Unit testing

Jan Tourlamain

Introduction

1 Scope

This document describes how to write unit tests.

2 Requirements

To execute the lab you need the following:

Microsoft .NET 2.0 framework or above

Microsoft Visual Studio 2005 or above

NUnit

Rhino Mocks

A developer machine

3 Unit test

3.1 Definition

A unit test is a piece of code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterwards. If the assumptions turn out to be wrong, the unit test has failed. A “unit” is a method or function.

A unit test is not an integration test.

3.2 NUnit Framework

Testing without a framework is often ad hoc and difficult for other developers to understand. Using a testing framework makes you write tests easily and in a structured manner. NUnit is one of the existing frameworks to test your .NET code. It is free to use and is an open source product.

3.2.1 Naming conventions:

Naming standards are important because they give us comfortable rules and templates that outline what we should explain about the test. It is common to add a new class library to your solution that will contain all your test classes.

Project	[ProjectUnderTest].Test
Class	[ClassName]Tests
Method	[MethodName]_[StateUnderTest]_[ExpectedBehavior]

3.2.2 Attributes:

NUnit uses an attribute scheme to recognize and load tests. The following attributes are the most common used.

[TestFixture]	Class holds automation testing.
[TestFixtureSetUp]	Allow setting up state once before all tests.
[TestFixtureTearDown]	Allow clearing up after all tests have ran.
[Test]	Method contains automated test to invoke.
[SetUp]	Setting up the state before tests. Runs each time before a test.
[TearDown]	Clearing out the state after tests. Runs each time after a test.
[Category("Fast Test")]	To set up your tests to run under specific test categories. (ex fast tests, slow tests)

4 System Under Test: The RssController

What do you do if you like reading "De standaard" or "Le soir", but hate having to browse there every hour to see what's changed? The answer lies in RSS, which let people subscribe to the news that interests them, and have their computer automatically check for updates.

We will test a basic RssController that can be used by a GUI to view RSS feeds.

Before we can test the RssController we need to understand how RSS works. Below is an example RSS feed, and you'll see that the channel (the news feed) has a title, description and link.

You'll see that there are two <item> elements, but there could easily be hundreds, depending on how big the news site is. The <item> tags contain the actual news

items.

```
<?xml version="1.0" ?>
<rss version="2.0">
  <channel>
    <title>My Excellent Site</title>
    <description>There's lots of great content here - please subscribe!</description>
    <link>http://www.example.com</link>

    <item>
      <title>Mono rocks!</title>
      <description>Free .NET takes over world</description>
      <link>http://www.example.com/news/mono</link>
      <guid>http://www.example.com/news/mono</guid>
    </item>

    <item>
      <title>Mono beats PHP</title>
      <description>Consistent function rules</description>
      <link>http://www.example.com/news/monovsphp</link>
      <guid>http://www.example.com/news/monovsphp</guid>
    </item>
  </channel>
</rss>
```

You'll also notice that each <item> has identical <link> and <guid> elements. 'GUID' is short for globally unique identifier, and is any value that is unique to that exact story across the whole internet. This is required for RSS feeds, as it's used to let RSS programs know if they've seen that news story before or not. Each time an item is read, it is added to the GuidCache. The GuidCache is actually a text file. Once an rss item is added to this file it is not longer displayed in the item list.

A user can also subscribe to multiple seeds. Those seeds are also stored in a text file (SiteList).

5 Install NUnit

Go to the Framework folder of this lab and install the NUnit framework.

Do a default installation (you know how it works: next, next, next … finish).

6 Setup Project structure

Copy the RSS solution to your hard drive.

Open the solution.

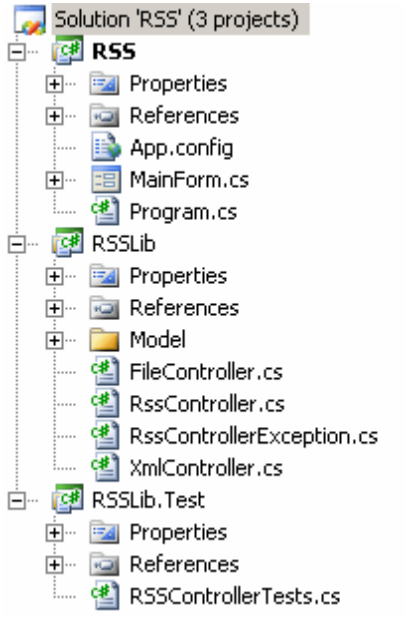
We will test the class RSSController from the RSSLib project. Make sure the RSSLib project is build on your pc.

Add a new C# Class Library. Give the library a name according the naming conventions.

Add a reference to the RSSLib project.

Add a reference to the .NET library nunit.framework

Add a class that will contain the tests for the RssController. Give the class a name according the naming conventions.



7 Writing your first unit tests

7.1 Make your class test-ready

Go to the RssControllerTests.cs.

Add the using statements for “NUnit.Framework” and “RSSLib”:

```
using NUnit.Framework;
using RSSLib;
```

You should tell NUnit that this class contains testing methods by adding the [TestFixture] attribute above the class declaration:

```
[TestFixture]
   public class RssControllerTests
   {
```

7.2 Testing the constructor

7.2.1 Test description:

The RssController has two string parameters: a pathSiteList and a pathGuidCache. The constructor should initialize those two strings. We will test if that's correctly done.

7.2.2 Implementation:

First we'll make a new testing method.

Remark: A testing method never has input parameters and doesn't return anything.

Add the method RssController_PathSiteList_Empty to the RssControllerTests class.

```
public void RssController_PathSiteList_Empty()
```

Add the attribute [Test] above the method.

```
[Test]
public void RssController_PathSiteList_Empty()  { }
```

We need to check if pathSiteList is an empty string after the RssController is initialized. Here for we can make use of the Assert class. It's a class with static methods located in the NUnit.Framework, and its purpose is to declare that a specific assumption is supposed to exist. If the arguments that are passed into the Assert class turn out to be different than what we're asserting, NUnit will realize the test has failed and will alert us. We can optionally tell the Assert class what message to alert us with if assertion fails.

Add the following statement to your testmethod:

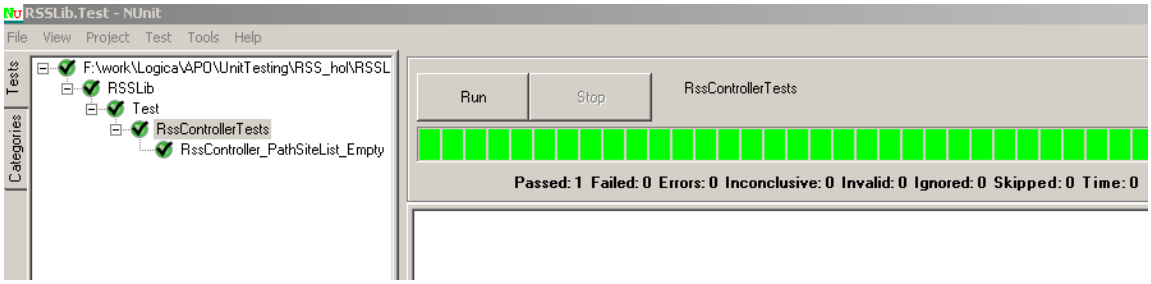
```
RssController rssController = new RssController();
Assert.AreEqual("", rssController.PathSiteList, "PathSiteList should be initialized by the
constructor");
```

7.2.3 Running the test:

Build the RSSLib.Test project and make sure a dll is created.

Open the NUnit GUI tool.

Go to File -> Open project… and select the RSSLib.Test.dll from the RSSLib.Test project. Click on "RssControllerTests" and hit for Run. NUnit will now execute all tests (in our case just RssController_PathSiteList_Empty).



7.2.4 Introducing the setup

We also want to test if the PathGuidCache is initialized by the constructor. Make a new test method “RssController_PathGuidCache_Empty” and make the assertion.

```
[Test]
public void RssController_PathGuidCache_Empty()
{
  RssController rssController = new RssController();
  Assert.AreEqual("", rssController.PathGuidCache, "PathGuidCache should be initialized by the
constructor");
```

If you look at both test methods you’ll see that the initialization of the RssController appears in both methods. We don’t want to do this for all the following testing methods in this lab.

Make a new method “Setup” without input or output parameters. Add the attribute [SetUp] above the method. NUnit will execute this method every time before each test method is executed. The name of the method itself doesn’t really matter, but it makes it easy for other developers to find your setup method.

Add a private variable RssController, name it _rssController and put the initialization in the “Setup” method. The initialization of the RssController in the test methods can be removed.

Your class should look like:

```
[TestFixture]
   public class RssControllerTests
   {
     private RssController _rssController;

       [SetUp]
       public void Setup()
       {
      _rssController = new RssController();
     }

       [Test]
       public void RssController_PathSiteList_Empty()
       {
      Assert.AreEqual("", _rssController.PathSiteList, "PathSiteList should be initialized by the
constructor");
```

```
        [Test]
        public void RssController_PathGuidCache_Empty()
        {
                Assert.AreEqual("", _rssController.PathGuidCache, "PathGuidCache should be
initialized by constructor");
    }
```

7.2.5 Check if the tests are working

Return to the NUnit GUI. Each time you build your test project, NUnit will load the new assembly.

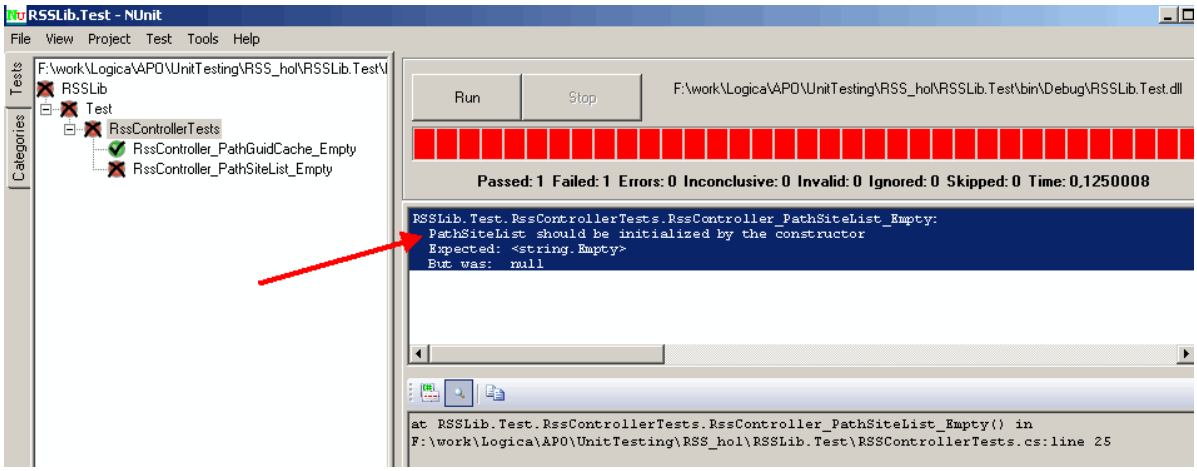
Run the tests. Both test should succeed.

Go to the RssController class in the RSSLib project and comment line 36

```
    //this.PathSiteList = pathSiteList;
```

Build the project and run the tests again.

The RssController_PathSiteList_Empty should fail because the string PathSiteList was never initialized. The error message in the NUnit GUI corresponds with the message from our Assert method.



Make sure all test are working. (uncomment line 36 in the RssController class)

8 Core testing

Having covered the basics in unit testing, we will now introduce core testing and refactoring techniques that are necessary for writing test in the real world.

8.1 Introducing stubs

Open the RssController class in the RSSLib project.

Before we test the LoadRssFeeds() method we need to know what it does. First the method declares some variables. Next a list with FeedAllocations are loaded from the PathSiteList:

```
lstFeedLocations = _fileController.LoadRowsFromFile(this.PathSiteList);
```

Here the trouble begins. Our method under test relies on an external dependency. In our tests we don't want external dependencies because the test can't control what that dependency returns to our code under test or how it behaves. If, in our case, the file is located on a server, our test will fail if the server is not connected (although our code is working correct). We can use stubs to fix this.

8.1.1 Refactoring the design:

We will refactor the design in 4 steps.

8.1.1.1.1Extract an interface to allow replacing underlying implementation

Add a new folder Interfaces to the RSSLib project

Add a new class IFileController.cs to the folder.

Add another class IXmlController (because our method under test is also using the XmlController who loads the rss feed from the internet)

Fix your dependencies by adding:

```
using RSSLib.Model;
```

8.1.1.1.2Inject stub implementation into a class under test

Go to the RssController class in the RSSLib project.

Change the private variables

From	To
private FileController _fileController; private XmlController _xmlController;	private IFileController _fileController; private IXmlController _xmlController;

Correct your dependencies:

```
using RSSLib.Interfaces;
```

Go to the FileCotroller class and inherit from the IFileController:

```
class FileController : Interfaces.IFileController
```

Go to the XmlController class and inherit from the IXmlController:

```
class XmlController : Interfaces.IXmlController
```

Next we also need to make an interface implementation in our testproject.

Add a new folder Stubs to the RSSLib.Test project.

Add a new class StubFileController in the Subs folder.

Implement the IFileController interface.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using RSSLib;
using RSSLib.Interfaces;
using System.IO;
```

```
namespace RSSLib.Interfaces
{
   public interface IFileController
   {
      List<string> LoadRowsFromFile(string strFilePath);
      void AppendTextToFile(string strFilePath, string strText);
      void DeleteFile(string strFilePath);
   }//end interface
}//end namespace
```

```
namespace RSSLib.Interfaces
{
   public interface IXmlController
   {
      List<RssItem> GetRssItemsFromFeed(string strFeedLocation);
      string GetTitleFromRssFeed(string strFeedLocation);
   }//end interface
}//end namespace
```

```
namespace RSSLib.Test.Stubs
{
   class StubFileController : IFileController
   {
      #region IFileController Members

      public List<string> LoadRowsFromFile(string strFilePath)
      {
        throw new NotImplementedException();
      }

      public void AppendTextToFile(string strFilePath, string strText)
      {
         throw new NotImplementedException();
      }

      public void DeleteFile(string strFilePath)
      {
         throw new NotImplementedException();
      }

      #endregion
   }//end class
```

Make sure the LoadRowsFromFile returns a List object:

```
public List<string> LoadRowsFromFile(string strFilePath)
{
  List<string> lstResult = new List<string>();
  lstResult.Add("http://newsrss.bbc.co.uk/rss/newsonline_world_edition/front_page/rss.xml");
  return lstResult;
}
```

This method will return a valid List<string> without using the file system.

Add a new class StubXmlController in the Subs folder.

Implement the IXmlController interface.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using RSSLib;
using RSSLib.Interfaces;

namespace RSSLib.Test.Stubs
{
   class StubXmlController : IXmlController
   {
      #region IXmlController Members
```

```
        public List<RSSLib.Model.RssItem> GetRssItemsFromFeed(string strFeedLocation)
        {
                throw new NotImplementedException();
        }

        public string GetTitleFromRssFeed(string strFeedLocation)
        {
           throw new NotImplementedException();
        }

        #endregion
    }//end class
}//end namespace
```

Make sure the GetTitleFromRssFeed returns a string:

```
public string GetTitleFromRssFeed(string strFeedLocation)
{
  return "APO Rss";
}
```

8.1.1.1.3Receive an interface as a property get or set

You can receive an interface at constructor level (=constructor injection). But, if your code under test requires more than one stub to work correctly without dependencies, adding more and more constructors(or more and more constructor parameters) becomes a hassle and it can even make the code less readable and less maintainable.

We prefer to use properties.

Open the class RssController in the RSSLib project

Add a FileController property and a XmlController property (getters and setters):

```
#if DEBUG
public IFileController FileController
{
  get { return _fileController; }
  set { _fileController = value; }
}
public IXmlController XmlController
{
  get { return _xmlController; }
  set { _xmlController = value; }
}
#endif
```

If we release our code, the properties will still be available. You can use a preprocessor directive to hide the property when released.

8.1.1.1.4Get a stub just before a method call

Go to the RssControllerTests.cs

Add a new test method “LoadRssFeeds_CorrectInput_ReturnsRssFeed()”

Create an instance of the stubs StubFileController and StubXmlController:

```
Stubs.StubFileController stubFileController = new Stubs.StubFileController();
Stubs.StubXmlController stubXmlController = new Stubs.StubXmlController();
```

Make sure your stubs for the rssController are plugged in:

```
_rssController.FileController = stubFileController;
_rssController.XmlController = stubXmlController;
```

Make the assertion:

We can make different assertions here. The method under test returns an generic list of RssFeeds. We can manually create a list in our testing method and assert that the same list is returned from our method under test. It’s much better to make sure a collection contains an expected item than to assert that both lists are equal. Otherwise if the list in your stub should grow (to make other tests) you should adapt all your other tests that are using this list.

So, we will assert that the Title from the first RssFeed is “APO Rss”:

```
Assert.AreEqual("APO Rss", _rssController.LoadRssFeeds()[0].Title, "Title of the first RssFeed
does not match.");
```

Run the test.

8.1.2 Organize your tests:

When you make a lot of tests, you should organize your tests. Normally you’ll make a distinction between “slow tests” and “fast tests”, but in our case we will distinct between “easy tests”, “stub tests” and “mock tests”. You can do that by adding the Category attribute.

Add the category “Stub tests” to our last test:

```
[Test]
    [Category("Stub tests")]
    public void LoadRssFeeds_CorrectInput_ReturnsRssFeed()
```

Add the category “Easy tests” to the other test methods.

Compile the test project

Go to the NUnit GUI

Select the tab categories

Add the stub tests

Return to the tab tests

If you select the RssControllerTest class and click on the run button, only the stub tests will run.

8.2 Exception testing

Sometimes we need to test if our program handles the exceptions as we expect.

Go to the RssController class in the RSSLib project

Go tho the LoadRssFeeds method.

If the PathSiteList does not exist, the method should throw an RssControllerException.

We will use our StubFileController to simulate a FileNotFoundException

Open the StubFileController in the RSSLib.Test project.

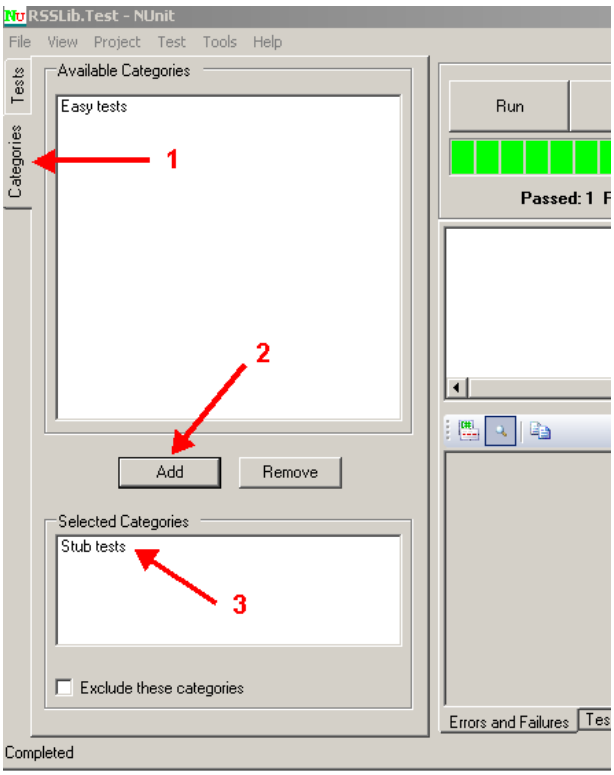
Add a private bool that will check if we want to simulate a FileNotFoundException:

```
private bool _blSimulateFileNotFoundException = false;
```

Add a public property SimulateFileNotFoundException that allow you to set the _blSimulateFileNotFoundException:

```
public bool SimulateFileNotFoundException
{
  get { return _blSimulateFileNotFoundException; }
  set { _blSimulateFileNotFoundException = value; }
}
```

Change the method LoadRowsFromFile so that we can simulate the exception if the _blSimulateFileNotFoundException is set to true:




```
List<string> lstResult = new List<string>();          lstResult.Add("http://newsrss.bbc.co.uk/
rss/newsonline_world_edition/front_page/rss.xml");
if (_blSimulateFileNotFoundException)
  throw new FileNotFoundException("Testfile not found");
return lstResult;
```

Write the test LoadRssFeeds_WrongInputFile_ThrowsException(). Make sure you set the Boolean for the exception before calling the LoadRssFeeds method:

```
[Test]
[Category("Stub tests")]
public void LoadRssFeeds_WrongInputFile_ThrowsException()
{
  Stubs.StubFileController stubFileController = new Stubs.StubFileController();
  stubFileController.SimulateFileNotFoundException = true;
  List<RSSLib.Model.RssFeed> lstResult = _rssController.LoadRssFeeds();
}
```

Run the test. The test should fail because we receive a RssControllerException. We can tell NUnit that we should receive this exception by adding the Exception attribute to our test:

```
[Test]
 [Category("Stub tests")]
[ExpectedException(typeof(RssControllerException))]
 public void LoadRssFeeds_WrongInputFile_ThrowsException()
```

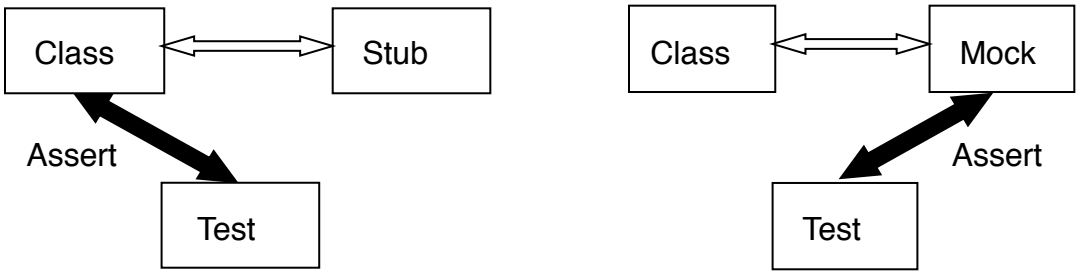
Run the test again. The test will pass because NUnit is expecting the error.

8.3 Introducing mocks

In all our previous tests we always tested the result of the method (=Result driven testing). But what if we need to test a void method with complex logic that needs to result in correct calls to other objects? Instead of result driven testing we need action driven testing. That's where mock objects come in handy.

8.3.1 Difference between stubs and mocks:

At first glance, the difference between stubs and mocks is subtle. The basic difference is that stubs can't fail tests, and mocks can. With stubs the assertion is performed on the class under test. The stub can never fail the test (it's plugged into the class under test).With mocks, the class under test communicates with the mock object. The test will use a mock object to verify whether the test failed or not.



8.3.2 Writing your first mock test:

We will write the stub test LoadRssFeeds_CorrectInput_ReturnsRssFeed, but now with a mock object because we want to know for sure that the LoadRowsFromFile (method from the IFileController) and the GetTitleFromRssFeed (method from the IXmlController) are invoked.

Add the using statement for Rhino.mocks

Add a new test method LoadRssFeeds_CorrectInputWithMock_ReturnRssFeed.

Add a category "Mock tests"

Initialize a MockRepository:

```
[Test]
[Category("Mock tests")]
public void LoadRssFeeds_CorrectInputWithMock_ReturnsRssFeed()
{
  MockRepository mocks = new MockRepository();
```

Create a mock object iXmlController:

```
IXmlController iXmlController = mocks.DynamicMock<IXmlController>();
```

Create a stub object for the IFileController:

```
IFileController iFileController = mocks.Stub<IFileController>();
```

Create a fake list to return:

```
List<string> lstFeed = new List<string>();
lstFeed.Add("FakeRssLocation");
```

We will set the expectations in the following using statement:

```
using (mocks.Record())
{
…
}
```

When we call the LoadRssFeeds method, two external methods should be triggered:

The LoadRowsFromFile method from the IFileController. This method will return a list with strings containing the feeds. We will tell our mock that, when the LoadRowsFromFile method is called a fake list must be returned:

```
Expect.Call(iFileController.LoadRowsFromFile(_rssController.PathSiteList));
LastCall.Return(lstFeed);
```

You can write those two statements into one big statement:

```
Expect.Call(iFileController.LoadRowsFromFile(_rssController.PathSiteList)).Return(lstFeed);
```

The GetTitleFromRssFeed from the IXmlController. This method should return a title:

```
Expect.Call(iXmlController.GetTitleFromRssFeed(lstFeed[0])).Return("APO RSS");
```

Make sure your mock and stub for the rssController are plugged in:

```
_rssController.FileController = iFileController;
_rssController.XmlController = iXmlController;
```

Invoke the LoadRssFeeds method:

```
List<RSSLib.Model.RssFeed> lstResult = _rssController.LoadRssFeeds();
```

Verify if all asserted expectations have been met:

```
mocks.VerifyAll();
```

Your test class should look like:

```
[Test]
[Category("Mock tests")]
public void LoadRssFeeds_CorrectInputWithMock_ReturnsRssFeed()
{
  MockRepository mocks = new MockRepository();
  IXmlController iXmlController = mocks.DynamicMock<IXmlController>();
  IFileController iFileController = mocks.Stub<IFileController>();
  List<string> lstFeed = new List<string>();
  lstFeed.Add("FakeRssLocation");
  using (mocks.Record())
  {
    Expect.Call(iFileController.LoadRowsFromFile
    (_rssController.PathSiteList)).Return(lstFeed);
    Expect.Call(iXmlController.GetTitleFromRssFeed(lstFeed[0])).Return("APO RSS");
  }//end mocks.Record
  _rssController.FileController = iFileController;
  _rssController.XmlController = iXmlController;
  List<RSSLib.Model.RssFeed> lstResult = _rssController.LoadRssFeeds();
  mocks.VerifyAll();
}
```

Run the test

Mark that we didn't write any extra stub-classes as we did before.

8.3.3 Difference between dynamic mock and StrictMock

Add an extra feed to the list lstFeed in the method LoadRssFeeds_CorrectInputWithMock_ReturnsRssFeed:

```
lstFeed.Add("FakeRssLocation2");
```

By doing so our GetTitleFromRssFeed method will be triggered twice (once for each feed).

Compile and run the test again.

The test will work because we used a Dynamic mock. A dynamic mock can only fail if an expected method was not called.

A strict mock can fail in two ways: when an unexpected method is called or when expected methods aren't called.

Change

```
IXmlController iXmlController = mocks.DynamicMock<IXmlController>();
```

To

```
IXmlController iXmlController = mocks.StrictMock<IXmlController>();
```

Run the test again

The test will fail because for each feed our method under test will call the GetTitleFromRssFeed method. We told the mock that we expect that method only once.

You can add an expect statement to make the test work:

```
Expect.Call(iXmlController.GetTitleFromRssFeed(lstFeed[1])).Return("APO RSS");
```

Run the test again

In this case it's better to use a DynamicMock because it's not bad that the GetTitleFromRssFeed method is called multiple times.if we add more feeds to our list, our test will still work without losing any functionality.

8.3.4 Another mock example

The GetRssItemsFromFeed in our class under test loads the GuidCache file only if it's not loaded yet or if it's empty.

We can check this behavior via a mock test.

Add a new test method GetRssItemsFromFeed_GuidCacheIsOnlyLoadedFirstTime_True() and initialize the necessary variables:

```
[Test]
[Category("Mock tests")]
public void GetRssItemsFromFeed_GuidCacheIsOnlyLoadedFirstTime_True()
{
  MockRepository mocks = new MockRepository();
  IFileController iFileController = mocks.StrictMock<IFileController>();
  IXmlController iXmlController = mocks.Stub<IXmlController>();
  List<string> lstGuidItems = new List<string>();
  lstGuidItems.Add("GuidItem1");
  lstGuidItems.Add("GuidItem2");
  List<RSSLib.Model.RssItem> lstRssItems = new List<RSSLib.Model.RssItem>();
  lstRssItems.Add(new RSSLib.Model.RssItem { Title = "Item1", Guid = "GuidItem1", Link =
"LinkItem1" });
  lstRssItems.Add(new RSSLib.Model.RssItem { Title = "Item2", Guid = "GuidItem2", Link =
```

Set the expectations:

```
using (mocks.Record())
{
  Expect.Call(iFileController.LoadRowsFromFile
  (_rssController.PathGuidCache)).Return(lstGuidItems);
  Expect.Call(iXmlController.GetRssItemsFromFeed
  ("FakeRssLocation")).Return(lstRssItems);
```

We will call the GetRssItemsFromFeed method twice. The first time two methods will be called (LoadRowsFromFile and GetRssItemsFromFeed). The second time, the cache may not be loaded again (because it is already loaded). Only the GetRssItemsFromFeed may be called again.

Add the following expectation and close using statement:

```
  Expect.Call(iXmlController.GetRssItemsFromFeed
  ("FakeRssLocation")).Return(lstRssItems);
}
```

Make sure your mock and stub for the rssController are plugged in:

```
_rssController.FileController = iFileController;
_rssController.XmlController = iXmlController;
```

Invoke the GetRssItemsFromFeed twice: and verify if all asserted expectations have been met:

```
_rssController.GetRssItemsFromFeed(new RSSLib.Model.RssFeed { Title = "APO Test", Location
= "FakeRssLocation" });
_rssController.GetRssItemsFromFeed(new RSSLib.Model.RssFeed { Title = "APO Test", Location
= "FakeRssLocation" });
```

Verify if all asserted expectations have been met:

```
mocks.VerifyAll();
```

Run the test.

8.4 Next steps

Now you've learned the basics and some core testing skills start writing your own unit tests. And maybe in a next step test driven development is waiting for you.

8.5 Sources:

The art of unit testing with Examples in .NET (Roy Osherhove)

http://www.nunit.org/

http://www.ayende.com/projects/rhino-mocks.aspx

http://www.tuxradar.com/content/hudzilla-coding-academy-project-five

http://aspalliance.com/
1456_Beginning_to_Mock_with_Rhino_Mocks_and_MbUnit__Part_2.1

http://python.net/crew/tbryan/UnitTestTalk/slide3.html