

# Step-By-Step Unit Testing with ASUnit

## Part Two

By Tim Beynart 4/03/2007

### Introduction

In the previous article, I outlined getting ASUnit running. In this article I will show some unit testing techniques. All the following code is ActionScript 2.

### Example 1: “Asynchronous” Testing

ASUnit provides a test case for loading XML as a local data set. This test case is referred to as “asynchronous”, but this is a misnomer. When the tests are run, they aren’t following a request/response cycle. The tests load a class with an xml file and perform operations to test the efficacy of the data handling. By nature, unit testing involves testing a single method with no dependencies on any other test methods or production methods. In most cases, a Flash programmer is concerned with network activity (i.e. requesting data from a Web Service) when writing asynchronous methods. Compared to Java and the like, Flash is not multithreaded, nor is it capable of direct interactions with databases and other servers. These more complex systems’ testing frameworks do not really apply to ActionScript. ASUnit only provides a test case for loading XML and has no other asynchronous utilities are packaged in the framework.

For this example I am going to create a class called **XMLExample** that parses some XML and exposes the contents through getters. I want to write tests to make sure the class returns the correct results from a known xml file. Since this is Test First development, we need to write our tests before we touch the XMLExample class.

Open the AsUnit.exe again and follow the steps from the previous article to create our test suite. However, this time in step 6 call the class **XMLExample**, and in step 7 select “asynchronous” instead of “standard” in the Create Test Case field. Remember that the class paths AsUnit.exe writes are mangled, so you need to fix the import statements. The resulting **XMLExampleTest** class will look like this:

```
import asunit.framework.TestCaseXml;
import com.example.XMLExample;
import asunit.framework.TestCase;

class XMLExampleTest extends TestCase {
    private var className:String = "XMLExampleTest";
    private var instance:XMLExample;
    private var xmlData:TestCaseXml;

    public function XMLExampleTest(testMethod:String) {
        super(testMethod);
    }
    public function run():Void {
        xmlData = new TestCaseXml("pathToYourXmlFile.xml", this);
    }

    public function onXmlLoaded(node:XMLNode):Void {
```

```

        super.run();
    }

    public function setUp():Void {
        var data:XMLNode = xmlData.cloneNode(true);
        instance = new XMLExample();
    }

    public function tearDown():Void {
        delete instance;
    }

    public function testInstantiated():Void {
        assertTrue("XMLExample instantiated", instance instanceof XMLExample);
    }

    public function test():Void {
        assertTrue("failing test", false);
    }
}

```

Notice the inclusion of **TestCaseXml**. In the **run** method, which is called when the test case is instantiated, an XML document is loaded into the test case instance. This XML is duplicated for each test that is run in **XMLExampleTest** when the **setUp** method is called. This ensures that an intact XML object is available to each test; any operations on the XML in one method will not affect any other.

Open the **AllTests.as** class. Notice that the AsUnit.exe appended a line:

```

class AllTests extends asunit.framework.TestSuite {
    private var className:String = "AllTests";

    public function AllTests() {
        super();
        addTest(new ExampleTest());
        addTest(new XMLExampleTest());
    }
}

```

With each new test class, a constructor should be added to **AllTests**. Remember, **AllTests** is called by the **ExampleTestRunner** class in the **FLA**. **AsUnit.exe** is a timesaver because it takes care of the tedious coding and lets you concentrate on writing tests.

Now compile the **FLA** to make sure everything is working correctly. You will see one failure from the **test ()** method of **XMLExampleTest**, this is the stub test **AsUnit.exe** generates to keep you on your toes.

For this example I am going to write a basic **RSS** parsing class. In true **TDD** spirit, I want to think about the design for a moment then begin with a test.

First off, I need to load some **XML**. In order to use the **asunit xml** test case, I need to provide a local file for the test to load as my control data. I am going to grab a **Flickr.com** **RSS** feed from my favorite bicycle racing team, **Jittery Joe's Pro Cycling**, and save it to a local file in the **tests** directory as **feed.xml**:

[http://api.flickr.com/services/feeds/photos\\_public.gne?id=47608576@Noo&format=rss2](http://api.flickr.com/services/feeds/photos_public.gne?id=47608576@Noo&format=rss2)

Here is a simple test to illustrate how to employ the xml in the test. In line 15 of XMLExampleTest, alter the path from “pathToYourXml.xml” to “feed.xml”. First, a small change to our test case to make it more efficient... add a member to the class like so:

```
private var data:XMLNode;
```

and alter the setup method:

```
public function setUp():Void {  
    data = xmlData.cloneNode(true);  
    instance = new XMLExample();  
}
```

Delete the test() method and write the following:

```
public function testSetData():Void{  
    assertTrue("testSetData returned false",instance.setData(data));  
}
```

In XMLExample, add the following method. Yes, it is overkill to test a simple setter like this, but when I apply better error handling later it makes for a good example:

```
public function setData(xml : XMLNode) : Boolean {  
    if(xml == undefined || xml==null){  
        trace("XMLExample.setData: argument 'xml' is undefined");  
        return false;  
    }  
    _xml = xml;  
    return true;  
}
```

Run the test and it should pass. Now I will add one getter method to the test:

```
public function testGetChannelTitle():Void {  
    instance.setData(data);  
    var str:String = instance.getChannelTitle();  
    assertTrue("testChannelGetTitle expected:'The Bean Team's Photos' received:"+str, str=="The Bean Team's Photos");  
}
```

And the implementation in XMLExample:

```
public function getChannelTitle() : String {  
    var node:XMLNode =  
    XPathAPI.selectSingleNode(_xml.firstChild,"/*/channel/title");  
    if(node == undefined || node==null){  
        trace("XMLExample.getChannelTitle: node not found");  
    }  
    return node.firstChild.nodeValue;  
}
```

Run the test again and this should pass.

This tests the XML parsing class against known values in a controlled XML file. The tests are

not concerned with network activity or loading XML from a server. When writing client code (and Flash is a client), unit tests are not intended to be applied to network transactions. Strive for tests that are independent from external factors; writing tests that require an internet connection and a running service undermines containment and portability and goes against convention. When implementing a class that uses the XML load/onLoad methods, remain aware of this and write robust error handling in place of unit tests.

## Testing Error Handling

So the XML methods are passing the tests. That's certainly cause to go grab a coffee, but the work is only just beginning. Once this code goes into production there is no mechanism track down the failures. Now it is time to implement formal error handling.

Replace the testSetData method in XMLExampleTest with the following:

```
public function testSetData():Void{
    try{
        instance.setData(data);
    }catch(err:Error){
        fail("testSetData failed: "+err.message);
    }
}

public function testSetDataError():Void{
    try{
        instance.setData(null);
        fail("testSetDataError did not throw an error");
    }catch(err:Error){
    }
}
```

And alter the setData method in XMLExample:

```
public function setData(xml : XMLNode) : Void {
    if(xml == undefined || xml==null){
        throw new Error(this+ "setData argument is not XML");
    }
    _xml = xml;
}
```

This code introduces the `fail` assertion, which is handy for cases like this. These 2 tests illustrate a technique for making sure the methods are properly throwing errors while still passing the tests.

## Mock Objects & Testing Events

Mock objects are classes that temporarily stand in for true implementations (such as a network service). “Mock object” can refer to the simplest class of pre-loaded getters to a full blown class with complicated private methods. Extreme Programming purists will sniff at applying the term “mock object” to something that does not implement a certain level of complexity (see <http://www.mockobjects.com>), but ASUnit embraces a very simplified approach.

While ASUnit provides a rudimentary utility for mocks, there are more complex frameworks available for automating mock tests in ActionScript. If you are curious about this approach, as2lib provides a framework based on Java mock object tools:

<http://api.as2lib.org/0.9.3/>

[http://api.as2lib.org/0.9.3/org\\_as2lib\\_test\\_mock\\_MockControl.html](http://api.as2lib.org/0.9.3/org_as2lib_test_mock_MockControl.html)

### **Conclusions**

I hope this article helps you out. I am writing this section in October of 2007 and am now beginning to use unit testing in AS3. I will provide an updated tutorial as soon as I can! If you have questions, shoot me an email: [tim@timbeynart.com](mailto:tim@timbeynart.com).

Thanks!