Step-By-Step Unit Testing with ASUnit Part One

By Tim Beynart 3/09/2007

Introduction

This article will explain in detail how to use the ASUnit framework to write unit tests for ActionScript. It is aimed at an experienced AS developer who is curious about unit testing but has been frustrated by the lack of documentation for ASUnit. To understand the concepts and code I present here, you should have a strong grasp of OOP, asynchronous execution, and error handling using try…catch. All the code in this article is ActionScript 2.

There are myriad resources for understanding the philosophy and concepts behind Test Driven Development (TDD), so here I will focus on the nuts and bolts of unit testing and “test first” code authoring. In the conclusion, I will provide several links and some of my own insights so you can dive deeper into the TDD methodology.

This is part one of a two-part article.

A simple test example

A test case is simply a method in a test class that fires another method on a target class then examines the results. The code below is a very basic example to illustrate how you test the output of an addition method:

```
Import Example;
Class ExampleTest{
Public function testAdd():Void{
var example = new Example();
var result = example.add(2,3);
If(result!=4){
trace(“TEST FAILED. Expected:5, received:” +
result);
}
}
```

```
Class Example{
public function
add(num1:Number,num2:Number):Number{
Return num1+num2;
}
```

Notice that there is no code in the Example class that has anything to do with testing. Unit tests are distinct from the production code, and test classes exist independently of the classes being tested. The ExampleTest class creates an instance of Example then starts testing it. You use a test case to ensure that the method you are testing behaves the way you expect.

Note: Testing is concerned only with public members of a class. The idea is to make sure your class behaves the way anyone using it might expect. You do not (and cannot) write tests for private members. The assumption is that a successful test on a public member assures any relevant private members as well.

ASUnit.org: Download and Install

ASUnit is available at http://asunit.org/. ASUnit is a framework that provides a more elegant and rigorous test environment than the crude example above, and is based on the conventions set by other testing platforms like Java's JUnit.

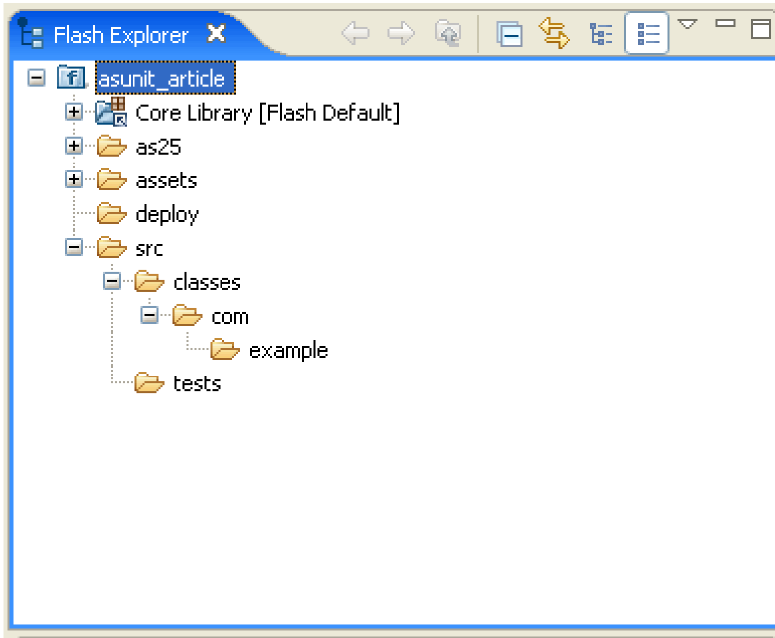
On the ASUnit web site, you will see 3 different downloads. One is just the ActionScript classes, the second contains the AsUnit.exe desktop executable, and the third is a Flash IDE extension that creates test cases for you in Flash. I recommend you download the package that includes the AsUnit.exe executable, which is what I will be using in this article. The AsUnit.exe is a desktop program that creates your test classes for you. The download also includes the ASUnit classes for AS2 and AS3 projects.
Windows: http://sourceforge.net/project/showfiles.php?group_id=108947&package_id=168969
OSX: http://aralbalkan.com/798

The installer will create an **AsUnit** directory in your Program Files that contains the executable and a **framework** directory. The framework directory contains 3 directories: as2, as25, and as3. The as2 directory is there for backwards compatibility with the original ASUnit release. The as25 directory is the latest release for ActionScript 2 testing, and is the one you will be using. It contains asynchronous test cases and several refinements to the original as2 release. And of course the as3 directory is for testing ActionScript 3.

Using the ASUnit Framework

The following is how I set up my tests, and though you may decide to do things differently. There isn't any right or wrong here, just loose convention and personal preference.

Create a folder structure like the following, using a copy of the **as25** folder:
[Note: In the screen shots I am using Eclipse (http://www.eclipse.org) with the Flash Development Tool (FDT) plugin (http://fdt.powerflasher.com) .]

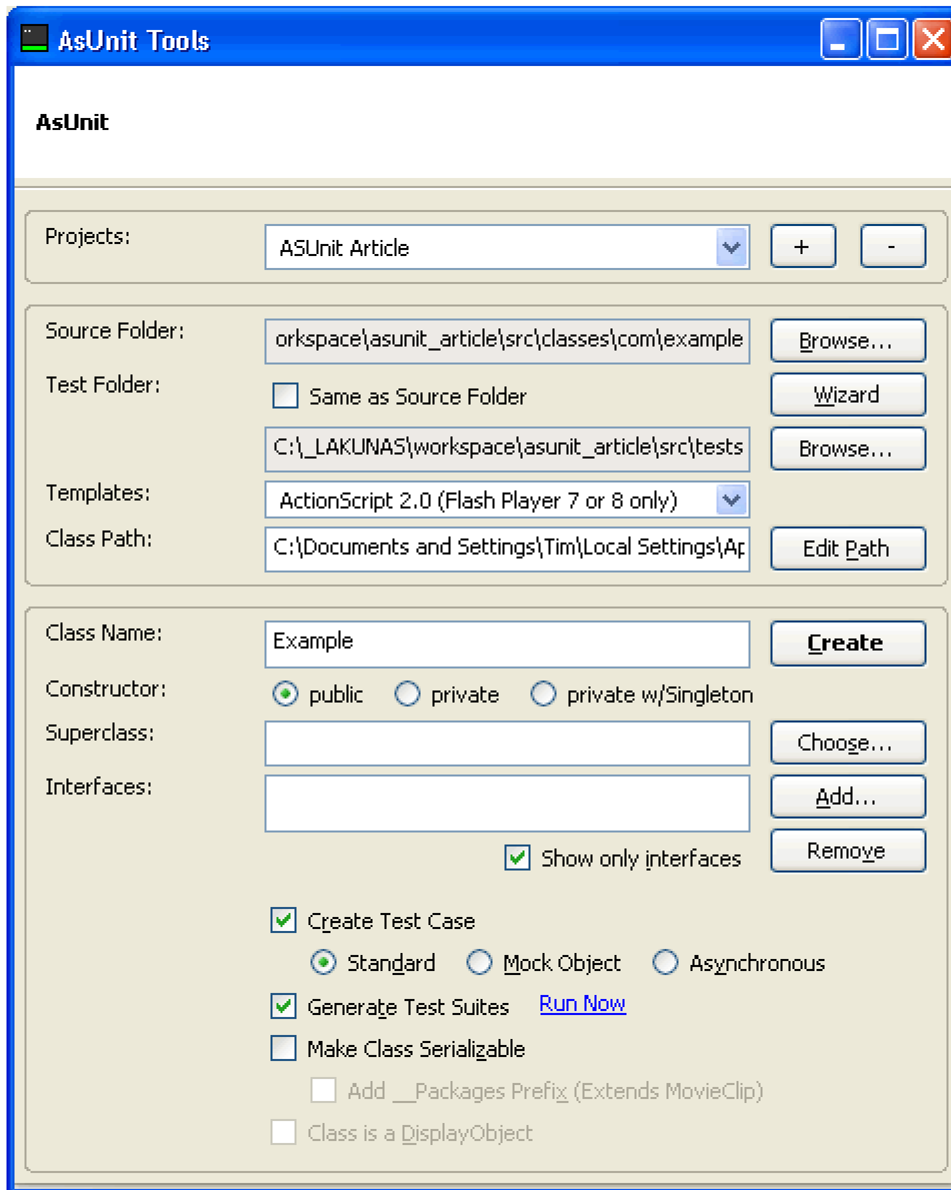


If you are using FDT, add **as25**, **classes**, and **tests** to the classpath.

Create the Test Classes

The next step is to run the AsUnit.exe program to create the test classes. The program is a bit

counterintuitive, so here's a step by step:



1. Click on the "+" next to the Projects drop down, and give your project a name.
2. In the Source Folder area, browse to the course folder for the class you want to test. Keep in mind that the class file does not exist yet! AsUnit.exe will generate the empty class for you. For the example it is C:\...\src\classes\com\example
3. Deselect "Same as Source Folder" for Test Folder.
4. In the Templates drop down, select "ActionScript 2.0 (Flash 7 or 8 Player only)", this instructs the program to use the as25 framework directory.
5. Browse to the folder you want the test classes to be generated in. For the example it is C:\...\src\tests
6. In the Class Name field, type "Example", select the "public" radio button.
7. Select "Create Test Case", select the "Standard" radio button.

8. Click “Create” next to the Class Name field. You should get a dialog that tells you “Example.as” and “ExampleTest.as” were created.
9. Select “Generate Test Suites”, click “Run Now”. You should get a dialog that says “Test Suites Created Successfully.”
10. OK, you are done for now with AsUnit.exe!

What Just Happened?

AsUnit.exe created 3 classes for us: **Example.as**, **ExampleTest.as**, and **AllTests.as**. Example is a stub file with nothing beyond a constructor. This is a good thing, since you want to write tests before you write any real code. ExampleTest contains the framework setup code to begin writing tests against the Example class. AllTests contains a reference to each test class we produce; if you were to run AsUnit.exe again to create a new test, a reference to it would appear in the updated AllTests. When you create the FLA in the next step, it will utilize the AllTests class to run the testing process.

It’s a good idea to look at the generated code in ExampleTest to get an idea of what the framework requires:

```
class ExampleTest extends TestCase {
	private var className:String = "ExampleTest";
	private var instance:Example;

	public function ExampleTest(testMethod:String) {
		super(testMethod);
	}

	public function setUp():Void {
		instance = new Example();
	}

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

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

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

Note: AsUnit.exe isn’t smart enough to add the qualified classpaths when creating the classes, so make sure you update the Example class definition and the import statement in ExampleTest.

This test extends TestCase, which provides assertion methods and the core functionality to run tests. A detailed examination of the ASUnit classes is outside of the scope of this article, but it’s important to

understand how a test suite works. Each test method that is run will use a single, discrete instance of the Example class. So, when **testInstantiated** is run by the framework, an instance of Example is created to test on in the setup method. When the next test is run, called **test**, *another* instance of Example is created. In other words, **setup** and **teardown** are fired for each test in the class. If this seems a little fuzzy, it will make more sense when you see some real tests later in the article.

Create the FLA to run our tests.

Now that the test classes are set up, you need a way to compile them. Here's how to set up the FLA to render ASUnit:

1. Move the file **as25/AsUnitTestRunner.as** to **src/tests** and rename it **ExampleTestRunner.as**. Make sure you update the class definition in the file.

2. Update the ExampleTestRunner class, making sure the main method creates an instance of the class:

```
import asunit.textui.TestRunner;

class ExampleTestRunner extends TestRunner {

	public function ExampleTestRunner() {

		start(AllTests);

	}
	public static function main():Void {
		var runner = new ExampleTestRunner();

	}
```

3. Open the Flash IDE, create a new movie called "ExampleTestRunner.fla" and save it to the **src/tests** directory.

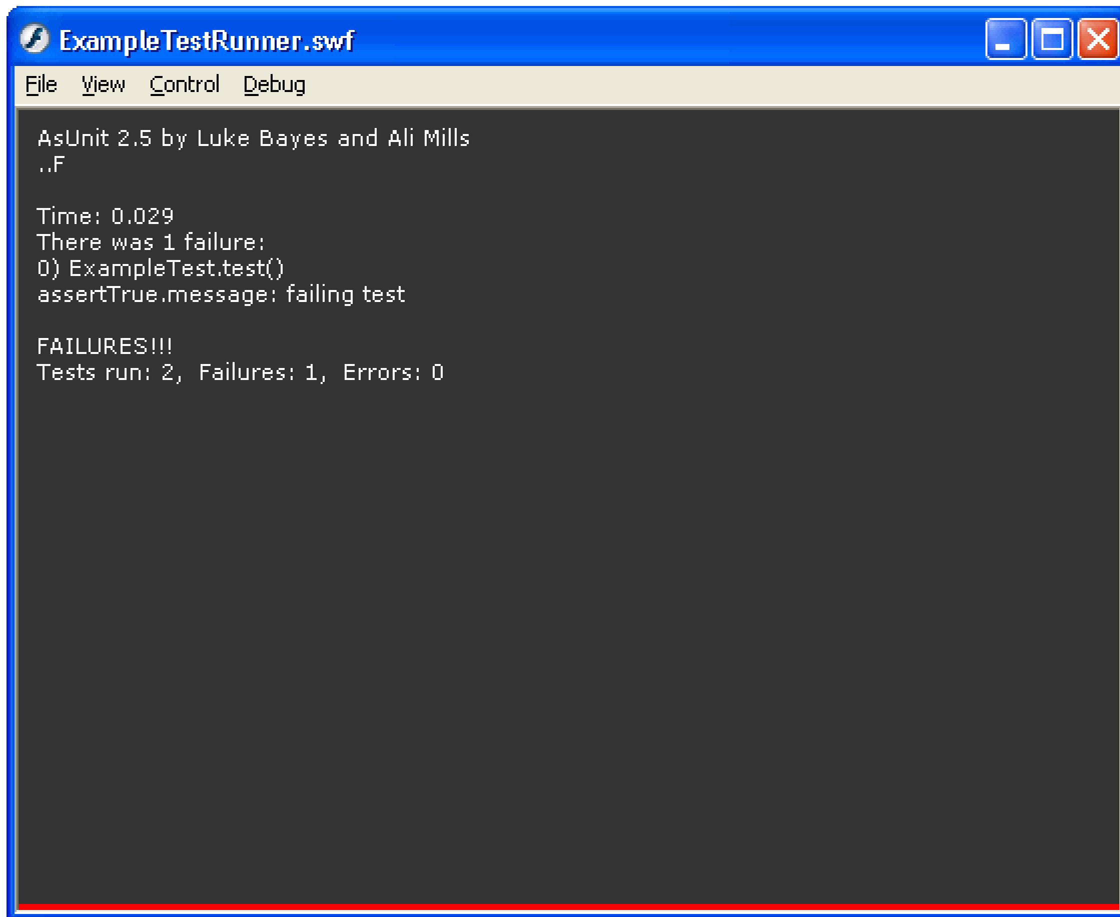
4. In the publish settings, add the classpath to your **classes** directory.

5. In the main timeline, add the following code. (Since this is a static method and this fla resides in the test directory, you do not need a constructor or import statement.):

```
ExampleTestRunner.main();
```

Run a Test!

Run a test movie. You should see the following screen :



Bummer, it failed. Let's examine why…

First of all, look at the method that failed. ASUnit provides the method name for every failed test, which is shown in the dialog as ExampleTest.test(). Open ExampleTest.as and scroll to the test() method. You will see:

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

This is a dummy test inserted by the AsUnit.exe to force a failure. Before I explain the syntax, you should understand the mechanism at work here. When the method is called by the framework, it runs an assertion. Assertions (in this case, `assertTrue`) are test methods provided by the framework and are designed for specific tests. For example, `assertTrue` will fail if the operation tested returns anything other than `true`; `assertFalse` will fail if anything other than `false` is returned. An assertion requires a string argument for the message to print if it fails, and an operation, object, or objects as arguments depending on the application. To clarify, the method syntax is `assertion("failure message", operation);`

In the example, the message is `"failing test"`, and the operation is `false`. It fails because, obviously, `false` is not `true`, so the swf prints the failure with the custom message.

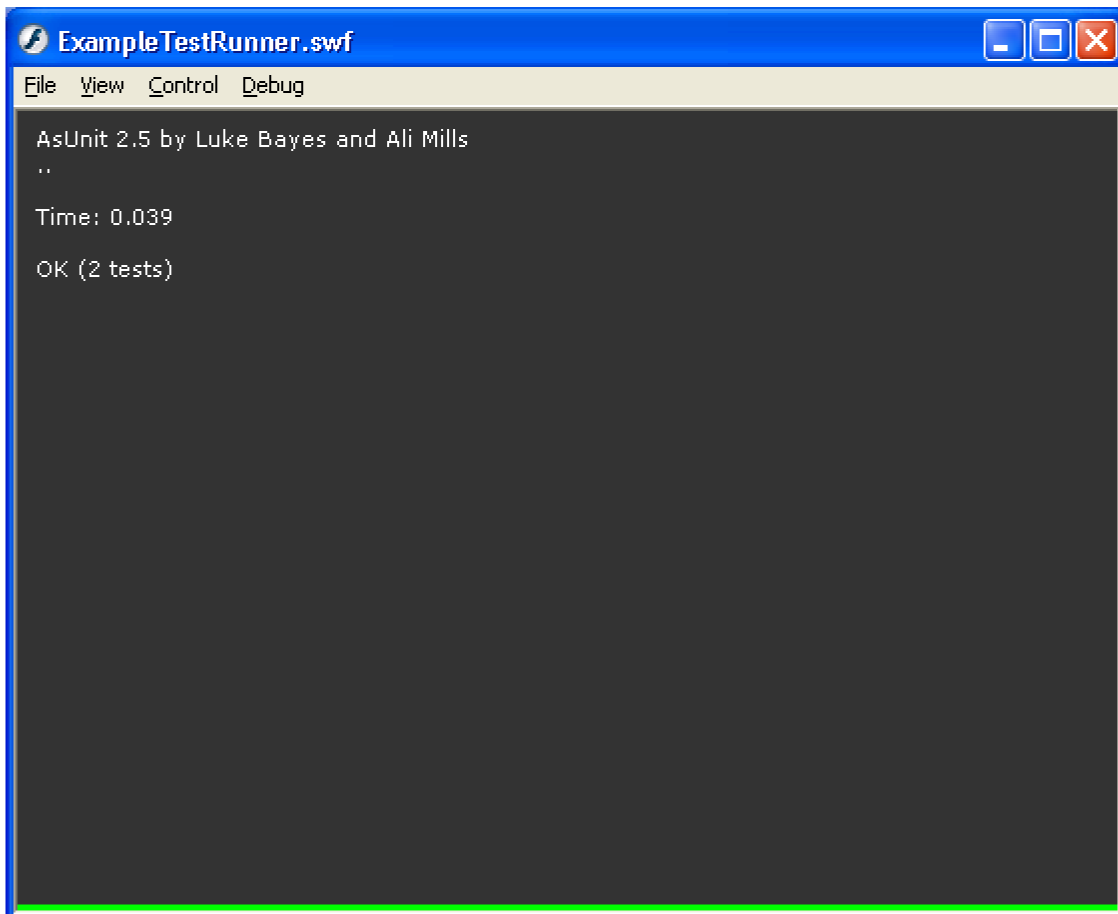
For another example, recreate the addition test from the beginning of the article. In ExampleTest, delete the bogus **test** method and add the following:

```
public function testAdd():Void {
	var result:Number = instance.add(2,3);
	assertTrue("Expected:5 Received:"+result, result==5);
}
```

Now in Example you need to write the add method in the Example class:

```
public function add(num1:Number,num2:Number):Number{
	return num1+num2;
}
```

Run the FLA again and you will see that the tests pass:



You should fiddle with the addition test to see what happens when you trigger failures. The assertions available in ASUnit are:

```
assertTrue(msg:Object, assertion:Boolean)
assertFalse(msg:Object, assertion:Boolean)
assertEquals(msg:Object, assertion1:Object, assertion2:Object)
assertSame(msg:Object, object1:Object, object2:Object)
assertNotNull(msg:Object, assertion:Object)
assertNull(msg:Object, assertion:Object)
assertUndefined(msg:Object, assertion:Object)
assertNotUndefined(msg:Object, assertion:Object)
fail( userMessage:String )
```

End of Part one

Hopefully you can get ASUnit set up using the information in this article. In the next article I will show some real world examples from a set of common classes I use in development. Once you get your head around the mechanics and syntax of unit testing, you can explore its benefits and discover where you can best apply it in your projects.

I must thank my colleague Isaac Rivera for teaching me pretty much everything you read in this article.