78

Unit Testing and Test-Driven Development with FlexUnit

This chapter covers the practice of unit testing and the *best*-practice of test-driven development, using FlexUnit, the open source unit testing framework for Flex. When practiced together, these techniques are invaluable for developers, helping them to write more robust and better designed software. Bugs can be caught early, before entering production, when the task of fixing is more costly.

Overview

Unit testing is the practice of testing the small pieces of functionality that make up an application to ensure they operate correctly. These small pieces are the *units* and they are tested to ensure that each one satisfies its contract by returning the expected results, handling errors appropriately and collaborating properly with dependent interfaces. If each unit of a system is deemed to be correct, then we may perhaps infer that the system as a whole is correct. In reality, such a claim would be presumptuous, but unit testing does provide many benefits and reassurances.

Test-driven development is a technique that takes unit testing to its logical extreme, by advocating that unit tests are written first off, before the functionality that will actually be tested has even been implemented. Unsurprisingly, the unit test will initially fail, since the functionality has not been written, but when it does succeeds, the developer knows that their code is correct. This process can seem back-to-front to newcomers, but its benefits can be quite remarkable, as this chapter aims to demonstrate.

This chapter covers both unit testing and test-driven development in detail, beginning with a discussion of the main benefits, before moving on the examine the open-source FlexUnit library available for unit testing Flex applications. Those readers already familiar with unit testing in other programming languages may wish to skip to the second half of the chapter, which contains practical examples and further topics, such as mock objects and event testing. The examples in this chapter are based on a simple video portal application.

Why Write Tests?

It is worth spending a little time on motivational talk, since we developers often prefer to hack away in darkened rooms, oblivious to the bugs creeping into our code. So, why write unit tests and *why on earth* write them before the code to be tested?

Prevent Bugs in the First Place

A simple mistake in an algorithm can bring a beautiful application crashing to the ground. Perhaps an error message will be displayed or maybe the client will just stop responding. In any case, the user is disheartened and deterred from using the application again. The cost of fixing such a problem can be large, since the developer who wrote the original code may have jumped ship, and the fix itself could have unintentional consequences. By writing unit tests during development, many bugs like these can be caught before they have the chance to propagate in production code. This produces more robust, higher-quality software and reduces development time and costs.

Build the Confidence to Refactor

Making changes to a large application can be hair-raising. There could be hundreds or thousands of classes, each with their own properties and functions and the documentation may be scarce or out-of-date. Understanding the full implications of a small change in one part of the system is a task fit for a computer and not a human. By writing unit tests and collecting these together to form test suites that can be run easily and repeatedly, developers are given the confidence to refactor and extend the system. They will be informed quickly of any side-effects of their changes by a failing test case, allowing them to resolve the problem before committing new code.

Drive Improvements in Design

The process of writing the test case first, before the functionality under test has been implemented, prompts the developer to consider its behavior. In effect, by writing the test, the developer is trying out the API and will quickly notice if it is poorly designed. With the conventional approach of coding first and testing later, time is more likely to be wasted coding something with a design flaw that will need to be scrapped or revised later. So test-driven development drives improvements in design and reduces unnecessary development effort.

Write Testable Software

In order to write a unit test, we must devise an API that can be tested. So unit testing encourages us to design classes that can be easily tested. An analogy can be drawn with the manufacturing industry, where consumer products often incorporate design features that allow quality assurance tests to be performed automatically. More simplistically, think of a ball-point pen with transparent casing. That design feature allows writers to see when the ink has run out. Class design decisions should be influenced by similar considerations of testability.

It is common to write a class that creates objects of another class internally and then collaborates with them. For instance, a Cairngorm command may create a business delegate object and call a function on it. Test-driven development tends to promote more loosely-coupled designs, where collaborations take place through interfaces rather than concrete classes. These interfaces can be simulated or *mocked* for testing purposes, so the collaborating class can be tested in isolation without dependencies. This approach is known as inversion-of-control or dependency-injection.

Happy Developers and Effortless Documentation

There are many more benefits to test-driven development discussed in various places. For instance, apparently we developers are simple creatures who receive a psychological boost every time we see a test success, thus sustaining long hours at the keyboard! In seriousness, test cases are a good measure of progress and they provide reassurance that functionality has been implemented correctly. Test cases also provide a form of implicit developer documentation, describing in readable code the expected behavior of the classes under test.

Technology and Terminology

This section introduces the FlexUnit library and summarizes the basic terminology of unit testing, in preparation for the detailed examples that follow.

The FlexUnit library

FlexUnit is the most established unit testing framework available for Flex applications. It provides everything that is needed to begin writing unit tests and practicing test-driven development. FlexUnit was based on JUnit 3 in Java, though a few features have been tailored specifically for Flex. The latest version includes an elegant new client for running unit tests and interpreting their results, known as the *Test Runner* and pictured in Figure 78-1.

Figure 78-1: The new FlexUnit Test Runner

Unit Tests, Test Cases and Test Suites

In FlexUnit, a *unit test* is a function that tests the behavior of a class in some way. By convention, a unit test focuses on a small aspect of behavior, such as the operation of function under certain conditions. There may be multiple unit tests for a single function that each test different aspects of its behavior, such as its response to normal parameters and boundary conditions as well as its error handling.

Multiple unit tests are grouped together to form *test cases*, and test cases are then aggregated to form *test suites*. These test suites are built into a composition containing all the test cases and unit tests for a project and this is known as the *All Tests* suite. Figure 78-2 shows a composition of unit tests, test cases and test suites.

Figure 78-2: A composition of test suites, test cases and individual tests

Running Tests

The FlexUnit test runner executes a test suite by recursively processing each test case and executing all the unit tests. When a unit test is performed, the outcome is recorded and presented by the test runner. The outcome of a unit test may be *success,* in which case the test completed without interruption; *failure*, in which case some specific but incorrect behavior was detected by the test; or *error*, in which case a problem unanticipated by the unit test occurred.

Test-Driven Development by Example

The most benefit is derived from unit testing when combined with test-driven development, so the examples in this section do precisely that. The example scenario involves creating a class for filtering video objects by keyword. The test cases will be created before the filter class, in order to reflect upon its design and ensure its testability, before expending effort on its implementation.

Preparing for Unit Testing

A Flex Builder or standalone project must be prepared for unit testing. This involves adding the FlexUnit library to the project build path and creating a source folder for storing test cases, test suites and other related classes.

Download FlexUnit

FlexUnit contains the base classes for implementing and running unit tests. It is distributed as a ZIP archive that can be downloaded from:

* `http://code.google.com/p/as3flexunitlib`

The distribution contains a `flexunit` folder with the following contents:

* `licence.txt` - licensed under the terms of the New BSD License
* `bin/flexunit.swc` - the unit testing framework library
* `docs/` - the AS3 documentation
* `src/` - the source code, should you need it

Add the Library

To add the FlexUnit SWC to the build path of a Flex Builder project, copy the `flexunit.swc` file into the `lib` directory of the project. The build path can be viewed through the Project Properties dialogues to verify that the `flexunit.swc` is included in the list of libraries.

If the command-line compiler is being used instead, the `flexunit.swc` must still be added to the build path. This is typically achieved through an Ant build script or compiler argument.

Create the Test Folder

It is best to story unit tests in a different source folder from the main project code, but with a parallel package structure. This prevents the test cases from cluttering up the main source folder, but still makes them easy to locate. It also gives them access to internal properties of the classes under test, which can sometimes be useful.

To create a `test` folder in Flex Builder, bring up the project properties dialogue and select:

* Flex Build Path > Source Path > Add Folder > Browse > New Folder

Store test cases in a separate source folder with the same package structure as the main source folder.

Creating a Test Case

Imagine building a video portal that needs to filter videos based on a keyword. The filtering will simply match a keyword with the title of each video. This is already enough information to begin unit testing!

In FlexUnit, unit tests are specified in classes that extend `flexunit.framework.TestCase`. A unit test is defined in its own function, with a descriptive name starting with `test`. Here is an example of an initial test case for a class that will filter videos by keyword.

```
package proflex.model.video
{
    import flexunit.framework.TestCase;

    public class VideoKeywordFilterTest extends TestCase
    {
        // The object that will be tested
        private var filter:VideoKeywordFilter = new VideoKeywordFilter();

        public function testFiltersVideoWithMatchingTitle():void
        {
            fail("Unit test not yet implemented");
        }
    }
}
```

The class is named `VideoKeywordFilterTest` because it is going to test a class called `VideoKeywordFilter`. It is conventional to name a test case by appending `Test` to the end of the name of the class that will be tested.

At the top of the class, a private variable named `filter` is defined to reference the object that will be tested. Since the `VideoKeywordFilter` class doesn't actually exist yet, an empty class with this name must be created to successfully compile the test case.

A `testFiltersVideoWithMatchingTitle()` unit test is defined to test the behaviour of the `VideoKeywordFilter` class when filtering videos based on their titles. The implementation invokes the `fail()` function, inherited from the `TestCase` base class. This is to indicate that the filtering functionality has not yet been implemented. If the function body was left empty instead, the test would misleadingly report a success.

Name test cases `TestMyClass` where `MyClass` is the class under test.

At Give unit test functions descriptive names that begin with `test`.

Call the `fail()` function from any unimplemented test cases.

Assembling a Test Suite

Before the simple test case can be run, a test suite must be assembled. Recall that a test suite collects together test cases to form a composition that can be processed by the FlexUnit test runner. In some programming languages, there is tooling to automatically generate test suites at runtime, but in Flex they are currently written by hand. This is straightforward, if somewhat tedious.

A test suite is defined in a class that extends `flexunit.framework.TestSuite`. In most cases, a test suite contains only a constructor, which invokes the inherited `addTest()` function repeatedly to assemble the test suite. A simple test suite that adds the `VideoKeywordFilterTest` is shown below:

```
package proflex.model.video
{
    import flexunit.framework.TestSuite;

    public class AllVideoTests extends TestSuite
    {
        public function AllVideoTests()
        {
            addTest(new TestSuite( VideoKeywordFilterTest));
            // add further test cases here
        }
    }
}
```

Note that the test case class must be wrapped in a `TestSuite` object before being passed into the `addTest()` function.

Failing the Test!

With the test case written and the test suite assembled, the test can finally be run. Of course, a test failure is expected initially, since no filtering functionality has actually been written. This step is psychological; the disappointment of an initial failure makes a later success more rewarding! It also reassures us that the foundations are in place for test-driven development to begin in earnest.

The tests contained in a test suite can be run using the `TestRunnerBase` component of FlexUint. This provides a user interface for viewing the progress of the tests as they run and interpreting the results afterwards. To use the `TestRunnerBase` component, a new MXML application should be created in the main source folder (not the test folder, since Flex Builder does not support this). It is conventional to name the application `TestRunner.mxml`. An example is given below.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:flexunit="flexunit.flexui.*"
    creationComplete="creationCompleteHandler()">

    <mx:Script>
        <![CDATA[
            import proflex.model.AllVideoTests;

            private function handleCreationComplete() : void
            {
                testRunner.test = new AllVideoTests();
```

```
                    testRunner.startTest();
                }
            ]]>
        </mx:Script>

        <flexunit:TestRunnerBase id="testRunner" width="100%" height="100%"/>

    </mx:Application>
```

With all the luck in the world, this test case would still fail, but do not despair...

Figure 78-3: The test initially fails since no functionality has been implemented.

Prepare, Invoke, Assert

The skeletal test case needs to be fleshed out, and in doing so, the design of the `VideoKeywordFilter` class can be considered more deeply. A small design decision has in fact already been taken: the encapsulation of the video filtering behaviour inside a class of its own.

A unit test usually has three stages:

1. prepare – setting up the objects involved in the unit test.
2. invoke – calling the function that is being tested.
3. assert – confirming that the correct behaviour has taken place

The preparation code is usually nothing more than setting a few properties so they are in the correct state for the test to begin. We will see later how preparatory code that is relevant to more than one test can be moved into a `setUp()` function. The invoke stage involves calling the function under test, so that its implemented behaviour takes place. The assert stage is then used to determine whether or not the expected behaviour of the class actually took place.

Assert Functions

FlexUnit includes a collection of assert functions for this purpose. These are defined in the `flexunit.framework.Assert` class and inherited by `flexunit.framework.TestCase`, so they are available in any test case that is created.

- `assertEquals( failureMessage, expectedValue, actualValue )`
- `assertFalse( failureMessage, actualValue )`
- `assertNotNull( failureMessage, actualValue )`
- `assertNotUndefined( failureMessage, actualValue )`
- `assertNull( failureMessage, actualValue )`
- `assertStrictlyEquals( failureMessage, expectedValue, actualValue )`
- `assertTrue( failureMessage, actualValue )`
- `assertUndefined( failureMessage, actualValue )`

Examining Actual Values

An assert function examines an `actualValue` to determine whether it satisfies a condition. For example, the `assertEquals()` function could be used to determine whether the expected value of `2` was actually returned by an `addNumbers()` function call when called with the parameters `1` and `1`. Or the `assertNotNull()` function could be used to confirm that a `getName()` function does not return `null`. When an assert function is unsatisfied, an error is deliberately thrown by FlexUnit to indicate a unit test failure. This error is handled by the FlexUnit test runner and presented to the user.

It is important to note the ordering of the `expectedValue` and `actualValue` parameters on the equality assertions. These can easily be muddled up, which causes the test runner to display misleading information, reporting that the expected result occurred rather than the actual result.

Ensure the expected value is specified before the actual value when using the `assertEquals()` and `assertStrictlyEquals()` assertions.

Failure Messages

It is important to note the names and ordering of the parameters on the assertion functions, since these are not apparent from the FlexUnit ASDoc. The initial `failureMessage` parameter is optional, but its use is strongly recommended. This message is displayed by the FlexUnit test runner whenever a unit test fails, so it should guide the developer towards fixing the broken test. The `failureMessage` parameter also enhances the self-documenting quality of a test case.

Use the `failureMessage` parameter to provide a concise yet descriptive failure message.

```
assertEquals(
    “Video title was not properly updated”,
    expectedTitle,
    video.title );
```

Fleshing Out the Test Case

The test case can now be implemented, helping to drive out the design of the video filtering functionality. Since the class needs to filter based on keyword, there needs to be a way to specify that keyword. The simplest solution would seem to be a `keyword` setter on the `VideoKeywordFilter` class. With the keyword set, the filtering itself can be performed through a `filterVideo()` function that takes a `Video` parameter and returns `true` if the video should be filtered or `false` otherwise.

These decisions lead to the following unit test:

```
public function testFilterVideoWithMatchingTitle() : void
{
    var video:Video = new Video(“Flex, Lies and Videotape”);

    filter.keyword = "Flex";

    var filtered:Boolean = filter.filterVideo(video);
```

```
        assertTrue(
            "The filter should have detected a match in the video title",
            filtered );
    }
```

The preparation stage involves creating a new `Video` object and setting a `keyword` property on the filter object (the `VideoKeywordFilter` object being tested). The title of the video contains the word “Flex” which is also used as the keyword property. The invoke stage is simply a call to a `filterVideo()` function on the filter object, and the assert stage involve checking that the video was correctly filtered using the `assertTrue()` function.

Before now, the `VideoKeywordFilter` was nothing more than an empty class. In the process of writing the unit test, a simple design for keyword and filtering functionality has emerged. And this design already has the merits of usability and testability, since we were able to write the unit test with ease.

Write unit tests first to guide the design of the class under test, ensuring usability and testability.

Some Real Coding

In order to compile and run the test case, the `keyword` setter and `filterVideo()` function need to be added to the `VideoKeywordFilter` class. At this stage, the initial implementation can also be written.

```
package proflex.model.video
{
    public class VideoKeywordFilter
    {
        private var _keyword:String;

        public function set keyword(value:String):void
        {
            _keyword = value;
        }

        public function filterVideo(video:Video):Boolean
        {
            return video.title.indexOf(_keyword) >= 0;
        }
    }
}
```

The value passed to the keyword setter is backed by the private `_keyword` property and the `filterVideo()` function works using the `indexOf()` function from the `String` class, called on the `title` property of the `Video`.

The project can now be rebuilt and the test runner launched again. Cross your fingers...

Figure 78-4: Success at last!

Happy and Sad Paths

The first test case tests "the happy path" through our filter class. That is, it tests the class under normal conditions, by passing a sensible parameter to the `filterVideo()` function and confirming that the expected behaviour took place. However, this is not sufficient to consider the class complete. The test needs to be strengthened to consider the corner cases and boundary conditions.

Strengthen the Tests

A corner case is an unusual circumstance that may need to be handled specially, while a boundary condition is a value close to, on the edge, or just beyond the expected range of a parameter. A number of additional unit tests for the `VideoKeywordFilter` class are given below.

```
public function testDoesNotFilterNullVideo():void
{
    filter.keyword = "Flex";

    var filtered:Boolean = filter.filterVideo(null);

    assertFalse("A null video should not be filtered", filtered);
}

public function testDoesNotFilterVideoWithoutTitle():void
{
    var video:Video = new Video(null);

    filter.keyword = "Flex";

    var filtered:Boolean = filter.filterVideo(video);

    assertFalse(
        "A match should not have been found with a video that has no title.",
        filtered);
}

public function testFiltersAnythingWhenKeywordIsNull():void
{
    filter.keyword = null;

    var video:Video = new Video("Flexas Chainsaw Massacre");

    var filtered:Boolean = filter.filterVideo(video);

    assertTrue("All videos should match a null keyword", filtered);
}
```

Note first of all that the new tests have descriptive names. If the "test" prefix is removed, they can be read as descriptions of the intended functionality of the `VideoKeywordFilter` class. For example, the `VideoKeywordClass` "does not filter [a] video without [a] title".

Strengthen the Code

When the new, strengthened test suite is run again, the results reveal that the implementation of `filterVideo()` function is not yet robust. In fact, the corner cases of a `null` video and a `null` title have not been handled at all, resulting in errors that are reported by the test runner. These errors can be overcome by adding a pair of entry conditions to the `filterVideo()` function.

```
public function filterVideo(video:Video):Boolean
{
    if (hasNoKeyword()) return true;
    if (isInvalid(video)) return false;

    return video.title.indexOf(_keyword) >= 0;
}

private function isInvalid(video:Video):Boolean
{
    return video == null || video.title == null;
}

private function hasNoKeyword():Boolean
{
    return _keyword == null;
}
```

When the `filterVideo()` function is called, a check is made to ensure there is a keyword. If not, then all videos should be filtered, so `true` is returned. Next the validity of the video is checked to ensure it is not `null` and has a title. If the video is invalid, then it cannot be filtered so `false` is returned. If processes reaches beyond these two entry conditions, then the existing algorithm is used to detect a match between the video title and keyword.

Enough is Enough

Since a unit test case is normally concerned with the behaviour of a specific class, it is usually easy to identify the corner cases and boundary conditions that need to be tested. However, it can be counterproductive to be obsessive about this. Instead of attempting to test every possible way of invoking any piece of functionality, prioritize the most plausible error conditions. Reflect on the complexity and importance of the behaviour and decide what level of testing is required.

The Virtuous Cycle

When practicing test-driven development, the cycle of strengthening the test cases, detecting a weakness in the software, then responding by strengthening the code leads to ever more robust software. Even when bugs are reported in production software, they are fed back into the process. The first step is always to write a new unit test that reproduces the bug. Then effort can focus on fixing the issue so the new unit test passes. The result is a stronger test suite that will prevent the same bug from occurring again.

Set-up and Tear-down

Although a test case can contain multiple test functions, each one is performed afresh, using a new instance of the test case. In other words, the test case class is instantiated, the first test is performed, then the instance is made eligible for garbage collection. This process is repeated for the remaining tests.

Figure 78-5: The life cycle of a test case

There are two intermediate stages in the life-cycle of a test case: *set-up* and *tear-down*. The `TestCase` base class defines `setUp()` and `tearDown()` functions that may be overridden to perform initialization and clean-up before and after every test. The `setUp()` function is often used to prepare a number of member variables required by the unit tests, and these are known as the *test fixture*. The `tearDown()` function is used less often, for performing clean-up duties, so the outcome of one test does not interfere with another.

The `VideoKeywordFilterTest` can be refactored to take advantage of the set-up phase and remove duplicate code. There is no need for tear-down here, since the test will already clean-up after itself by way of the garbage collector. Tear-down is rarely needed, but one exception is any test that alters the state of a singleton, which may cause side-effects in subsequent tests if not properly reversed. The refactored test case including a `setUp()` implementation is given below.

```
public class TestVideoKeywordFilter extends TestCase
{
    private var filter:VideoKeywordFilter;
    private var video:Video;

    override public function setUp():void
    {
        filter = new VideoKeywordFilter();
        filter.keyword = "Flex";
        video = new Video("Flex, Lies and Videotape");
    }

    override public function tearDown():void
    {
        // no need for tear down here
    }

    public function testFilterVideo():void
    {
        var filtered:Boolean = filter.filterVideo(video);

        assertTrue(
           "The filter should have detected a match in the video title",
           filtered);
    }

    public function testFilterNullVideo():void
    {
        var filtered:Boolean = filter.filterVideo(null);

        assertFalse("A null video should not be filtered", filtered);
    }

    public function testFilterVideoWithNullTitle():void
    {
        video.title = null;

        var filtered:Boolean = filter.filterVideo(video);

        assertFalse(
           "A match should not have been found with a video that has no title.",
           filtered);
    }

    public function testFilterWithNullKeyword():void
    {
        filter.keyword = null;

        var filtered:Boolean = filter.filterVideo(video);

        assertTrue("All videos should match a null keyword", filtered);
    }
}
```

Don't write a tearDown() function if the garbage collector will clean-up automatically for you.

If multiple test cases require the same test fixture, refactor this into a common base class and extend this when required.

White Belt in Testing

That brings the example to a close. It may have been a little simplistic, but it is the process that is most important to grasp. Put a test suite and test runner in place at the beginning of a project so that test cases can be written early enough to guide the design. Try out the various assert functions, make good use of failure messages, move preparation code into `setUp()` functions, and always apply descriptive names for test functions to enhance their self-documenting quality. When a project progresses in this way, through development, into production and beyond, its quality will continually improve.

Further Topics

This section provides some further guidance, including tips for organizing tests in larger projects and an example of the mock object testing technique. The section ends with references for more information on continuous integration and event testing, and a high-level overview of two related subjects -- behaviour driven development and test coverage – both on the horizon for Flex and AIR developers.

Organizing Test Suites

For a small project, it may be sufficient to create a single test suite that aggregates all the test cases together. For larger projects, it is better to create a hierarchical composition of test suites, with an `AllTests` suite at the root level. An example of the root node of such a composition is given below.

```
package proflex
{
    public class AllTests extends TestSuite
```

```
        {
            public function AllTests()
            {
               addTest(new AllVideoTests());
               addTest(new AllCommentaryTests());
               ...
            }
        }
    }
```

Each package within the test folder that contains test cases should have its own test suite. For example, a `proflex.model.video` package should have an `AllVideoTests` class.

Removing Dependencies with Mock Objects

When a class interacts with another class, it has a dependency on that class. And when that class interacts with yet another class, a secondary dependency exists. Perhaps the second class then performs some kind of service interaction, so a dependency exists beyond the Flex application into the service layer. This common scenario can make it tricky to unit test the original class, since the test case must understand the dependencies with other classes and external systems.

Dependencies like these introduce complexity and frailty to a test suite, but they can be overcome using a technique known as mock objects. At its most simple, a *mock object* is an object that stands in for another object during testing, in order to eliminate or at least reduce dependencies. The mock object takes the place of the real object, and the unit test simply ensures that the correct actions are performed against the mock object. A mock object sometimes contains a small amount of logic to record the invocation that take place against it for later verification.

In some languages, there are libraries to make mock object testing very easy, by automating the creation of mock objects at runtime. Unfortunately no such library exists for Flex at the time of writing, so mock objects are mostly coded by hand. A detailed example follows.

The Collaborating Class

The `UploadVideoModel` class uploads video files using a delegate to perform the actual uploading. There are two bindable status properties for indicating whether or not an upload is currently in progress and whether or not an error has occurred. This is the class that will later be tested using a mock object.

```
public class UploadVideoModel implements IResponder
{
    [Bindable]
    public var uploading:Boolean = false;

    [Bindable]
    public var error:Boolean = false;

    private var delegate:IUploadVideoDelegate;

    public function UploadVideoModel(delegate:IUploadVideoDelegate)
    {
        this.delegate = delegate;
    }

    public function uploadVideo(videoFile:FileReference):void
```

```
        {
            uploading = true;
            error = false;
            delegate.uploadVideo(videoFile, this);
        }

        public function result(data:Object):void
        {
            uploading = false;
            error = false;
        }

        public function fault(info:Object):void
        {
            uploading = false;
            error = true;
        }
    }
```

The Collaboration Interface

Video uploading is performed through a delegate interface, so the `UploadVideoModel` is decoupled from any specific uploading mechanism. The delegate interface is passed into the `UploadVideoModel` constructor, which is an example of a the *inversion-of-control* or *dependency-injection* pattern.

```
    public interface IUploadVideoDelegate
    {
        function uploadVideo(
            videoFile:FileReference,
            responder:Responder):void;
    }
```

The Real and Mock Implementations

There are two implementations of the `IUploadVideoDelegate` interface. The first is the real implementation, which actually streams the video file to a remote server. The second is a mock implementation that allows us to test the behaviour of the `UploadVideoModel` class without concern for its server-side dependency. The `MockUploadVideoDelegate` contains simple logic to support testing.

```
    public class MockUploadVideoDelegate implements IUploadVideoDelegate
    {
        public var videoFile:FileReference;
        private var responder:IResponder;

        public function uploadVideo(
            videoFile:FileReference,
            responder:IResponder):void
        {
            this.videoFile = videoFile;
            this.responder = responder;
        }

        public function sendResult():void
        {
            responder.result(new ResultEvent(ResultEvent.RESULT));
        }

        public function sendFault():void
        {
            responder.fault(new FaultEvent(FaultEvent.FAULT));
        }
    }
```

The mock defines two public functions for sending result and fault events to the responder, which is passed in to the `uploadVideo()` function. These functions allow the mock delegate to be controlled from within a unit test, instead of being governed by the responses sent back from the server. They facilitate test cases for happy and sad paths through the `UploadVideoModel`.

Unit Testing with the Mock

The test case that makes use of the mock delegate is given below. Some preparation is carried out in the `setUp()` function, where the `UploadVideoModel` is constructed. Here the mock delegate is injected into the constructor instead of the real delegate. Two tests are defined, one for testing a successful video upload and the other for testing a failed upload.

```
    public class TestUploadVideoModel extends TestCase
    {
        private var model:UploadVideoModel;
        private var mockDelegate:MockUploadVideoDelegate;
        private var videoFile:FileReference;

        override public function setUp():void
        {
            mockDelegate = new MockUploadVideoDelegate();
            videoFile = new FileReference();
            model = new UploadVideoModel(mockDelegate);
        }

        public function testUploadsVideo():void
        {
            model.uploadVideo(videoFile);
            assertUploadInProgress();
            mockDelegate.sendResult();
            assertUploadComplete();
            assertFalse(
               "An uploading error should not have occurred",
               model.error );
        }

        public function testHandlesFaultWhileUploadingVideo():void
        {
            model.uploadVideo(videoFile);
            assertUploadInProgress();
            mockDelegate.sendFault();
            assertUploadComplete();
            assertTrue(
               "An uploading error should have occurred",
               model.error);
        }

        private function assertUploadInProgress():void
        {
            assertEquals(
               "The video file was not passed into the delegate",
               videoFile,
               mockDelegate.videoFile);

            assertTrue(
               "The uploading should be in progress",
               model.uploading);
        }

        private function assertUploadComplete():void
        {
            assertFalse(
               "The uploading should be complete",
               model.uploading);
        }
    }
```

Note that the above test case has two private functions containing assertions: `assertUploadInProgress()` and `assertUploadComplete()`. These are called from both the `testUploadVideo()` and `testHandlesUploadVideoFault()` tests. It is good practice to move repeated assertions into their own functions in this way, thus preventing duplication.

Move repeated assertions into their own functions with descriptive names.

Mocks and Mockery

Mock objects provide a powerful mechanism for unit testing classes without dependencies on other classes or remote services. This can simplify unit tests, because complex preparation code is not required, while also speeding up the test suite and enabling it to run even when the server-side is down. However, developing mock objects requires effort in Flex. The amount of code required to create a mock should be small, and if not, it may be time to step back and consider a simpler design. If the real dependencies of a class can be prepared easily, then the argument for creating a mock object becomes less convincing. The purist approach of abstracting every dependency and testing each class in complete isolation is likely to be unproductive. In conclusion, master the technique of mock objects, but use it with care, to hide complex dependencies and truly simplify unit tests.

Continuous Integration with Ant and Maven 2

Continuous integration is the process of automating project builds and testing in order to identify and resolve integration issues early. This helps ensure that software remains correct and properly integrated throughout its development. There are several commercial and open source continuous integration servers, including Cruise Control, Team City and Electric Cloud. These servers tend to integrate with source control repositories and execute Ant or Maven scripts in order to perform the builds and run the tests.

Peter Martin, of Adobe Consulting, developed the FlexUnit Ant task and a special Flex test runner for use in build scripts and continuous integration environments. These task and test runner work closely together: the Ant task launches the Flex test runner application, and opens a socket for receiving the test results. The Flex test runner executes the tests and transmits the results to the Ant task through a local network connection.

The Ant task, Flex test runner and an example project can be downloaded from the following blog post:

* http://weblogs.macromedia.com/pmartin/archives/2006/06/flexunit_ant.cfm

A Flex 2 plug-in is also available for Maven 2 that supports compilation, dependency management and FlexUnit testing. The plug-in uses the same Flex test runner as the Ant task. Further details are available from the following URL:

* http://www.servebox.com/foundry/doku.php?id=m2f2plugin

Eventful Test Cases

The events that a class dispatches form an important part of its contract with clients, yet for one reason or another they are seldom tested. Perhaps this is because event declarations are optional in Flex and specified as metadata, or perhaps it is because the FlexUnit assertions were mostly ported from JUnit and not tailored specifically to Flex. In either case, programming mistakes and design flaws in event logic can have far-reaching consequences, effecting dependent event listeners and the classes beyond.

A FlexUnit extension, know as `EventfulTestCase`, is available to make it simple to test event dispatching logic, including Cairngorm event dispatching. Downloads and documentation are available for free from:

* http://blogs.adobe.com/tomsugden/2008/01/post.html

Testing event dispatching logic involves three steps: recording the expected events; performing the function under test; then asserting that the expected events actually occurred. An example test is shown below that ensures a *videoChange* event is dispatched whenever a rating is added to a video.

```
public function testAddRating():void
{
   expectEvent(video, VideoChangeEvent.VIDEO_CHANGE);
   video.addRating(5);
   assertEvents(
       "A change event for the video event should have been dispatched.");
}
```

Test Coverage

Test coverage is the process of analyzing the code that is executed while a test suite runs, in order to understand what code is really being tested and, more importantly, to identify any code that has been missed. A test coverage report can be used to inform the development of new unit tests and to reassure developers and stakeholders that software is thoroughly unit tested. Test coverage is best incorporated into a project build process at an early stage, to encourage developers to build better tests, rather than towards the end when the results are likely to cause alarm!

Alex Uhlmann and Joe Berkovitz both developed test coverage tools for Flex in parallel without knowledge of each others endeavors. Alex made use of the new profiling instrumentation provided by Flash Player 9 to gather line coverage metrics as a test suite ran, while Joe extended the Flex compiler to provide a deeper level of instrumentation, tracking the conditional branch coverage. Joe's solution has now been released on Google Code under the name Flexcover and the pair have begun collaborating on an enhanced test coverage client. For more information about Flexcover, visit the project web page:

* http://code.google.com/p/flexcover

Behaviour-Driven Development

Behaviour-driven development (BDD) was created by Dan North in response to test-driven development. Dan had witnessed confusion and misunderstandings whenever he attempted to teach test-driven development, so he set out to improve matters. What began as a simple rewording of TDD, placing greater emphasis on behaviour than tests, evolved into a framework for end-to-end functional testing.

The first change suggested by BDD was that test function names should be rewritten as sentences in the language of the business domain. This could enhance their self-documenting quality to the point where a pretty-printed list of test function names made sense to business users, analysts and testers. From this change, it became apparent that BDD had the potential to provide a consistent vocabulary, or *ubiquitous language,* for application analysis in general, improving communications between analysts, developers, testers and the business.

JBehave was the first Java framework for BDD that attempted to formalized acceptance criteria into structured scenarios that could be executed automatically. This framework defines various interfaces to represent small parts of these end-to-end scenarios. By implementing these interfaces, and assembling the parts together in different combinations, a comprehensive suite of end-to-end functional tests can be assembled and executed automatically.

At the time of writing, there is no similar BDD framework for Flex. However, some of the best practices of BDD can also be applied to TDD and unit testing. In particular, developers should always strive to write descriptive FlexUnit test cases using the domain language. Perhaps a BDD framework will soon emerge for Flex, but in the meantime, more details can be found at:

* http://dannorth.net/introducing-bdd

Summary

This chapter has tried to explain the benefits of unit testing and demonstrate through examples that it is a straight-forward practice to learn and apply. Unit tests help to verify that the small parts that make up a system are operating correctly, providing the developer with reassurance that their code is correct. Developing a comprehensive suite of unit tests can prevent many bugs from occurring in the first place, while giving developers the confidence to refactor without fear of unintended consequences.

Test-driven development takes unit testing to the extreme and brings with it greater benefits. In particular, it helps to improve software design by prompting developers to consider and use their classes before expending any effort implementing them. This can prevent bad ideas from getting into the code base, while ensuring that classes are testable by design from the outset.

Both unit testing and test-driven development are skills that need to be honed. It is easy to get started but the same pitfalls exist as in normal software development. Care must be taken to write simple, structured and readable tests that don't contain redundancies or duplication. As we develop more sophisticated Flex and AIR applications for the enterprise, testing becomes vitally important to ensure quality. After becoming comfortable with unit testing and test-driven development, many developers will insist upon it.