getDirToList(args);
    Console.WriteLine(string.Format("Directory: {0}", dirToList));

    var dirInfo = new DirectoryInfo(dirToList);
    foreach (var fileInfo in dirInfo.GetFiles())
        Console.WriteLine(fileInfo.Name);
}
```

#### **Step 4: Run All The Tests And See The New One Pass**

Run the tests, which pass.

#### **Step 5: Refactor**

Is there any duplication that can be refactored away? Yes, there is a little. There are two calls to `Console.WriteLine`. The underlying runtime will probably buffer, but that shouldn't be relied upon. The code could be refactored to use a single call with a `StringBuilder`. However this is premature optimisation and could cause `DirList` to use more memory than it really needs.

#### **Step 6: Run all the tests and see them pass**

Following the refactor, rerun the tests to make sure nothing has been broken. When carrying out a significant change, like changing the way text is formatted or output, it is easy to make a simple mistake.

#### **Repeat...**

The next step is to write a test that lists the files in a directory specified as a command line argument. To do this, as well as creating a temporary directory, some temporary files need to be created and checked against the output from `DirList`. I'll leave that as an exercise for the reader. If you do write the test you should find that you do not need to add any code to make it pass. This should make you question whether it is really necessary. In truth it is. The test describes and checks for some behavior that may get broken by a change in the future. Without the test you wouldn't know when the code broke.

You may be thinking that the next step is to return to the acceptance tests and start adding tests for the next story, which describes the format of the output. You could do that, but it would be extremely cumbersome in terms of keeping the tests in line with the changing files in the current directory or creating temporary files of the correct size and

creation date. You would end up with quite brittle tests as every time a file was added to the directory, removed from the directory or its size was modified the test would break. It's far easier to write a set of unit tests instead. That way you can fake the file system and have complete control over the "files".

The formatting story describes one of several possible ways of formatting the output of the directory listing. It is of course possible that other formatting styles may be required in the future, so the obvious implementation is the strategy pattern [Strategy]. At the base of the strategy pattern is an interface that provides a point of variation that can be used to customise the format :

```
public interface Formatter
{
    string Format(IEnumerable<FileInfoMapper> files);
}
```

Another advantage in using an interface is that it makes the formatting strategy easy to mock should we need to for future tests (I'll discuss Mock Objects shortly). Implementations of `Formatter` take a collection of `FileInfoMapper` objects and return a formatted string. You may have been expecting it to take collection of `FileInfo` objects. To write unit tests for the format strategies you need to be able to create a collection of `FileInfo` objects and set the name, size, creation date and creation time. A quick look at the documentation for `FileInfo` [[FileInfo](#)] tells us that we can give a `FileInfo` object a name, but we cannot set the size, creation date or creation time. These properties are set when a `FileInfo` object is populated by examining a real file on the disk. We could write an integration test (I'll describe integration tests shortly too) that creates lots of different files of different sizes and somehow fix the creation date and time, but this would be cumbersome. A better solution is to create an interface that provides the properties we want from the `FileInfo` object:

```
public interface FileInfoMapper
{
    string Name { get; }
    long Length { get; }
    DateTime CreationTime {get;}
}
```

and then write an adapter [Adapter] that extracts the properties from the `FileInfo` object in production and a fake [Fake] object, that implements the same interface, which can be used in testing. Before we look at the formatter strategy implementations, unit tests and fake, we need to satisfy ourselves that an adapter will work. The best way to do this is to write one.

At this point I am switching from writing acceptance tests in Python to writing an integration test in C# with NUnit [NUnit]. Integration tests usually run in the same process as the code they are testing, so using the same runtime, even if not the same language, is helpful. NUnit is a testing framework for .Net, which is usually used for unit testing but is also ideal for integration testing. An integration test tests the the interaction between one or more units. Usually a unit is a class, or the interaction between a unit and an external resource such as a database, file system, network etc. In the next test we're testing the interaction between the `FileInfo` class and the `FileInfoAdapter` class. It just so happens that this also requires interaction with the file system. Integration tests can be developed test first, just like acceptance tests.

I hope that by now you understand the 6 steps of TDD, so from now on I will assume you're following them as the code develops.

```
[TestFixture]
public class FileInfoAdapterTest
{
    private string tempFile;

    [SetUp]
    public void SetUp()
    {
        tempFile = Path.GetTempFileName();
        var file = new StreamWriter(tempFile);
        file.WriteLine("Test Driven Development!");
        file.Close();
    }

    [TearDown]
    public void TearDown()
    {
        File.Delete(tempFile);
    }

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

```

        var fileInfo = new FileInfo(tempFile);
        var adapter = new FileInfoAdapter(fileInfo);
        Assert.That(adapter.Name, Is.EqualTo(fileInfo.Name));
        Assert.That(adapter.Length, Is.EqualTo(fileInfo.Length));
        Assert.That(adapter.CreationTime, Is.EqualTo(fileInfo.CreationTime));
    }
}

```

This NUnit integration test fixture has a pair of setup and tear down methods and a test method. A test fixture is a C# class denoted by the `[TestFixture]` attribute. When NUnit sees the `[TestFixture]` attribute on a class it knows it contains tests that it can run. Setup methods, denoted by the `[SetUp]` attribute, are run before every test method in the test fixture. Test methods are denoted by the `[Test]` attribute and contain tests. Tear down methods, denoted by the `[TearDown]` attribute are run after every test method.

`FileInfoAdapterTest` creates a temporary file, with some content, in the setup method before each test. It uses the temporary file to create a `FileInfo` object and uses it to create a `FileInfoAdapter` and then asserts each of the properties against the `FileInfo` object in the test method. Finally it deletes the temporary file in the tear down method. I have said a couple of times now that creating files on disk for the purposes of testing is cumbersome and it is. However, in this scenario we only need a single file that is easily created and destroyed. This test won't even compile, let alone pass, so we need to write some code to make it pass.

As it was so simple I went ahead and wrote the final implementation of `FileInfoAdapter` straight away, rather than in small steps.

```

public class FileInfoAdapter : FileInfoMapper
{
    private readonly FileInfo fileInfo;

    public FileInfoAdapter(FileInfo fileInfo)
    {
        this.fileInfo = fileInfo;
    }

    public string Name
    {
        get { return fileInfo.Name; }
    }
}

```

```

public long Length
{
    get { return fileInfo.Length; }
}

public DateTime CreationTime
{
    get { return fileInfo.CreationTime; }
}
}

```

This is perfectly acceptable. You do not have to go through painstakingly slow steps of implementation. If you can see the solution and you have the tests written, just implement it.

Now we're ready to write the unit tests for the formatters. As was hinted at earlier, a unit test is a test that tests a single unit that does not interact with other units or resources. It may, however, interact with mocked dependencies as we'll see shortly. Here's the first test:

```

[Test]
public void testThatAllDetailsArePresentInFormat()
{
    const string NAME = "file.txt";
    const long LENGTH = 10;
    var NOW = DateTime.Now;

    var files = new List<FileInfoMapper> { BuildFileInfoMapper(NAME, LENGTH, NOW) };
    var formatter = new DetailsFormatter();
    var output = formatter.Format(files);

    Assert.That(output, Is.StringContaining(NAME), "Name");
    Assert.That(output, Is.StringContaining(LENGTH.ToString()), "Length");
    Assert.That(output, Is.StringContaining(NOW.ToShortDateString()), "Date");
    Assert.That(output, Is.StringContaining(NOW.ToShortTimeString()), "Time");
}

```

This test simply checks that all the required details are present in the output. The only part of the test that is not shown is the `BuildFileInfoMapper` method which is defined as:

```

private static FileInfoMapper BuildFileInfoMapper(
    string name, long length, DateTime creationTime)
{
    return new FakeFileInfoMapper(name, length, creationTime);
}

```

}

It's just a method that creates a `FakeFileInfoMapper` from the arguments supplied.

```
private class FakeFileInfoMapper : FileInfoMapper
{
    private readonly string name;
    private readonly long length;
    private readonly DateTime creationTime;

    public FakeFileInfoMapper(string name, long length, DateTime creationTime)
    {
        this.name = name;
        this.length = length;
        this.creationTime = creationTime;
    }

    public string Name { get { return name; } }

    public long Length { get { return length; } }

    public DateTime CreationTime { get { return creationTime; } }
}
```

A fake is a type of mock object that implements a dependency for testing purposes. A mock object is a test class that replaces a production class for testing purposes. A mock object is usually a simpler version of the production class used to help break a dependency of the class under test or used to sense how the class under test interacts with the production version of the mock. In this case `FakeFileInfoMapper` is a `FileInfoMapper` that allows us to specify `FileInfo` details without having to create a real `FileInfo` object or a file on disk. You'll also notice that the asserts in the test all have messages:

```
Assert.That(output, Is.StringContaining(NAME), "Name");
Assert.That(output, Is.StringContaining(LENGTH.ToString()), "Length");
Assert.That(output, Is.StringContaining(NOW.ToShortDateString()), "Date");
Assert.That(output, Is.StringContaining(NOW.ToShortTimeString()), "Time");
```

Some people believe that a test method should only have a single assert. In a lot of cases this is a very good idea as it helps track down exactly where any failure is more easily. It could easily be argued, for that very reason, that you should have separate test methods for asserting that name, length, date and time are all present in the output string. In most cases I feel that assert messages should not be used either. They're like

comments and we all know that people write (usually unnecessary) comments describing code, later the code gets changed, but the comment is not updated making it invalid and often misleading. That risk is present here, but the advantage of having a single concise, clear test with multiple asserts that are easily identified in failure by the short message, outweighs the risk. My general rule is to have each test method only test one thing, except in circumstances like this where pinpointing the cause of a failure can be easily identified with a message. Currently the test won't build, let alone pass as there is no `DetailsFormatter` class, so we need to write it:

```
public class DetailsFormatter : Formatter
{
    public string Format(IEnumerable<FileInfoMapper> files)
    {
        return "";
    }
}
```

The test will now build, but the test will not pass as the `Format` method returns a blank string and the first assert is expecting a string with a file name in it. The simplest implementation to get it to pass looks like this:

```
public string Format(IEnumerable<FileInfoMapper> files)
{
    var output = new StringBuilder();

    foreach(var file in files)
        output.Append(string.Format("{0} {1} {2} {3}",
                                    file.Name,
                                    file.Length,
                                    file.CreationTime.ToShortDateString(),
                                    file.CreationTime.ToShortTimeString()));

    return output.ToString();
}
```

Looking at the test you could argue that creating a `StringBuilder` and iterating through all the `FileInfoMapper` objects is not the simplest way to get the test to pass. The alternative is to just get the first `FileInfoMapper` from the enumerable, as we know there is only one, and return a string built from that. However the code to get the first item from an enumerable is quite verbose and it's easier to write the `foreach` expression, especially knowing that we'll need it later. If we return from the `foreach` we need a second return at the end of the method to keep the compiler happy, so it's

just as easy to use a `StringBuilder`, again knowing that we'll need it later when we have multiple `FileInfoMapper` objects. This thinking ahead and implementing slightly more functionality than is strictly necessary could be considered "gold plating" and goes against the TDD principle of not implementing any more functionality than is needed to make the test pass. However, TDD is not a straight jacket and in a few cases a little more functionality is ok. The second test asserts that the output is of the correct length:

```
[Test]
public void testThatFormatIsCorrectLength()
{
    const string NAME = "file.txt";
    const long LENGTH = 10;
    var NOW = DateTime.Now;

    var files = new List<FileInfoMapper> { BuildFileInfoMapper(NAME, LENGTH, NOW) };
    var formatter = new DetailsFormatter();
    var output = formatter.Format(files);

    Assert.That(output.Length, Is.EqualTo(40));
}
```

The test builds, but it does not pass as the string returned from the `Format` method is not long enough. Also there is a lot of duplication with the previous test. Duplication in test code is bad for the same reason it's bad in production code. If you have to change the duplicated code, you have to change it everywhere it's duplicated. If you've refactored the duplication away, you only have to make the change in one place. We'll look at that in a moment once we've got the test to pass. The easiest way to get the test to pass is just to pad the output:

```
return output.ToString().PadRight(40);
```

Of course this doesn't give the the final format that we want, but it's enough to get *this* test to pass. I don't like magic numbers. Magic numbers are numbers or string literals in the code that don't clearly explain what they are. Here the number `40` represents the row length. It could be changed to a static member variable, but it makes more sense to pass it in through the constructor in case it's ever changed in the future. It also makes the test more expressive.

```
public class DetailsFormatter : Formatter
{
    private readonly int rowLength;
```

```
public DetailsFormatter(int rowLength)
{
    this.rowLength = rowLength;
}

public string Format(IEnumerable<FileInfoMapper> files)
{
    ...
    return output.ToString().PadRight(rowLength);
}
}

private const int ROW_LENGTH = 40;
...

[Test]
public void testThatFormatIsCorrectLength()
{
    ...
    Assert.That(output.Length, Is.EqualTo(ROW_LENGTH));
}
}
```

Now that the test is passing we can look at removing duplication by introducing a setup method.

```
private const int ROW_LENGTH = 40;

private const string NAME = "file.txt";
private const long LENGTH = 10;
private DateTime NOW = DateTime.Now;
private List<FileInfoMapper> files;

private Formatter formatter;

[SetUp]
public void setUp()
{
    files = new List<FileInfoMapper> { BuildFileInfoMapper(NAME, LENGTH, NOW) };
    formatter = new DetailsFormatter(ROW_LENGTH);
}

[Test]
public void testThatAllDetailsArePresentInFormat()
{
    var output = formatter.Format(files);
    Assert.That(output, Is.StringContaining(NAME), "Name");
    Assert.That(output, Is.StringContaining(LENGTH.ToString()), "Length");
}
```

```
        Assert.That(output, Is.StringContaining(NOW.ToShortDateString()), "Date");
        Assert.That(output, Is.StringContaining(NOW.ToShortTimeString()), "Time");
    }

    [Test]
    public void testThatFormatIsCorrectLength()
    {
        var output = formatter.Format(files);
        Assert.That(output.Length, Is.EqualTo(ROW_LENGTH));
    }
}
```

Before we go any further the tests must be run again to make sure nothing has been broken. Then it's back to the tests, of which there are at least three more to write.

```
testThatFileNameAtBeginningAndDetailsAtEnd
testThatLongFileNamesAreTruncatedToFitIntoRowLength
testThatMultipleFilesAreDisplayedOnSeperateLines
```

They all involve writing a new failing test, writing the code to make it pass and then refactoring away duplication. These are all steps that we have seen a number of times already, so I'll leave them as exercises for the reader. Therefore all that remains is to integrate the `DetailsFormatter` into `Main`:

```
private const int ROW_LENGTH = 60;

static void Main(string[] args)
{
    var dirToList = getDirToList(args);
    var output = new StringBuilder(string.Format("Directory: {0}\n", dirToList));

    var files = toMappers(dirToList);
    var formatter = new DetailsFormatter(ROW_LENGTH);
    output.Append(formatter.Format(files));
    Console.Write(output.ToString());
}

private static IList<FileInfoMapper> toMappers(string path)
{
    var dirInfo = new DirectoryInfo(path);
    var files = new List<FileInfoMapper>();

    foreach (var fileInfo in dirInfo.GetFiles())
        files.Add(new FileInfoAdapter(fileInfo));
    return files;
}
```

and rerun the acceptance tests. They pass! Although they didn't pass the first time I ran them. One of the files in the current directory was so long its name got truncated and appeared to be missing, therefore failing one of the tests. This was easily fixed by increasing `ROW_LENGTH`, rebuilding and running the acceptance tests again. It could be argued that this breaks the original specification and that the test should be updated instead. However, I prefer to look at this as the TDD process highlighting a flaw in the original specification and making a change to it. Agreed by the relevant stakeholder of course.

## Finally

In this whirlwind tour of Test Driven Development I have discussed the what, why and hows. We've been through a very simple, yet all encompassing example of developing an application test first from the outside in. From acceptance tests that run outside the system to unit and integration tests that run on the internal units of the system. I hope it has encouraged you to try TDD for yourself and to read the books I have mentioned, and others, for a deeper look.

## Acknowledgments

My thanks go to Allan Kelly for suggesting that I write a piece on test Driven Development. I've thoroughly enjoyed it. To Chris O'Dell for thorough review and not being scared to tell me when she thinks I'm wrong and to Steve Love, Roger Orr, Matthew Jones and Ric Parkin for review and encouragement. To Rachel Davies for the description of acceptance testing and to Caroline Hargreaves for the diagrams.

## References

[Beck] Test Driven Development by Kent Beck. Addison Wesley. ISBN: 978-0321146533

[Freeman] Growing Object Orientated Software Guided by Tests by Steve Freeman & Nat Pryce. Addison Wesley: ISBN: 978-0321146533

[Davies] When To Write Story Tests by Rachel Davies: <http://agilecoach.typepad.com/agile-coaching/2011/07/when-to-write-story-tests.html>

[Feathers] Working Effectively with Legacy Code by Michael Feathers. Prentice Hall. ISBN: 978-0131177055

[Kelly] Implications of the Power Law by Allan Kelly:  
<http://allankelly.blogspot.com/2008/03/implications-of-power-law.html>

[PowerLaw] The Power Law: [http://en.wikipedia.org/wiki/Power\\_law](http://en.wikipedia.org/wiki/Power_law)

[Nagappen] Realizing quality improvement through test driven development: results and experiences of four industrial teams by Nachiappan Nagappan & E. Michael Maximilien & Thirumalesh Bhat & Laurie Williams: [http://research.microsoft.com/en-us/groups/ese/nagappan\\_tdd.pdf](http://research.microsoft.com/en-us/groups/ese/nagappan_tdd.pdf)

[Boehm] Software Engineering Economics by Barry W. Boehm. Prentice Hall. ISBN: 978-0138221225

[Pilgrim] Dive Into Python: <http://diveintopython.org/>

[NUnit] NUnit .Net Testing Framework: <http://www.nunit.org/>

[Facade] The Facade Design Pattern: [http://en.wikipedia.org/wiki/Facade\\_pattern](http://en.wikipedia.org/wiki/Facade_pattern)

[Strategy] Strategy Design Pattern: [http://en.wikipedia.org/wiki/Strategy\\_pattern](http://en.wikipedia.org/wiki/Strategy_pattern)

[FileInfo] `FileInfo` documentation: <http://msdn.microsoft.com/en-us/library/system.io.fileinfo.aspx>

[Adapter] Adapter Design Patter: [http://en.wikipedia.org/wiki/Adapter\\_pattern](http://en.wikipedia.org/wiki/Adapter_pattern)

[Fake] Fake Objects: [http://en.wikipedia.org/wiki/Mock\\_object](http://en.wikipedia.org/wiki/Mock_object)

Source code: <http://paulgrenyer.net/dnld/DirList-Abridged-0.0.1.zip>