

21apps | and the Point is?

# Unit Test SharePoint Solutions – Getting into the Object Model

Beginners Guide to Test Driven Web Part  
Development

Andrew Woodward



21apps

## Table of Contents

Unit Testing SharePoint Solutions – Getting into the Object Model.....	3
Initial Information.....	3
Platform .....	3
Mocking SharePoint.....	3
Call it a Fake.....	3
Why fake it?.....	3
Typemock Isolator .....	4
Random Message Web Part.....	4
User Story.....	4
Step 1. Test Project.....	5
Step 2. Random Message Project.....	9
Step 3. Making the tests fail.....	10
Step 4. Fix the failing test, add more tests, fix them and repeat .....	12
Step 5. Testing SharePoint Object Model .....	14
Step 6. Completing the Tests.....	18
Step 7. Creating the web part .....	21
Conclusion.....	28
Revision History.....	29
Copyright.....	29

## Unit Testing SharePoint Solutions – Getting into the Object Model

The first document in the Beginners Guide to Test Driven Web Part Development provided an introduction to unit testing, a look at how TDD influences your design and makes your code testable. That document deliberately avoided testing the SharePoint Object Model as in order to do this we need to introduce the concept of mocking. This second document in the series will cover the concept of mocking, look at one tool that makes this possible and demonstrate some of the very recent additions that make unit testing the SharePoint object model really easy.

### Initial Information

#### Platform

The following technologies and versions are used throughout this white paper

- Windows SharePoint Services v3
- Microsoft Office SharePoint Server 2007
- Windows Server 2008
- Visual Studio 2008
- Typemock Isolator v5.1.1 (or later)

Although the examples are based on the current release of SharePoint and Windows Server most of the techniques will be valid for Windows SharePoint Services V2, SharePoint Portal Server 2003, Windows Server 2003 and Visual Studio 2005.

All development examples will be based on C#.

#### Mocking SharePoint

We have introduced a new term in this document **Mocking**. If you do a search on mocking you will find lots of articles on what mocking is, its subtle differences from stubs and often complex examples. As this is a beginner's guide we need to try and simplify the terminology:-

#### Call it a Fake

*Fake - One that is not authentic or genuine*

#### Why fake it?

Unit tests should run quickly, they should run on every developer's machine with minimal/no configuration and should only be interested in the specific item of code being tested. An essential part of unit testing is encouraging developers to test often; If you have slow tests or continual configuration changes developers will very quickly drop unit testing as it will just take up too much time.

When developing SharePoint code we often make calls into the SPSite object; if we were to do this for real our unit tests would be slow and have a dependency on each environment having the same configuration of web application and sites.

We are also not looking to test the SPSite object, we have to make the assumption that this has been tested by Microsoft. We are however looking to test our code which has a dependency on the SPSite object so we need to replace the real SPSite object with a fake one so our code can still run.

In this document we will use the excellent [Typemock Isolator](#) to fake our SharePoint Object Model calls.

## Typemock Isolator

Typemock Isolator has become the tool of choice for unit testing SharePoint as it supports faking everything including the sealed classes found throughout the object model. Isolator is a commercial product although recently it has been made available for free for use on open-source projects.

This document will be based on Version 5.1.1 of Isolator to make use of the Arrange-Act-Assert APIs and additional features that make faking SharePoint significantly easier than ever before.

Typemock have [embraced SharePoint](#) which is great news for us SharePoint developers and will help us to do TDD on our Web Part.

## Random Message Web Part

Test Driven Development (TDD) follows the mantra;

1. Write a test that fails
2. Make it work
3. Repeat

*I've excluded the refactoring and the removal of duplication for brevity.*

We are going to adopt this approach; however before we can get to this we need to understand what it is we are developing.

## User Story

In our example we want to test the SharePoint object model, specifically we want to cover the SPSite, SPWeb, SPList and SPListItem objects as this covers some of the most common interactions with SharePoint. The user story for this would read something like:

As a **power user** I want to **display a random message from a list of user defined messages** so that **it can be added to the portal and provide a random message to the portal users.**

As a developer we can estimate the effort and discuss the best ways to do this, we find out from these discussions that the user actually wants to be able to store basic text based messages in a

### Isolator Arrange – Act – Assert API

NUnit follows the AAA approach to testing, arrange your test, act on your object under test and then assert to validate the test.

Isolator has introduced an AAA approach from version 5 which makes testing much easier to understand. For more information on the new API see the Typemock

[Isolator AAA API – The Basics](#)

[Isolator AAA API – Creating Fakes](#)

SharePoint list, add this random message to a number of Web Part Pages and also be able to configure the lists used as the source of the messages. The key elements of our development are:

1. Develop web part
2. Provide custom property value that has shared storage for the list
3. Select a random message from the list

Our development does not require the creation of a custom lists template as we can use the out of the box custom list and the title column for the message text.

The following steps will walk through in detail how we approach the development and unit testing of our web part. As a developer we have already worked out the basic design of the solution we will have a simple class library that will perform the logic and a web part that will call this to render the text.

### Step 1. Test Project

We're doing TDD so we need to create our Test Project first. Yes we do this before we start doing any web part code as TDD will drive the development.

1. Load Visual Studio and create a new C# Class Library project and solution as in Figure 1. Create Unit Test Project and Solution

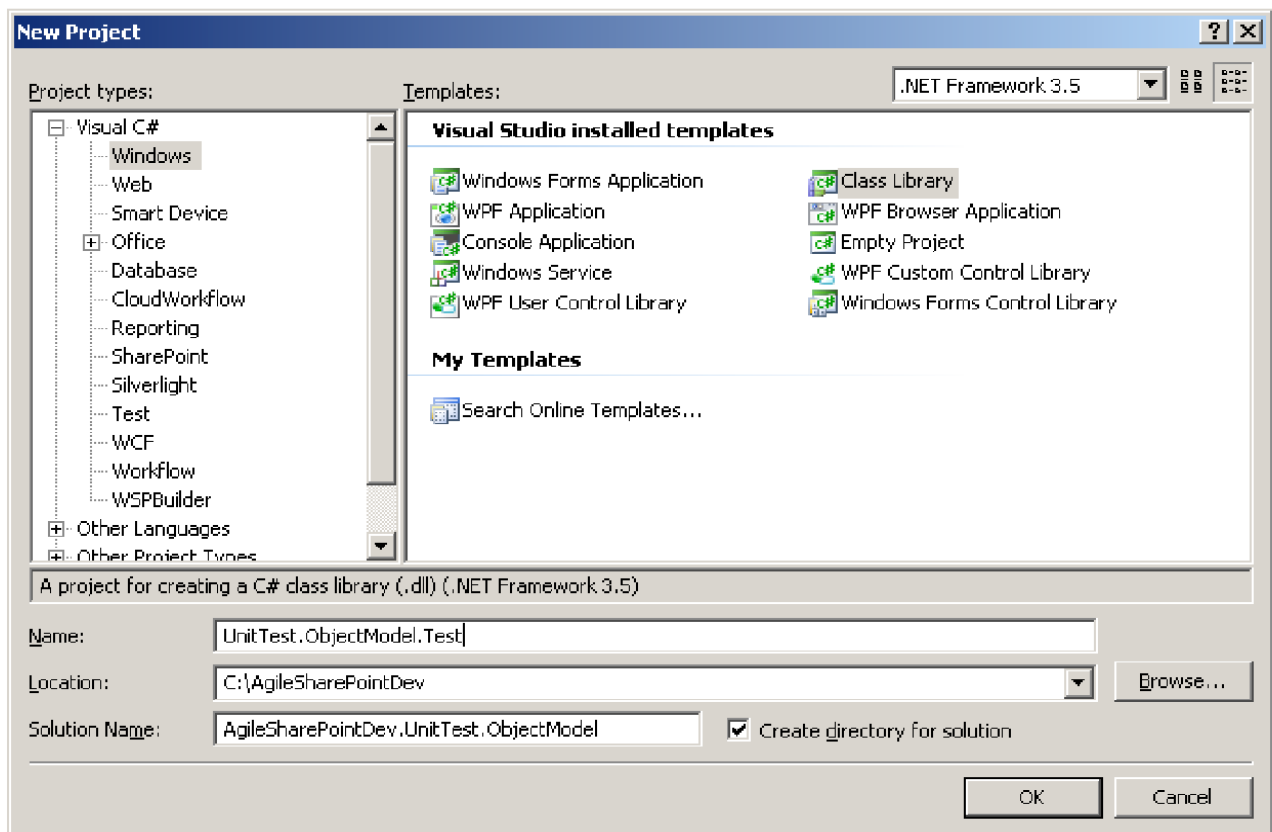


Figure 1. Create Unit Test Project and Solution

2. Add a reference to the NUnit.framework by right clicking on the project and choosing Add Reference, as in Figure 2. Add reference to NUnit framework

3. If you are using TestDriven.Net also add the NUnit.framework.extensions to make the NUnit test GUI available from the right click menu.

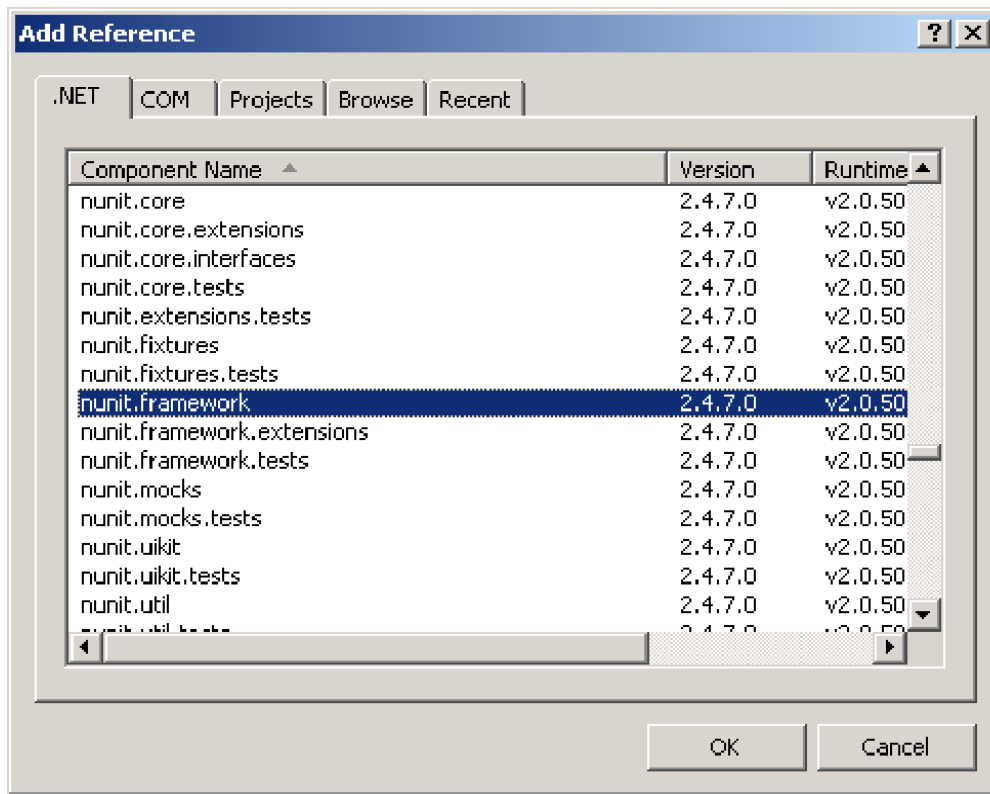


Figure 2. Add reference to NUnit framework

4. We are also going to be using Typemock Isolator's new AAA API (see Typemock Isolator) so add a reference to this as in Figure 3. Typemock Isolator – Arrange-Act-Assert.

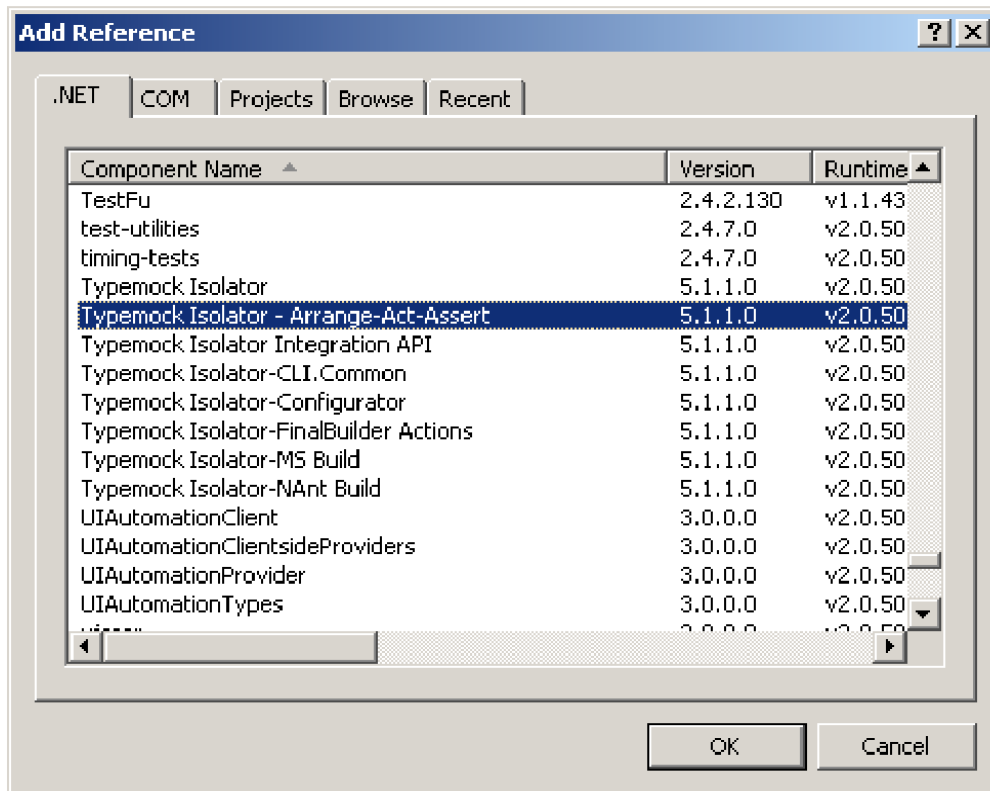


Figure 3. Typemock Isolator - Arrange-Act-Assert

5. Rename our class to match the class that we intend to test, if you rename the file this will also update the class declaration to match. You should now have a class similar to Figure 4. Initial project code.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace UnitTest.ObjectModel.Test
{
    public class RandomMessage
    {
    }
}
    
```

Figure 4. Initial project code

6. We have looked at the requirements and one of the first parts we will develop is the method to return a random number that is within a specified range. The thought here was that we would know from our SharePoint list how many messages are available and this function will select one of these at random.
7. Add code for our first test as in Figure 5. Our first Test.
  - a. Add a using statement for NUnit.Framework ①

- b. Add the [TestFixture] attribute to the class declaration ②. This tells NUnit that this class has tests to be run.
- c. Add a new Test ③; it is recognised good practice to name your tests using the Method, Scenario, Behaviour syntax as this aid both the developer to think about what they are trying to test but also to highlight what has broken if the test should go Red in the future.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using NUnit.Framework; ①

namespace UnitTest.ObjectModel.Test
{
    [TestFixture] ②
    public class RandomMessage ③
    {
        [Test] ③
        public void GetRandomNumber_PositiveRange_ReturnValueInRange()
        {
            //
        }
    }
}
    
```

Figure 5. Our first test

### Alternative Test Naming

Behaviour Driven Development (BDD) takes an alternative approach to test class and method naming, basing the test classes on behaviours and the methods are named as 'should' statements. To keep the white papers consistent we will continue to use the Method, Scenario, Behaviour format but will be posting updates based on this alternative approach.

<http://behaviour-driven.org/>

8. Now looking back at what the mantra stated we are looking to write a test that initially fails, the test we have written above will pass (go ahead try it – if you have TestDriven.Net installed just right click on the project and choose Run Tests). As in part one of the [Beginner Guide to Web Part Development- The Basics](#) we are going to separate our test code from our production code so we need to add a new project for our Random Message class.

## Step 2. Random Message Project

Test Driven Development means you write the tests first and often the tools within Visual Studio allow you to generate the methods automatically. However as we are splitting our test and production code we need to manually add the project, class and method stub.

1. Add a new C# Class project to the solution called UnitTest.ObjectModel, our namespace here is to help organise the code between the different white papers – your namespace will often reflect you organisation and project.
2. Rename the default Class1.cs to RandomMessage.cs and allow the code to be updated.
3. Add a Public method to you class called GetRandomNumber with one integer parameter, you code should look something like Figure 6. RandomMessage class code

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

namespace UnitTest.ObjectModel
{
    public class RandomMessage
    {
        public int GetRandomNumber(int max)
        {
            throw new NotImplementedException();
        }
    }
}
```

Figure 6. RandomMessage class code

4. You will note that I have added an Exception to show this method has not yet been implemented ①.

### Step 3. Making the tests fail

Now we have a class and method against which to test we write our tests. A quick check to ensure we have not done anything silly (CTRL+SHIFT+B to build the solution).

1. Add a Project reference to our UnitTest.ObjectModel project to our UnitTest.ObjectModel.Test project. Your solution should look similar to Figure 7. Add project reference.

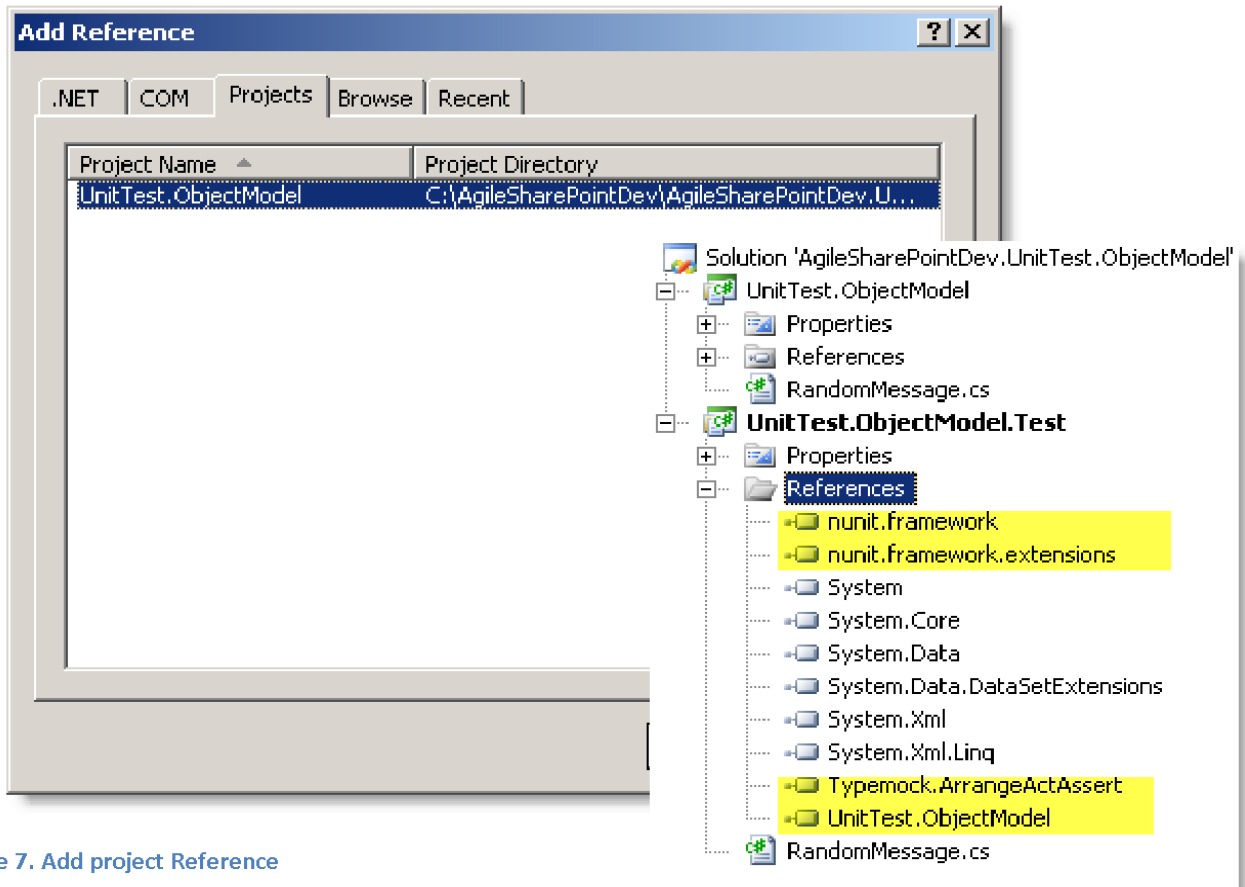


Figure 7. Add project Reference

2. We can now fill in our test method with code to call GetRandomNumber.

#### Listing 1. Initial Test – GetRandomNumber\_PostiveRange\_ReturnValueInRange

```
namespace UnitTest.ObjectModel.Test
{
    [TestFixture]
    public class RandomMessage
    {
        [Test]
        public void GetRandomNumber_PostiveRange_ReturnValueInRange()
        {
            ObjectModel.RandomMessage message = new ObjectModel.RandomMessage();

            int messageId = message.GetRandomNumber(1);

            Assert.AreEqual(0, messageId);
        }
    }
}
```

3. We have made a call to our GetRandomNumber method with a maximum value of 1 and have Asserted that the returned value should equal 0. We have zero based index in C#.
4. Run the tests and you should now get a big Red failed warning as in Figure 8. Written a test that fails.

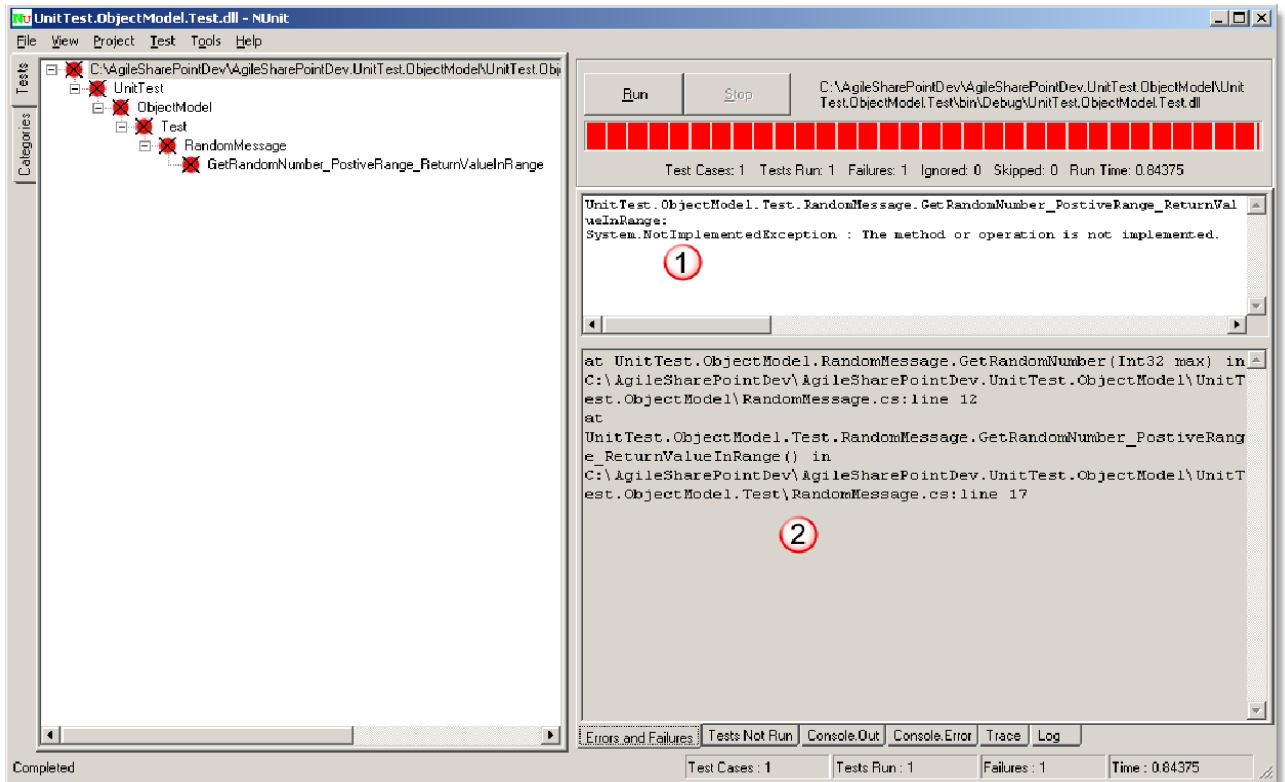


Figure 8. Written a test that fails

5. From the NUnit console we can see which test failed from the tree view but also from the error description ① which also show the System.NotImplementException. You can also get additional information about the error from the details pane ②.

## Step 4. Fix the failing test, add more tests, fix them and repeat

We have defined an initial test; we now need to make the code pass this test. Repeating our mantra for TDD we are on the Make it work step.

1. Write a test that fails
2. Make it work
3. Repeat

In this example, unlike the approach we adopted in the first white paper, we are going to take slightly bigger steps as we really want to start looking at testing the SharePoint Object Model. Working through the process we will define tests for a maximum range and tests for exceptions when the value is negative or zero as this is not a valid range.

The code in the Listing 2. GetRandomNumber method shows the class code and the code in Listing 3. GetRandomNumber Tests show the tests used to validate the method.

### Listing 2. GetRandomNumber method

```
public class RandomMessage
{
    public int GetRandomNumber(int max)
    {
        if (max < 1)
            throw new ArgumentOutOfRangeException("max",
                max,
                "Maximum value must be greater than or equal to 1");
        Random rnd = new Random();
        return rnd.Next(max);
    }
}
```

Formatting has been changed to make it fit in the document.

### Listing 3. GetRandomNumber Tests

```
[TestFixture]
public class RandomMessage
{
    private ObjectModel.RandomMessage message;

    [SetUp]
    public void Setup()
    {
        message = new ObjectModel.RandomMessage();
    }

    [Test]
    public void GetRandomNumber_PositiveRange_ReturnValueInRange()
    {
        int messageId = message.GetRandomNumber(1);
        Assert.AreEqual(0, messageId);
    }

    [Test, ExpectedException(typeof(ArgumentOutOfRangeException))]
    public void GetRandomNumber_Zero_ThrowException()
    {
        int messageId = message.GetRandomNumber(0);
    }
}
```

```
    }  
  
    [Test, ExpectedException(typeof(ArgumentOutOfRangeException))]  
    public void GetRandomNumber_Negative_ThrowException()  
    {  
        int messageId = message.GetRandomNumber(-1);  
    }  
  
    [Test]  
    public void GetRandomNumber_MaxValue_ReturnValueInRange()  
    {  
        int messageId = message.GetRandomNumber(int.MaxValue);  
  
        Assert.GreaterOrEqual(messageId, 0);  
        Assert.LessOrEqual(messageId, int.MaxValue);  
    }  
}
```

You will see from the tests that we have also added a Setup step, the mantra of refactoring code and removing duplication is equally as valid in your Tests as it is in your production code. In NUnit the [Setup] attribute identifies code that will be run before every test and is a good way to arrange things if it is common across all tests.

1. Run the tests again and you should see green as in Figure 9 Passed Initial Tests.

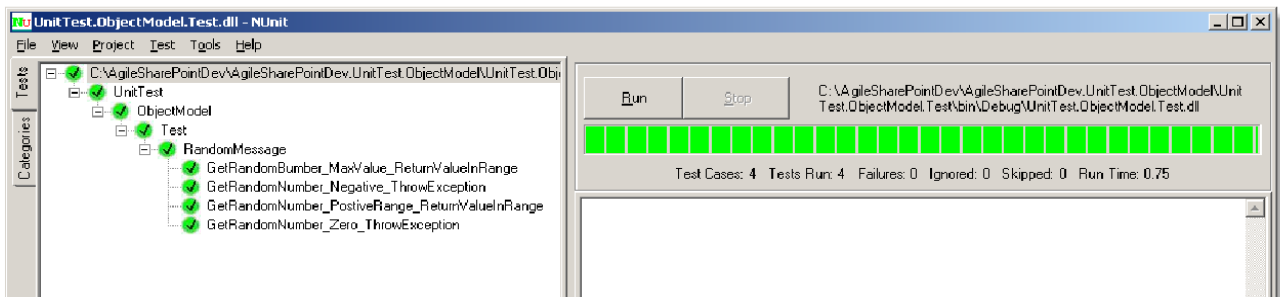


Figure 9. Passed Initial Test

OK that enough of the boring stuff now time to move onto testing the SharePoint object model.

## Step 5. Testing SharePoint Object Model

Taking a TDD approach we will add a new test, generate the method based on the test, run the failing test and then make it pass.

1. Add a new test that will check that a valid message is returned from a list with only one item. Initially the code will look something like Figure 10. New Test GetRandomMessage

```
[Test]
public void GetRandomMessage_SingleMessage_ReturnMessage()
{
    string rndMessage = message.GetRandomMessage("http://validsiteurl");
    Assert.AreEqual("SingleItem", rndMessage);
}
```

Figure 10. New Test GetRandomMessage

2. Using the features of Visual Studio or a 3<sup>rd</sup> party tool like ReSharper you can automatically generate the method GetRandomMessage <sup>1</sup> as shown in Figure 11. Generate method stub.

```
= message.GetRandomMessage("http://validsiteurl");
SingleItem"
```

Generate method stub for 'GetRandomMessage' in 'UnitTest.ObjectModel.RandomMessage'

Figure 11. Generate method stub

3. Running the tests should give you 4 Green passed and 1 Red failed tests, which is what we expect, our code has not broken the existing tests but our new test is expected to fail as we have not coded it yet.
4. Again we are only looking to write enough code each time to make our tests work, so completion of the logic will take us a few iterations, but we do this one step at a time - at this point you really are doing TDD 😊
5. Our method will have a number of interactions with the SharePoint Object model that we will need to fake.
  - a. Get the SPSite based on the URL
  - b. Open the relative web based on the URL
  - c. Get the SPList
  - d. Get the random item based on the SPList length
  - e. Get the list item title
6. Add a reference to SharePoint to both projects as in Figure 12. Add a Reference to SharePoint

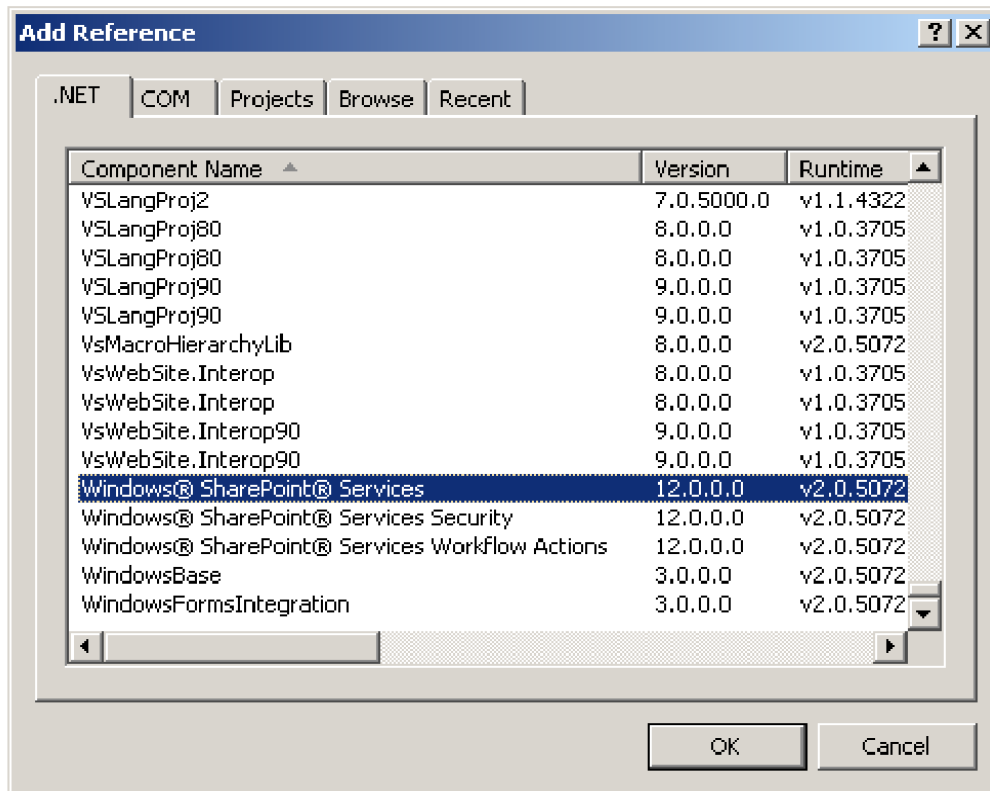


Figure 12. Add a Reference to SharePoint

7. A using statement using for the SharePoint and Typemock Isolator Arrange-Act-Assert API

```
Using TypeMock.ArrangeActAssert;
Using Microsoft.SharePoint;
```

8. Our method will create a new SPSite object which we want to fake and define some properties on in our test. We are doing this before the object is actually created so this is referred to as a **Future Instance**. We are not interested in most of the calls to the SPSite object we are faking so we use the Members.ReturnRecursiveFakes parameter (see sidebar Isolator Recursive Fakes)

```
SPSite site = Isolate.Fake.Instance<SPSite>(Members.ReturnRecursiveFakes);
```

**Isolator Recursive Fakes**

One of the great new features of Isolator is the ability to return recursive fakes. Previously your test would need to fake every call made to the SharePoint Object Model which made the tests very brittle. Recursive Fakes instruct Isolator to return a null value for everything; you only need to provide values that you need to perform the required test.

9. If our method accepted an SPSTite object as a parameter we would pass this fake object into the method. In our example our method accepts a URL and will create the SPSTite object internally so we need to instruct Isolator to use the one we created. This is done using the `Isolate.Swap` method

```
Isolate.Swap.NextInstance<SPSTite>().With(site);
```

`Isolate.Swap` tells Isolator to replace the next instance of the SPSTite object with the fake one we created.

10. In our initial test we are validating that a list with one item returns that items title, we therefore need to fake only a couple of things.

Fake the call to `ItemCount` which is used to pass into the `GetRandomNumber` method

```
Isolate.WhenCalled(() =>
site.OpenWeb().Lists["Messages"].ItemCount).WillReturn(1);
```

Fake the call to the first item in the message list.

```
Isolate.WhenCalled(() =>
site.OpenWeb().Lists["Messages"].Items[1].Title).WillReturn("Message
One");
```

11. At first you may think that we are just telling the code what to do. What we are actually doing is telling Isolator to return fake values for the items where we expect our method to interact with SharePoint. Our actual code still needs to have the correct logic to make the required calls, process the responses and return the correct message. Our new test should look like Figure 13. `GetRandomMessage` Test Faking the SharePoint OM.

```
[Test]
public void GetRandomMessage_SingleMessage_ReturnMessage()
{
    //Arrange
    SPSTite site = Isolate.Fake.Instance<SPSTite>(Members.ReturnRecursiveFakes);
    Isolate.Swap.NextInstance<SPSTite>().With(site);

    Isolate.WhenCalled(() => site.OpenWeb().Lists["Messages"].ItemCount).WillReturn(1);
    Isolate.WhenCalled(() => site.OpenWeb().Lists["Messages"].Items[0].Title).WillReturn("Message One");

    //Act
    string rndMessage = message.GetRandomMessage("http://validsiteurl");

    //Assert
    Assert.AreEqual("Message One", rndMessage);
}
```

Figure 13. `GetRandomMessage` Test Faking the SharePoint OM

12. Running the test will still give us the `NotImplementedException`, we need to write the code to make the test pass, this should look something like Figure 14. `GetRandomMessage` method and will be familiar to SharePoint developers.

```

public string GetRandomMessage(string WebUrl)
{
    string message = "";
    using (SPSite site = new SPSite(WebUrl))
    {
        using (SPWeb web = site.OpenWeb())
        {
            SPList messages = web.Lists["Messages"];
            int rndMessage = GetRandomNumber(messages.ItemCount);
            message = messages.Items[rndMessage].Title;
        }
    }
    return message;
}

```

Figure 14. GetRandomMessage method

13. What is evident is that our tests using the new Isolator syntax makes the tests far less brittle. We faked the SPSite recursively which effectively fakes our interaction with the SharePoint object model and we set a couple of properties to guide our test through the code.

#### Isolator API Recap

**Fake** a future instance of SPSite, normally used as parameters to methods or with Swap

```
SPSite site = Isolate.Fake.Instance<SPSite>(Members.ReturnRecursiveFakes)
```

**Swap** a value with the one provided, in this example the next instance

```
Isolate.Swap.NextInstance<SPSite>().With(site);
```

**WhenCalled** a the specified value with is returned

```
Isolate.WhenCalled(() =>
site.OpenWeb().Lists["Messages"].ItemCount).WillReturn(1);
```

## Step 6. Completing the Tests

We have tested the core functionality we now need to look at the exceptions that we could encounter. The GetRandomMessage method should handle the errors and return a friendly, but informative message advising of the issue.

1. Testing for a missing Messages list. When accessing the Lists collection an ArgumentException is thrown if the list cannot be found. The test for this will make use of the Isolate.WhenCalled().WillThrow method as in Listing 3.  
GetRandomMessage\_MessageListNotFound\_ReturnErrorMessage

Listing 3. GetRandomMessage\_MessageListNotFound\_ReturnErrorMessage

```
[Test]
public void GetRandomMessage_MessageListNotFound_ReturnErrorMessage()
{
    //Arrange
    SPSite site = Isolate.Fake.Instance<SPSite>(Members.ReturnRecursiveFakes);
    Isolate.Swap.NextInstance<SPSite>().With(site);

    Isolate.WhenCalled(() => site.OpenWeb().Lists["Messages"]).WillThrow(new
    ArgumentException("Value does not fall within the expected range.));

    //Act
    string rndMessage = message.GetRandomMessage("http://validsiteurl");

    //Assert
    Assert.That(rndMessage.StartsWith("Error"));
    Assert.That(rndMessage.Contains("Messages list could not be found"));
}
```

2. The next exception checks if the site or web exists, this is validated during the creation of the SPSite object and thows a FileNotFoundException.
3. Unfortunately the Isolator AAA API does currently not have a way of throwing exceptions on object initialisation so we have to revert to the use of Natural Mocks (see side bar). To do this we need to add a reference the Typemock as in Figure 15. Typemock Isolator Natural Mocks Reference.

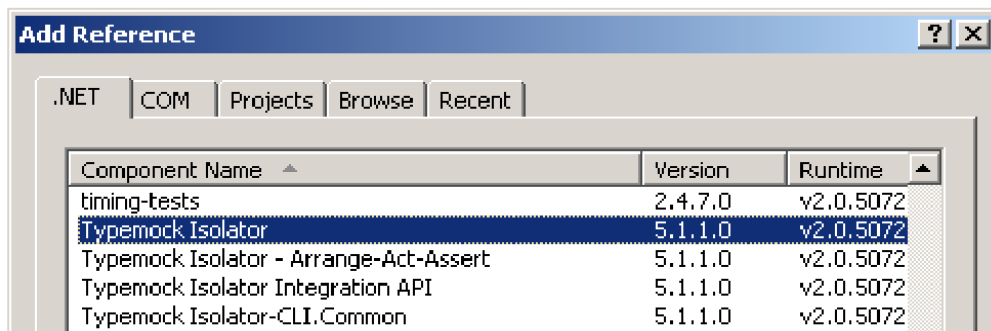


Figure 15. Typemock Isolator Natural Mocks Reference

- Using the Natural Mocks syntax we will record the things that we expect to happen, for this test it is reasonably simple. We use the ExpectAndThrow method to tell Isolator that we expect a new SPSite to be initiated and that we want Isolator to throw a FileNotFoundException (this is the exception that the SPSite throws if the site or web cannot be found). Our test code should look something like Listing 4.  
GetRandomMessage\_SiteNotFound\_ReturnErrorMessage

### Natural Mocks

Natural Mocks were introduced into TypeMock to make the syntax for the Record-Replay approach to mocking easier. It is still a valid approach and was how most SharePoint unit testing was completed before the AAA API was release.

<http://www.typemock.com/Docs/UserGuide/NaturalMocks.html>

#### Listing 4. GetRandomMessage\_SiteNotFound\_ReturnErrorMessage

```
[Test]
public void GetRandomMessage_SiteNotFound_ReturnErrorMessage()
{
    //Arrange
    using (RecordExpectations expectations = RecorderManager.StartRecording())
    {
        expectations.ExpectAndThrow(new SPSite("http://invalid_url"), new
FileNotFoundException());
    }

    //Act
    string rndMessage = message.GetRandomMessage("http://invalid_url");

    //Assert
    Assert.That(rndMessage.StartsWith("Error"));
    Assert.That(rndMessage.Contains("Site could not be found"));
}
```

- Running this test should fail this test; we need to add an additional try-catch to deal with this error. Our code should look something like Listing 5. GetRandomMessage with try-catch blocks.

#### Listing 5. GetRandomMessage with try-catch blocks

```
public string GetRandomMessage(string WebUrl)
{
    string message = "";

    try
    {
        using (SPSite site = new SPSite(WebUrl))
        {
            using (SPWeb web = site.OpenWeb())
            {
                try
                {
                    SPList messages = web.Lists["Messages"];
                }
            }
        }
    }
}
```

```

        int rndMessage = GetRandomNumber(messages.ItemCount);
        message = messages.Items[rndMessage].Title;
    }
    catch (ArgumentException)
    {
        message = "Error: Messages list could not be found";
    }
}
}
}
}
catch (FileNotFoundException)
{
    message = "Error: Site could not be found";
}
return message;
}
}

```

6. All of our tests now pass and we are happy that we have covered the code sufficiently. Depending on your coding style you may choose to refactor the try-catch blocks into one. The advantage of doing TDD is that we can make these changes, re-run the tests and if all pass we can be confident we have not broken anything. You refactored code will look something like Listing 6. Refactored GetRandomMessage.

### Listing 6. Refactored GetRandomMessage

```

public string GetRandomMessage(string WebUrl)
{
    string message = "";

    try
    {
        using (SPSite site = new SPSite(WebUrl))
        {
            using (SPWeb web = site.OpenWeb())
            {
                SPList messages = web.Lists["Messages"];
                int rndMessage = GetRandomNumber(messages.ItemCount);
                message = messages.Items[rndMessage].Title;
            }
        }
    }
    catch (ArgumentException)
    {
        message = "Error: Messages list could not be found";
    }
    catch (FileNotFoundException)
    {
        message = "Error: Site could not be found";
    }
    return message;
}
}

```

As in the previous white paper we have very quickly managed to code the unit tests and system code to perform the core requirements of our web part. We will continue with the web part class. What we have also achieved is a way to test code that works against the SharePoint object model and we can do this without the need for any specific configuration of SharePoint on the developers' environment.

## Step 7. Creating the web part

This white paper is not aimed at teaching you how to develop web parts, there is more than enough good examples of this available, if you want more detailed instructions on creating web parts you should look at the excellent labs available from Microsoft at <http://www.microsoft.com/click/SharePointDeveloper/>.

In order to complete the User Story we are now going to package up our unit tested class into a web part. As detailed in part one [Beginner Guide to Web Part Development- The Basics](#) we are not looking to test the Web Part framework, additional testing of the web part should be performed as part of integration and system testing.

We will cover the web part code and how to deploy this manually, which will lead us very nicely into the white papers on the other aspects of agile SharePoint development including Integration Testing, Automated solution building and Continuous Integration.

Let's get on and create our Web Part; we are not going to use any plug-ins for this like the Visual Studio Extensions for WSS

1. Add a new Project based on Class Library to the existing solution and call `UnitTest.ObjectModel.WebParts`, as in Figure 16. Add Web Part Project.

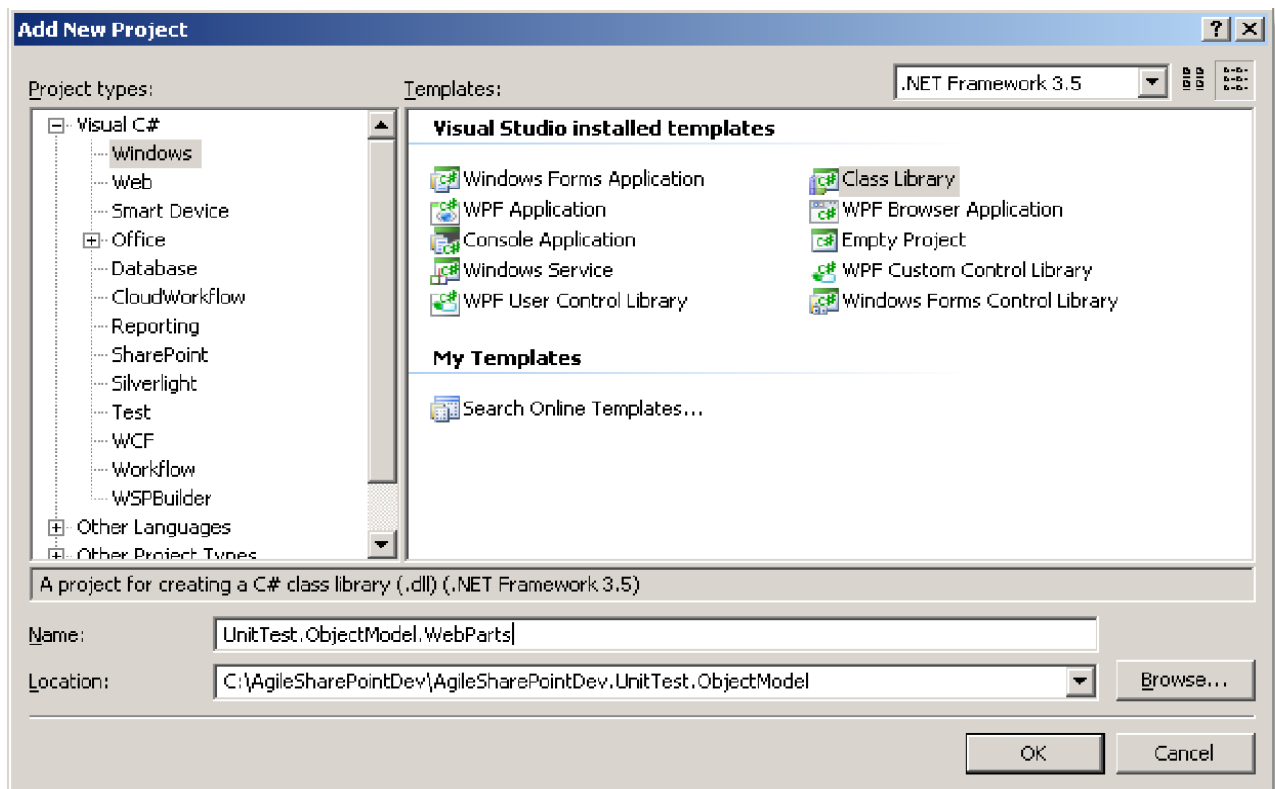


Figure 16. Add Web Part Project

2. Add a reference to `System.Web` by right clicking on the project and choosing Add Reference.
3. Rename the `Class1.cs` file to `RandomMessage.cs` and accept the change to the class name in the code.

4. Our Web Part is going to inherit from the ASP.Net Web Part class, Figure 17. Inherit from ASP.Net Web Part shows the code for this. This is the recommended best practice for SharePoint unless you have a specific requirement for the additional features provided by the SharePoint Web Part class, in our simple example we don't.

```
namespace UnitTest.ObjectModel.WebParts
{
    public class RandomMessage : System.Web.UI.WebControls.WebParts.WebPart
    {
    }
}
```

Figure 17. Inherit from ASP.Net Web Part

5. Always start with Hello World! One of the biggest problems with SharePoint development for new and experienced developers alike is the number of things that have to be in place for something to work. This is why I always start my web parts as Hello World web parts and get them on the page, I can then add in my code knowing that I have the configuration in place and any errors are down to my code. Add some simple Hello World code by overriding the RenderContents and add code to write Hello World as in Figure 18. Render Hello World

```
public class RandomMessage : System.Web.UI.WebControls.WebParts.WebPart
{
    protected override void RenderContents(System.Web.UI.HtmlTextWriter writer)
    {
        //base.RenderContents(writer);
        writer.WriteLine("Hello world.");
    }
}
```

Figure 18. Render Hello World

6. As this is beginners guide and not aimed at teaching you how to develop web parts I am going to keep this simple. We need to do the following to deploy our web part
  - a. Copy the DLL to the Web Applications bin directory (no deployment GAC here 😊)
  - b. Add a Safe Controls entry for the DLL
  - c. Add the Web Part to our web part gallery
7. I like to automate the coping of the Web Part as this allows me to do CTRL+SHIFT+B and have the DLL automatically updated. SharePoint defaults the IIS web site locations to

C:\inetpub\wwwroot\wss\VirtualDirectories\

Under this directory you will find a folder for each of your web applications, you will also see one for your central admin site and maybe your Shared Service Provider if you have created one. We need to copy our Web Part DLL into the bin directory of our web application, in my case this is

C:\inetpub\wwwroot\wss\VirtualDirectories\agile.21apps.dev\bin

8. Add a Post Build event to your Web Parts project in Visual Studio by right clicking on the project and choosing properties. This will show you a properties page and on the left you will have a Build Events Tab click this and you should see something like Figure 19. Build Events Tab. In here we are going to add a Post-build event, something we want to happen after the project has successfully built, to copy our compiled DLL into the bin discussed above.

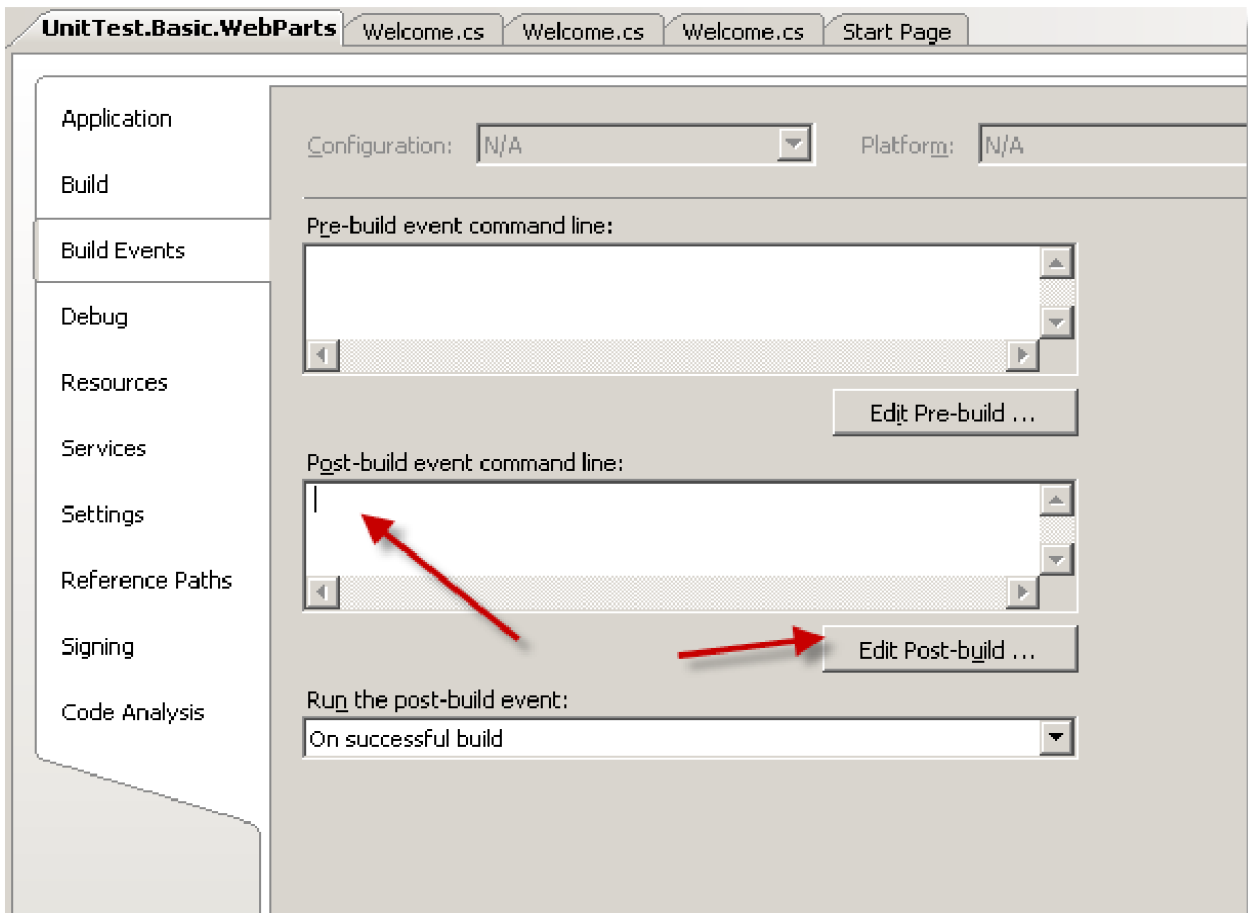


Figure 19. Build Events Tab

9. Post Build Event code will look something like this

```
cd "$(ProjectDir)"
xcopy "$(TargetDir)*.dll" "C:\inetpub\wwwroot\wss\VirtualDirectories\[Site]\bin\" /y
```

This script makes use of some of the Visual Studio functionality to evaluate the values at runtime, to see what's available and what they evaluate to in your project click the Edit Post-build button and click Macros.

In our example we are copying all of the DLLs (\*.dll) from the Target Directory (this will be debug/bin in our case) into the SharePoint Sites bin directory, we need to replace the [Site] with the directory in your environment. My post build event looks like this

```
cd "$(ProjectDir)"
```

```
xcopy "$(TargetDir)*.dll"  
"C:\inetpub\wwwroot\wss\VirtualDirectories\agile.21apps.dev\bin\" /y
```

10. Test the post build event now, CTRL+SHIFT+B to rebuild the solution. If the build fails you may have a directory name wrong. If it works you should have a copy of the `UnitTest.ObjectModel.WebParts` in your bin directory.
11. Adding a Safe Controls entry will be done manually for now. SharePoint requires that the DLL is registered as a Safe Control in web.config before it will render it as a web part. Open up the web.config file for your web application and find the SafeControls section, you will see a large number of controls already added including Microsoft's own controls. Add your safe controls entry (it should look the same as below if you used the same Namespaces as me).

```
<SafeControl Assembly="UnitTest.ObjectModel.WebParts"  
Namespace="UnitTest.ObjectModel.WebParts" TypeName="*" Safe="True"  
>
```

12. In the previous step we just told the web application that this web part is safe, we did not deploy it to anywhere. In this step we will manually add the web part to a Site collections web part gallery. I normally use a Team Site for testing web parts; from the root site choose Site Actions -> Site Settings and Web Parts (as per Figure 20. Web Parts Gallery).

Click New in the tool bar, this page sometimes takes a bit of time to load after you have changed web.config. The page shows a list of Web Parts that are available for this Gallery. The list is in alphabetic order so my web part `UnitTest.ObjectModel.WebParts` is at the bottom, with an automatically generated .webpart file. Select the web part and click populate.

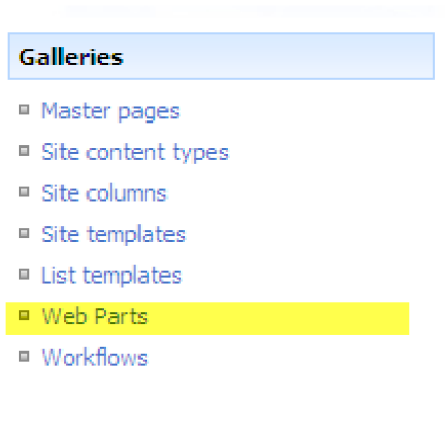


Figure 20. Web Parts Gallery

If you can't see your web part you have made a typing error in the SafeControls or the DLL is not in the in directory. This is the reason we do the Hello World build, to resolve any configuration issues before we start adding our code.

13. Navigate back to the site and add the web part to the page, it will be in the Miscellaneous section and have a Title of RandomMessage and no description (assuming you left the default values).

14. Assuming all is well you should see your Hello World web part displayed as per Figure 21. Hello World Web Part

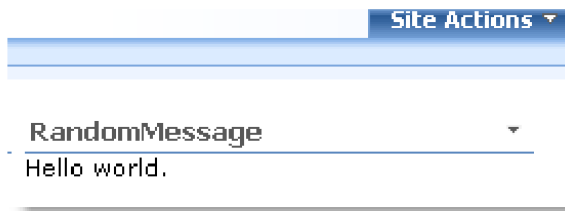


Figure 21. Hello World Web Part

15. Now let's add our Random Message code, you'll be surprised how easy this is now. Right Click on the WebParts project and add a Project reference to the UnitTest.ObjectModel project as per Figure 22. Add Project Reference

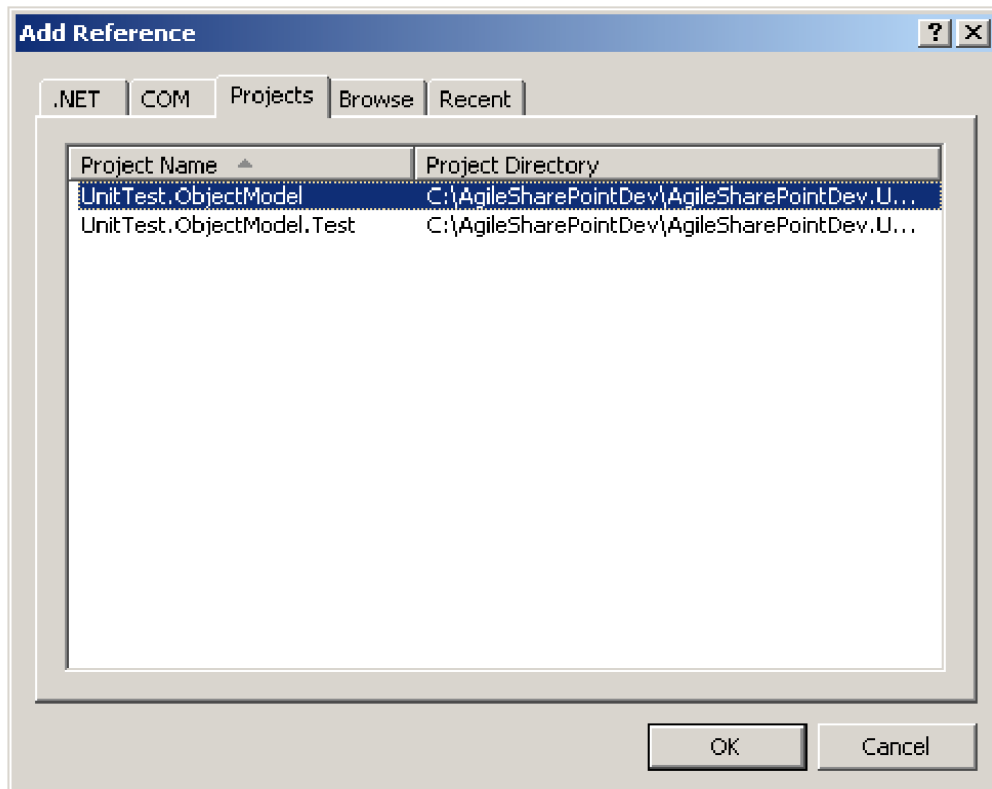


Figure 22. Add Project Reference

16. Web Parts provide the entire framework for collection information from the end user and the storage and retrieval of this information. We need to add a public property to hold the Url of the Message list.
17. We will also update the RenderContents method to call into our ObjectModel.RandomMessage method passing in the property we defined above. Your updated code should look something like Figure 23. Random Message Web Part Property

```

namespace UnitTest.ObjectModel.WebParts
{
    public class RandomMessage : System.Web.UI.WebControls.WebParts.WebPart
    {
        private string _messagesUrl = "";

        [Personalizable(PersonalizationScope.Shared),
        WebBrowsable(true),
        System.ComponentModel.Category("Agile SharePoint"),
        WebDisplayName("Random Message Site Url"),
        WebDescription("The fully qualified URL to the site with a Messages list.")]
        public string SiteUrl
        {
            get { return _messagesUrl; }
            set { _messagesUrl = value; }
        }

        protected override void RenderContents(System.Web.UI.HtmlTextWriter writer)
        {
            ObjectModel.RandomMessage randomMessage = new ObjectModel.RandomMessage();
            writer.WriteLine(randomMessage.GetRandomMessage(_messagesUrl));
        }
    }
}

```

Figure 23. Welcome Message Web Part Property

18. Rebuild the solution CTRL+SHIFT+B, and reload the page you added the web part, you may be presented with the error displayed in Figure 24. Security Exception, or a friendly (I use that term loosely) SharePoint error message. You have three options, and I leave it to you to decide.
  - a. Create a custom Code Access Security Policy for your web part granting it rights to the SharePoint object model (I would skip this now and have that created as part of your Continuous build process)
  - b. Update the trust level of the web application to medium (easiest quick fix)
  - c. Deploy the DLL to the GAC (you're not really going to do this are you?)

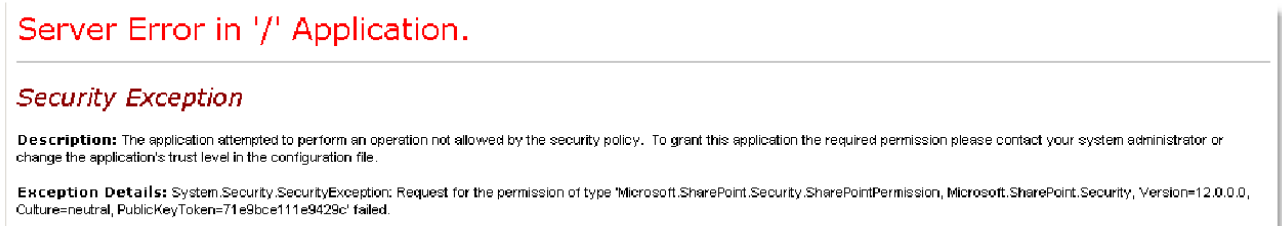


Figure 24. Security Exception

19. Assuming you are taking the second option, open the web.config file related to your web application, the one you added the SafeControl entry too, and change the following

```
<trust level="WSS_Minimal" originUrl="" />
```

To

```
<trust level="WSS_Medium" originUrl="" />
```

20. Reload the page and you should see an error like the Figure 25. URI Format Exception. This error is one that escaped us during our initial development; we'll go back and add another test for this and handle the exception. We're going to leave you to work out this test as it's really very similar to the previous exception test, if you need a hint download the code as it's included.

### Server Error in '/' Application.

*Invalid URI: The URI is empty.*

**Description:** An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

**Exception Details:** System.UriFormatException: Invalid URI: The URI is empty.

**Source Error:**

An unhandled exception was generated during the execution of the current web request. Information regarding the origin and location of the exception can be identified using the exception stack trace below.

Figure 25. URI Format Exception

21. Rebuild (CTRL+SHIFT+B), refresh your web page and you should see the error message you just coded as in Figure 26. URI format exception handled message.

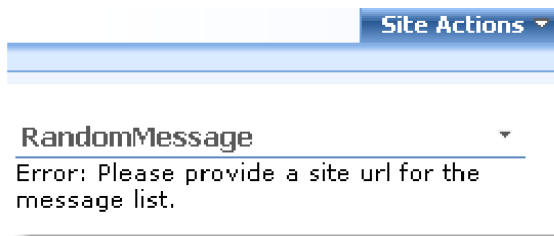


Figure 26. URI format exception handled message

22. Two things left to do, edit the web part and set the URL to the current site as in Figure 27. Site Url Web Part Property

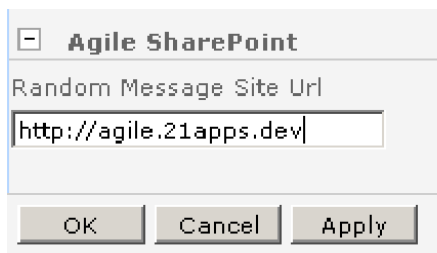


Figure 27. Site Url Web Part Property

23. Create a new Custom List, no need to add any additional columns as our message is based on the Title column.

- Go to Site Actions and click Create
- Choose Custom List
- Give the list a Name **Messages** – this is important as our code looks for a list with a name of Messages.

24. Add some items to the list, you probably have a better imagination than me with my Random Message 1, Random Message 2 entries 😊
25. Go back to your page with the web part and Voila! You should have a random message displayed as in Figure 28. Random Message Web Part. Refresh the page a few times and wonder in awe at your killer web part!

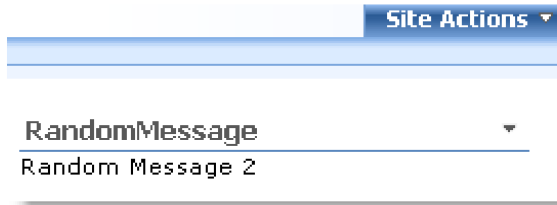


Figure 28. Random Message Web Part

If you've been diligent you may have noticed that there is a scenario our code does not cope with very well, if we have no items in the list our web part reports that the Messages list does not exist? We'll leave that to you to work through.

## Conclusion

In this white paper we looked at Unit Testing the SharePoint Object Model, to achieve this we introduced the excellent Typemock Isolator to enable us to fake our calls to SharePoint. Now you can write unit tests that both run quickly and also, more importantly, do not have a dependency on any specific installation/configuration of SharePoint on your development machine. The introduction of the AAA API in Typemock, specifically the ability to return recursive fakes, has meant that our SharePoint Unit Tests are significantly less brittle, easier to code and generally just brilliant.

## Feedback

Hopefully you have gained some valuable information from this white paper, we welcome any comments or suggestions for improving it or areas that you think need to be addressed, please post any feedback on this white paper at

<http://www.21apps.com/agile/unit-testing-sharepoint-getting-into-the-object-model/>

I look forward to your comments.

## Revision History

Version	Date	Change	By
V0.1	9 Nov 2008	Completed Draft	Andrew Woodward
V0.2	9 Nov 2008	AW Review	Andrew Woodward
V1.0	12 Nov 2009	Reformat, added additional Sidebars on Typemock API	Andrew Woodward

## Copyright

This white paper was written by Andrew Woodward, Principal Consultant for 21apps in the United Kingdom. You are free to distribute this white paper and use any of the code samples as long as you keep this copyright section and refer to the original post and author.