Unit Testing Asp.NET Applications with Moles and Pex

Tutorial for Writing Isolated Unit Tests

Version 0.93 – August 3, 2010

Abstract

This tutorial provides an introduction to writing isolated unit tests for Asp.NET applications by using:

- **Microsoft Moles 2010**, which supports unit testing by providing isolation by way of detours and stubs. The Moles framework is provided with Microsoft Pex, or can be installed by itself as a Microsoft Visual Studio® 2010 add-in.
- **Microsoft Pex 2010**, which automatically generates test suites with high code coverage. Microsoft Pex is a Visual Studio add-in for testing .NET Framework applications.

This tutorial is Technical Level 300. This tutorial assumes you are familiar with developing .NET applications and are building solutions with Asp.NET. To take best advantage of this tutorial, you should first:

- Install the Pex and Moles software and review concepts in “Getting Started with Microsoft Pex and Moles”
- Practice with basic Pex and Moles capabilities, as described in “Unit Testing with Microsoft Moles” and “Exploring Code with Microsoft Pex”

Note

- Most resources discussed in this paper are provided with the Pex software package. For a complete list of documents and references discussed, see “Resources and References” at the end of this document.
- For up-to-date documentation, Pex and Moles news, and online community, see http://research.microsoft.com/pex.

Contents

Testing Asp.NET Web Applications with Behaviors 3
Tutorial Prerequisites and Concepts 4
Getting Started with this Tutorial 5
Exercise 1: Creating the Sample Application and Test Projects 6
  Task 1: Create the Application Project 6
  Task 2: Create the Test Project 7
Exercise 2: Adding Behaviors to Isolate the Unit Test 8
  Using Behaved Types to Specify State and Behavior 8
  Task 1: Prepare the Test Project 9
  Task 2: Replacing the HttpContext.Current 11
  Task 3: Setting up the roles 12
Exercise 3: Adding Assertions to the Unit Test 14
  Task 1: Mutation Testing 14
  Task 2: Add Assertions 14
Exercise 4: Refactoring into a Parameterized Unit Test 15
  Task 1: Refactor the Unit Test 15
  Task 2: Write a Unit Test for the Null ContentType Case 16
(Optional) Exercise 5: Exploring the Parameterized Unit Test with Pex 17
  Task 1: Create a Pex Parameterized Unit Test 17
  Task 2: Tune the Pex Instrumentation 18
Exercise 6: Falling off Behaviors, Landing on Moles 19
  Task 1: Investigate a Missing Mole Issue 20
  Task 2: Use Moles to Fix the Test 20
Resources and References 22

Disclaimer: This document is provided "as-is". Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft, IntelliSense, SharePoint, Visual Studio, Windows Server, and Windows are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

© 2010 Microsoft Corporation. All rights reserved.

Testing Asp.NET Web Applications with Behaviors

Asp.NET Applications. Asp.NET is a free web framework that enables great Web applications to be written on top of the .NET framework.

The Unit Testing Challenge. The primary goal of unit testing is to take the smallest piece of testable software in your application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect. Unit testing has proven its value, because it often helps finding many defects in an early phase of the software development.

In order to execute unit tests early, you must isolate your production code from the environment. The most common approach to isolation is to write drivers to simulate a call into that code and create stubs to simulate the functionality of classes that the code uses. This can be tedious for developers and might cause unit testing to have a lower priority in your testing strategy.

It may be difficult to create unit tests for Asp.NET web applications because:

- You cannot execute the intrinsic Asp.NET types, such as **HttpContext**, **HttpRequest**, **HttpResponse**, **HttpRuntime**, without being connected to a live IIS Server.
- The intrinsic objects —including classes such as **HttpContext** and **HttpRuntime**—do not allow you to inject fake service implementations, because these classes are sealed types with non-public constructors.

Abstractions for intrinsic Asp.NET types.
Acknowledging the testability issue, Microsoft has provided a set of abstractions, such as **HttpContextBase**, **HttpRequestBase**, and so on…. These abstraction types allow you to author testable Asp.NET components. They are located in System.Web.Abstractions.dll for .NET 3.5 and in System.Web.dll in .NET 4.0. This document focuses on the untestable intrinsic Asp.NET types that are still widely used in the field and in legacy web applications.

Unit Testing for Asp.NET applications: Behaviors Powered by Pex and Moles.
This tutorial introduces you to processes and concepts for testing Asp.NET applications. The unit testing process uses:

- **Microsoft Moles**—a testing framework that allows you to isolate .NET code by replacing any method with your own delegate, bypassing any hard-coded dependencies in the .NET code.
- **Behaviors for Asp.NET** — a library that redirects Asp.NET API calls to an in-memory model of the actual Asp.NET functionality.
- **Microsoft Pex**—an automated testing tool that exercises all the code paths in .NET code, identifies potential issues, and automatically generates a test suite that covers corner cases.

Behaviors allow you to write robust isolated unit tests and to define tests in terms of state transformations, rather than relying on encoding low-level protocols of record and playback method interactions.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft Pex and the Moles framework help you overcome the difficulty and other barriers to unit testing Asp.NET web applications, so that you can prioritize unit testing in your strategy to reap the benefits of early defect detection in your development cycle.

CAUTION: Microsoft Pex will exercise every path in your code. If your code connects to external resources such as databases or controls physical machinery, make certain that these resources are disconnected before executing Pex; otherwise, serious damage to those resources might occur.

Tutorial Prerequisites and Concepts

Before launching the tutorial, check this list of prerequisites, software requirements, and suggested background information.

Prerequisites

To take advantage of this tutorial, you should be familiar with the following:

- Microsoft® Visual Studio® 2010
- C# programming language
- .NET Framework
- Basic practices for building, debugging, and testing software

Computer Configuration

These tutorials require that the following software components are installed:

- Windows Server® 2008 R2, Windows Vista®, or Windows® 7, or later operating system
- Visual Studio 2010 Professional

  Microsoft Pex and Microsoft Moles also work with Visual Studio 2008 Professional or any later edition that supports the Visual Studio Unit Testing framework.
- Microsoft Moles 2010 or Microsoft Pex 2010, which includes the Moles framework

  See: Moles and Pex download on Microsoft Research site

Getting Help

- For questions, see “Resources and References” at the end of this document.
- If you have a question, post it on the Pex and Moles forum.

Testing Concepts and Visual Studio

- **For experienced developers:** In this tutorial, you will learn new practices for unit testing.
- **For developers new to testing with Visual Studio:** In addition to the concepts introduced in this tutorial, use the following links to learn more about recommended testing practices for .NET, including unit testing practices, as described in the Visual Studio library on MSDN®:

© 2010 Microsoft Corporation. All rights reserved.

- “Unit Testing” in the Visual Studio Developer Center
- Regression tests
- Build verification tests (BVTs)
- Code coverage basics with Visual Studio Team System

Getting Started with this Tutorial

This tutorial introduces you to tools and processes for working around the difficulties of unit testing for Asp.NET applications. Specifically, you’ll learn:

- How to use behaved types.

  Behaved types for Asp.NET provide an in-memory model of Asp.NET intrinsic types that can be used in the context of unit testing. Unit tests written on top of behaved types are isolated, readable, and resilient to refactoring.

- How to use Pex to test Asp.NET applications using parameterized unit tests.

  Pex automatically generates a suite of closed unit tests with high code coverage. Basic processes for using Pex are described in “Exploring Code with Microsoft Pex.”

- How to extend behaved types with Moles.

  Detouring describes a process of intercepting a call to a method and redirecting the call to an alternative implementation—typically a test implementation. With Moles, you can detour any .NET method to user-defined delegates, including those methods based on Asp.NET libraries.

  You will learn how to use the Moles instrumenting profiler to detour calls to the Asp.NET intrinsic types. These detours can be used later on to bypass the Asp.NET intrinsic types and fake its behavior.

About the Sample Application in the Tutorial. In this tutorial, you test a simple Asp.NET Page that customizes the page title using authentication and caching as follows:

```
public void RenderTitle()
{
  if(this.User.IsInRole("friends"))
  {
    string title = (string)HttpContext.Current.Cache["data"];
    if(title == null)
      this.Cache["data"] = title = "Welcome friend";
    this.Title = title;
  }
  else
    this.Title = "Welcome stranger";
}
```


© 2010 Microsoft Corporation. All rights reserved.

The **RenderTitle** method represents a typical challenge for unit testing with Asp.NET applications if you are not using the abstraction classes. This method relies on **Page.User** and **HttpContext.Current** which should be used only in the context of a web request processed by an IIS server. The value of **IsInRole** is computed by the Asp.NET authentication mechanism and cannot be changed easily by the code. Moreover, the method uses the Asp.NET cache which requires IIS and introduces non-determinism in the code execution. As a final twist, the cache is accessed through both the **HttpContext.Cache** property and the **Page.Cache** property.

Important: Our sample code does not follow Asp.NET application best practices. This code is intended to showcase features of the Pex and Moles framework. The sample code might contain misuses of the Asp.NET methods in order to illustrate testing challenges.

Exercise 1: Creating the Sample Application and Test Projects

This first exercise describes the straightforward steps to create the sample application used in the rest of the tutorial and to create the test project that will be used with Pex and Moles.

Note:

- If you already program in Visual Studio, you can review these steps briefly to implement the sample code.
- If you are relatively new to programming in Visual Studio, these steps will guide you through the details of the Visual Studio interface.

Task 1: Create the Application Project

First, you must create a new project for the sample.

To create the sample application project

1. In Visual Studio, click **File** > **New** > **Project**.
2. In the left pane of the **New Project** dialog, click **Visual C#**, then click **Web**. In the center pane, click **Asp.NET Web Application**.
3. Rename the project **HelloWeb** and click **OK**.
4. In the **Solution Explorer** window of Visual Studio, right-click **Default.aspx** in the HelloWeb project, and click **View Code**.
5. In the file Default.aspx.cs, replace the body of the class with the following code, which is our basic sample application:

```
public partial class _Default:System.Web.UI.Page
{
  protected void Page_Load(object sender,EventArgs e)
  {
    this.RenderTitle();
  }

  public void RenderTitle()
  {
    if(this.User.IsInRole("friends"))
```


© 2010 Microsoft Corporation. All rights reserved.

```
    {
      string title = (string)HttpContext.Current.Cache["data"];
      if(title == null)
        this.Cache["data"] = title = "Welcome friend";
      this.Title = title;
    }
    else
      this.Title = "Welcome stranger";
  }
}
```

5. Press F5. The project should build successfully and launch a web browser that opens the default page. The title of this page should be 'Welcome stranger' which shows that our code was executed.
6. Close the web browser.

Task 2: Create the Test Project

First, you must create a new test project for the sample.

To create the test project

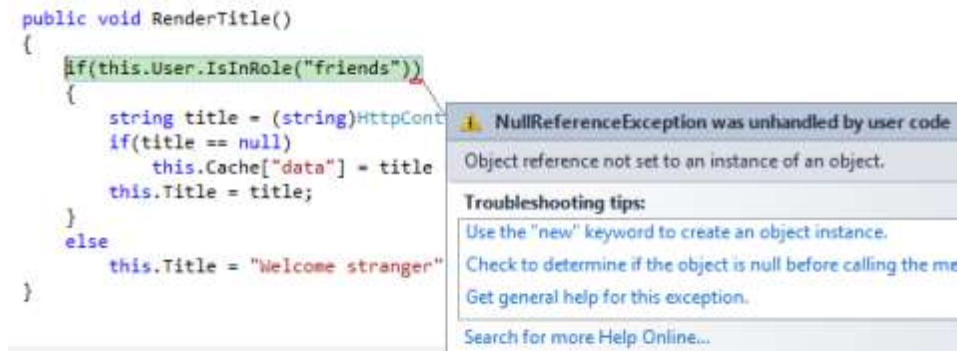
1. In Visual Studio, click **File** > **Add** > **New Project**. Make sure to use **Add** instead of **New**.
2. In the left pane of the **Add New Project** dialog, click **Visual C#**, then click **Test**. In the center pane, click **Test Project**.
3. Rename the Project **HelloWeb.Tests** and click **OK**.
4. In the **Solution Explorer** window of Visual Studio, right-click **HelloWeb.Tests** and click **Add Reference…**.
5. In the **Projects** tab, select the **HelloWeb** project and click **OK**.
6. Open the **Add Reference** dialog again, select the **.NET** tab and select the **System.Web** assembly. Click **OK**.
7. Rename the UnitTest1.cs file to **DefaultTest.cs**. Click **Yes** in the dialog asking whether the 'UnitTest1' code element should be renamed as well.
8. In the file **DefaultTest.cs**, replace the body of the class with the following code, which is our basic unit test:

```
[TestClass]
public class DefaultTest
{
  [TestMethod]
  public void RenderTitle()
  {
    _Default page = new _Default();
    page.RenderTitle();
  }
}
```

9. In the **Solution Explorer**, right click the **HelloWeb.Tests** project and click **Set As Startup Project**. Press F5 to launch the test execution under the debugger.

© 2010 Microsoft Corporation. All rights reserved.

10. The execution should stop at the this.User.IsInRole line with a NullReferenceException. The failure occurs because the User object is only initialized when we run in the context of IIS.



Exercise 1 Summary

In this exercise you created a sample project and a test project. You have tried to write a unit test against the method and encountered the challenges of unit testing. You will begin testing this project with the Moles framework in the next exercise.

Exercise 2: Adding Behaviors to Isolate the Unit Test

Before starting to write the test code, first recall what you are trying to test, leaving aside the technical details:

You are trying to test the **RenderTitle** method of the **_Default** class. This method should update the page title to “Welcome friend” if the user belongs to the “friends” role; otherwise leave the title as “Welcome stranger”.

The **RenderTitle** method presents several challenges for testing:

- **this.User** is a read-only property whose we cannot influence for testing purposes.
- **this.User** is **null** by default.
- **HttpContext.Current** is a property that can only run under IIS.
- **Cache** requires the IIS runtime to execute correctly.

Therefore, we will use behaved types from the Moles framework to isolate the code. Under the hood, behaved types use mole types to detour Asp.NET API.

The following sections describe how to use behaved types to specify the state of the environment for reuse in subsequent tests.

Using Behaved Types to Specify State and Behavior

A behaved type mimics a real type for testing purposes. A behaved type provides a way to create an instance of a type in a particular state and provides a set of behaviors that mimic the behaviors of that instance. Behaved types

© 2010 Microsoft Corporation. All rights reserved.

allow you to specify the state of the environment and can be reused in all tests. When a software change breaks the behaved type-based tests, only the behaved types need to be updated, not the tests relying on them.

As an example, the following code snippet sets the user as authenticated in the Asp.NET context using behaved types:

```
using System.Web.Behaviors;
using System.Security.Principal.Behaviors;

…

BHttpContext context = BHttpContext.SetCurrent();
BIPrincipal user = context.SetUser();
user.Roles.SetOne("friends");
```

The above snippet showcases key conventions of behaved types:

- Behaved types are named after the type they are modeling. They are located in a sub-namespace **System.Web.Behaviors** for the types from the **System.Web** namespace. Behaviors start with **B** followed by the type name.
- Behaved types have implicit conversions to their runtime type. For example, BHttpContext implicitly converts into HttpContext.
- Behaved types are built from behaved collections that the Moles Framework provides. These collections define a new family of operators to set the state, such as SetAll or SetOne. For example, Roles is a BehavedSet<string>.
- Internally, behaved types use Moles to redirect Asp.NET API calls to their fake state.

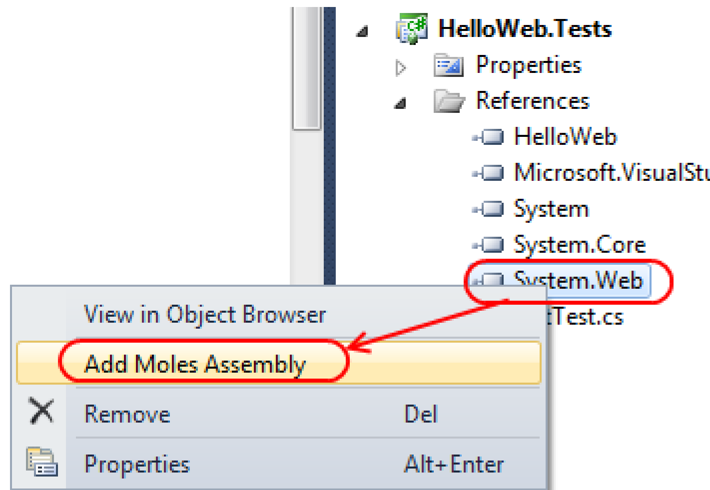
Task 1: Prepare the Test Project

To start, you need to add the Behaviors and Moles assemblies for mscorlib.dll and System.Web.dll which contains the Asp.NET types.

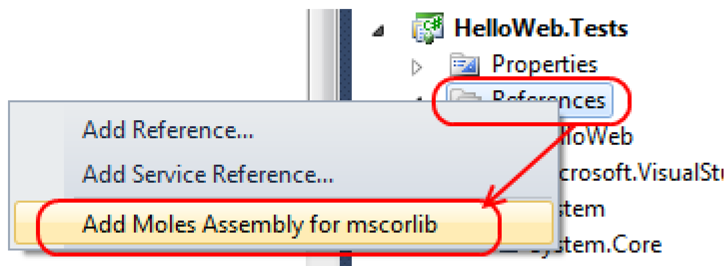
To prepare mole types and behaved types for Asp.NET

1. In the **Solution Explorer** window of Visual Studio, open the **HelloWeb.Tests** project, open the **References** node, right-click the **System.Web** reference and click **Add Moles Assembly**.

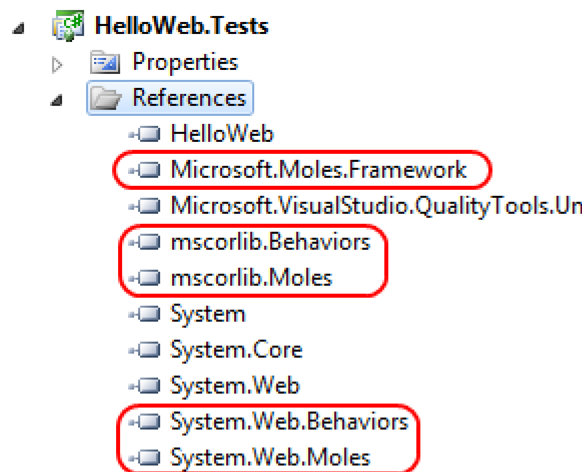
© 2010 Microsoft Corporation. All rights reserved.



2. Since we might need the behaved types for other .NET core types later on, e.g. IPrincipal, let's add behaved types for mscorlib.dll. Open the HelloWeb.Tests project, right-click the References node and click on **Add Moles Assembly for mscorlib**.



3. The Microsoft.Moles.Framework, mscorlib.Moles, mscorlib.Behaviors, System.Web.Moles and System.Web.Behaviors references have been added to your project



Your test project is ready for isolated Asp.NET unit testing.

© 2010 Microsoft Corporation. All rights reserved.

Task 2: Replacing the HttpContext.Current

In this task, you use the behaved type of HttpContext to start testing RenderTitle.

To set HttpContext.Current to its behaved type

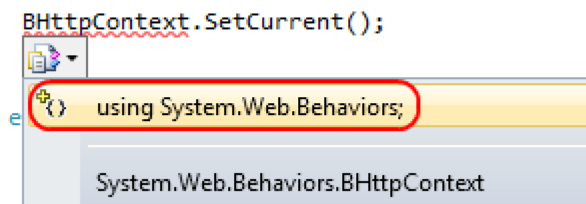
1. In Visual Studio, open **DefaultTest.cs**.
2. Update the RenderTitle method to call BHttpContext.SetCurrent() as follows:

```
using System.Web.Behaviors;

[TestClass]
public class DefaultTest
{
  [TestMethod]
  public void RenderTitle()
  {
    // Arrange
    BHttpContext context = BHttpContext.SetCurrent();
    _Default page = new _Default();

    // Act
    page.RenderTitle();
  }
}
```

Tip: Automatically adding missing namespaces. When you type the name of a type for which the namespace has not yet been imported, the editor might show a red square at the bottom left of the type name. If so, press the **CTRL+.** key combination to open the IntelliSense® menu, and then click the first menu item.



3. Right-click in the RenderTitle method and click **Run Tests**.

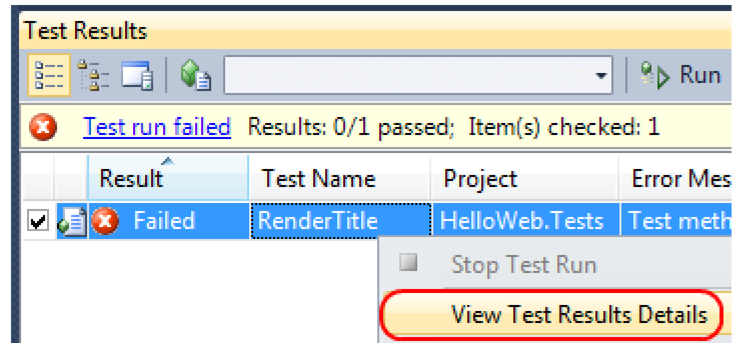
```
[TestMethod]
public void RenderTitle()
{
    // Arrange
    BHttpContext

    // Act
    _Default page
```

Refactor
Organize Usings
Run Tests
Generate Sequence Diagram

4. After the test runs, the **Test Results** view will appear with one test failure. Right-click the test failure row and click **View Test Results Details** to see what happened.

© 2010 Microsoft Corporation. All rights reserved.



5. The Moles Framework complains that the process is not "instrumented." In a nutshell, Moles are implemented through runtime instrumentation where a .NET profiler injects detours in method bodies just before those methods are compiled by the runtime.

6. Fix the issue above by adding the following HostType attribute to the test method, and then run the test again.

```
[TestMethod]
[HostType("Moles")]
public void RenderTitle()
{
```

7. Open the Test Result Details from the failing test and inspect the stack trace.

   Observe that a BehaviorMissingValueException was thrown when calling IsInRole in the RenderTitle method.

Error Stack Trace

```
Microsoft.Moles.Framework.Behaviors.BehavedCollection`1.EnsureArray()
Microsoft.Moles.Framework.Behaviors.BehavedCollection`1.Enumerator.MoveNext()
Microsoft.Moles.Framework.Behaviors.BehavedSet`1.Contains(T item)
System.Security.Principal.Behaviors.BIPrincipal.System.Security.Principal.IPrincipal.IsInRole(String role)
C:\Users\jhalleux\Documents\Visual Studio 2010\Projects\HelloWeb.Tests\mscorlib.Behaviors\System\S
HelloWeb._Default.RenderTitle() C:\Users\jhalleux\documents\visual studio 2010\Projects\HelloWeb\De
HelloWeb.Tests.DefaultTest.RenderTitle() C:\Users\jhalleux\documents\visual studio 2010\Projects\Hell
```

8. Click the **Default.aspx.cs** link and observe that the exception occurred at the following line:

```
if(this.User.IsInRole("friends"))
```

The BehaviorMissingValueException exception is thrown whenever the code requests a state that is not defined. In this case, the unit test does not yet specify the set of roles to which the user belongs so the test raises the exception.

Task 3: Setting up the roles

In this task, you will set the user roles so that he belongs to the "friends" role.

To set user roles

1. In Visual Studio, open **DefaultTest.cs**.

2. In the RenderTitle method, update the Arrange section with the following code:

© 2010 Microsoft Corporation. All rights reserved.

```
using System.Security.Principal.Behaviors;

…

[TestMethod]
[HostType("Moles")]
public void RenderTitle()
{
  // Arrange
  // set the current context
  BHttpContext context = BHttpContext.SetCurrent();
  // set the current user
  BIPrincipal user = context.SetUser();
  // user belongs to one role, friends
  user.Roles.SetOne("friends");
  _Default page = new _Default();
```

The comments in the updated test describe the intention of each line. Notice how it follows our initial goal to create a particular state of Asp.NET. The SetOne method specifies the entire state of the roles collections. It is typical of code that uses behaved types.

Tip: The C# **var** keyword can save you a lot of typing when defining locals in your tests. It is type safe and helps readability.

```
// set the current context
var context = BHttpContext.SetCurrent();
// set the current user
var user = context.SetUser();
```

3. Run the test and observe that it still fails.

4. Open the **Test Results Details** view and inspect the stack trace. The test fails when trying to access the Cache:

```
string title = (string)HttpContext.Current.Cache["data"];
```

5. Update the test case by getting the behaved type of the Cache through the Cache property then setting the state of the cache as empty through the Items property:

```
BCache cache = context.SetCache();
cache.Items.SetEmpty();
```

6. Run the test again and observe that it still fails but at a different location.

```
this.Cache["data"] = title = "Welcome friend";
```

As you may notice, we are getting further and further in the execution and isolating along the way.

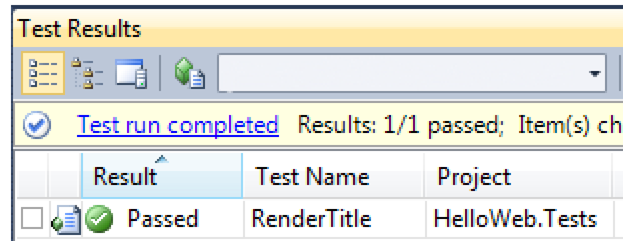
7. The previous test failed because the test accessed a base method provided by the Page class. To replace this method, we instantiate a BPage instance “around” the current page.

```
_Default page = new _Default();
BPage basePage = new BPage(page);
```

Note that we must also import the System.Web.UI.Behaviors namespace which defines BPage.

8. Run the test again and observe that it passes. You have successfully written an isolated unit test for Asp.NET!

© 2010 Microsoft Corporation. All rights reserved.



Exercise 2 Summary
In this exercise you used the behaviors to isolate the unit test.

Exercise 3: Adding Assertions to the Unit Test

In the previous exercise, you learned how to write an isolated unit test. However, the unit test cannot verify any properties about the program, because the test did not contain assertions. In this exercise, you will learn how to write assertions with behaviors to verify the correctness of the code.

Task 1: Mutation Testing

One way to evaluate the quality of a test suite is to seed the implementation with bugs. If the test suite does not find the bugs, you need more tests or better tests.

To seed a test suite with bugs

1. Open the **Default.aspx.cs** file and assign a random string to the Title field:

```
string title = (string)HttpContext.Current.Cache["data"];
if(title == null)
  this.Cache["data"] = title = "Welcome friend";
// this.Title = title;
this.Title = "????";
```

2. Run the test and observe that it still passes.

Why did the test pass? The reason is that the test did not verify that the Title field was actually updated correctly. In the next task, you will add an assertion in the test to verify this property.

Task 2: Add Assertions

After the call to RenderTitle, you want to assert that the value of Title is equal to “Welcome friend”.

To add an assertion

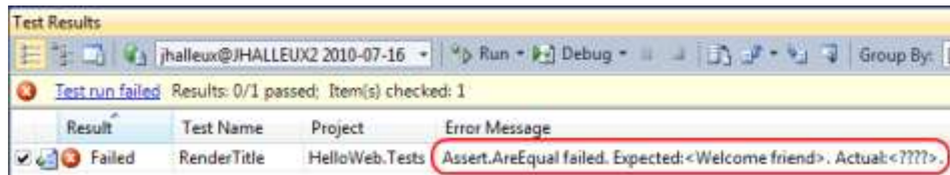
1. Go to the RenderTitle test method and add an Assert section to the test following the Act section:

```
// Act
page.RenderTitle();

// Assert
string title = page.Title;
Assert.AreEqual("Welcome friend",title);
```


© 2010 Microsoft Corporation. All rights reserved.

2. Run the test and observe that it fails with an assertion violation.



At this point, we know that the unit test is actually testing the value of the title. It is time to fix the implementation.

3. Go back to the RenderTitle method and undo the bug.
4. Run the test and observe that it passes again.
5. (optional) Turn on Code Coverage and run the tests. Investigate the code coverage.

Exercise 4 Summary

In this exercise you have injected a bug in the application to assess the quality of the unit test. You have used an assertion to validate that the code implementation follows the specification.

Exercise 4: Refactoring into a Parameterized Unit Test

In the previous exercise, you wrote a unit test that exercises the case where the user is in the “friends” role and the title was not cached. Other possible cases need to be tested as well, e.g. where the user is not authenticated:

```
if (this.User.IsInRole("friends"))
…
else
    this.Title = "Welcome stranger";
```

In this exercise, you will refactor the unit test into a parameterized unit test, reusing the already achieved isolation, in order to achieve greater code coverage.

Task 1: Refactor the Unit Test

You will use the Extract Method refactoring in Visual Studio to create a parameterized unit test from the unit test.

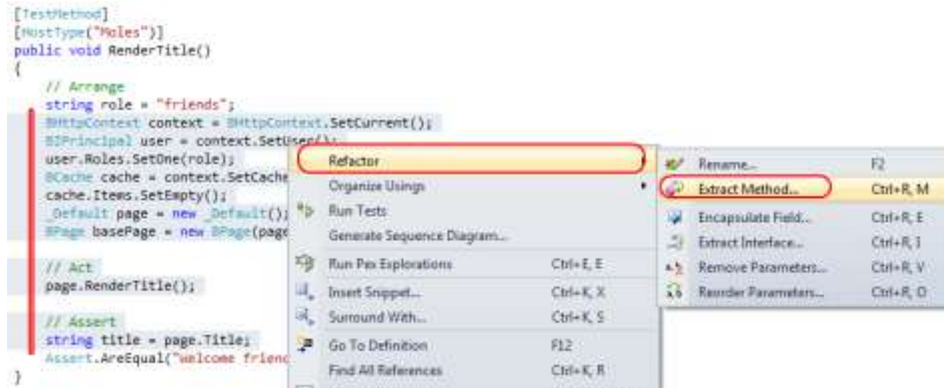
To refactor a unit test to create a parameterized unit test

1. In the RenderTitle test method, move the name of the role to the beginning of the method:

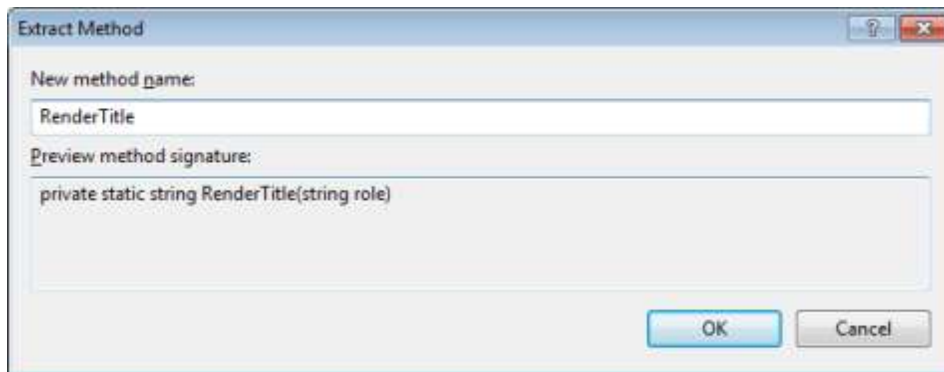
```
public void RenderTitle()
{
    // Arrange
    string role = "friends";
    ...
    user.Roles.SetOne(role);
```

2. Select the method body from the statement following the assignment you just added to the line before the Assert method call. Right-click in the editor, click **Refactor** and then click **ExtractMethod**.

© 2010 Microsoft Corporation. All rights reserved.



3. In the rename dialog, change the method name to RenderTitle and click **OK**.



4. To avoid confusion, rename the original unit test to RenderTitleForFriend.

```
[TestMethod, HostType("Moles")]
public void RenderTitleForFriend()
{
  string role = "friends";
  string title = RenderTitle(role);
  Assert.AreEqual("Welcome friend", title);
}
```

5. Run the test and observe that it still passes.

Task 2: Write a Unit Test for the Null ContentType Case

In this task, you will write a unit test that exercises the scenario where user is not in the “friends” role. You will use the refactored method to achieve this.

To create a unit test for the null ContentType scenario

1. Add the following unit test in the test class:

```
[TestMethod]
[HostType("Moles")]
public void RenderTitleForStranger()
{
    string role = "";
    string title = RenderTitle(role);
    Assert.AreEqual("Welcome stranger", title);
}
```


© 2010 Microsoft Corporation. All rights reserved.

Notice that it has a different assertion that matches the implementation of RenderTitle.

2. Run the newly created test and observe that it passes.
3. (optional) Turn on Code Coverage and run the tests. Investigate the code coverage.

Exercise 4 Summary

In this exercise you have refactored a unit test into a parameterized unit test. You have done a white box analysis by inspecting the code of the implementation to determine relevant values to exercise the code under test.

(Optional) Exercise 5: Exploring the Parameterized Unit Test with Pex

Requirements: Microsoft Pex.

This exercise requires Pex. If you do not have Microsoft Pex installed on your machine, skip this exercise.

Task 1: Create a Pex Parameterized Unit Test

You will convert the parameterized method into a parameterized unit test that Pex can analyze.

To create a parameterized unit test with Pex

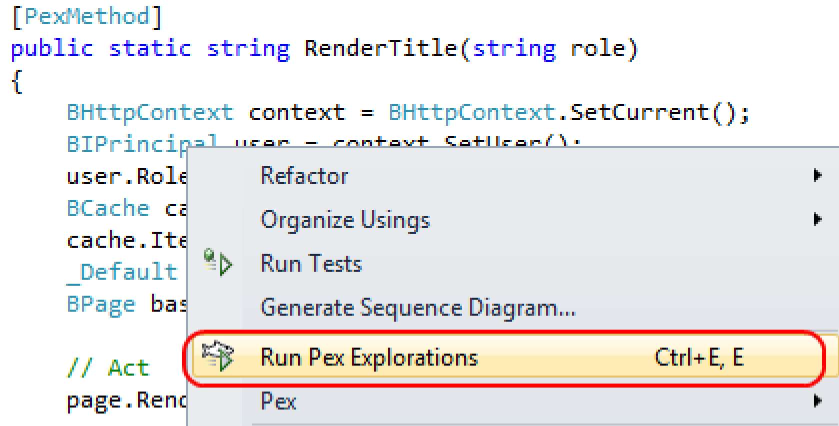
1. In the **Solution Explorer** window of Visual Studio, right-click **HelloWeb.Tests**, and click **Add Reference**.
2. In the **.NET** pane, select **Microsoft.Pex.Framework** and click **OK**.
3. Make the parameterized method RenderTitle public and add the PexMethod attribute:

```
using Microsoft.Pex.Framework;

[PexMethod]
public static string RenderTitle(string role)
{
```

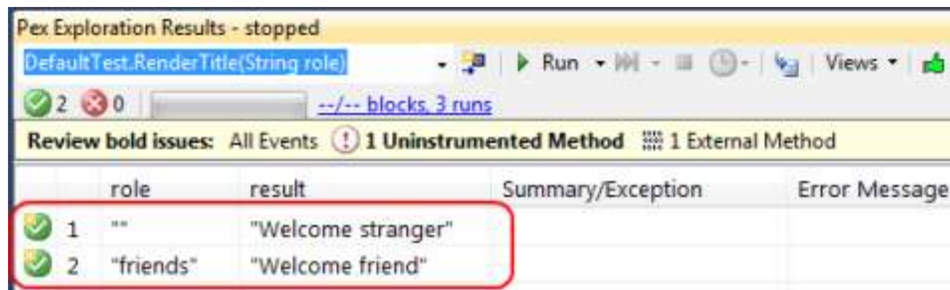
4. Delete the two unit test methods.
5. Right-click in the parameterized test method and click **Run Pex Explorations**.

© 2010 Microsoft Corporation. All rights reserved.



6. Pex will execute and display the **Pex Exploration Results** view.

   In this view, each row is associated with a generated unit test that Pex has added automatically to the test project. Each row is a set of inputs that exercise the program in a different way. The columns are mapped to the inputs and result value of the method. Observe that Pex generated two tests, one where role is “”, and one where the role is “friends”.

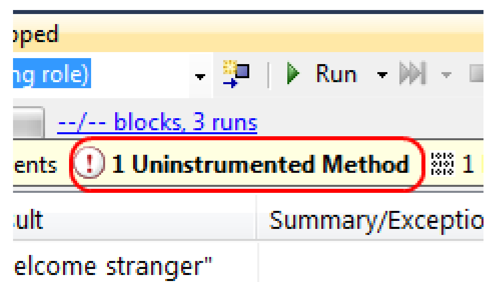


Task 2: Tune the Pex Instrumentation

As with Moles, Pex uses runtime instrumentation to track data and control flow. In this task, you will update the instrumentation options to track the program behavior in the HelloWeb assembly.

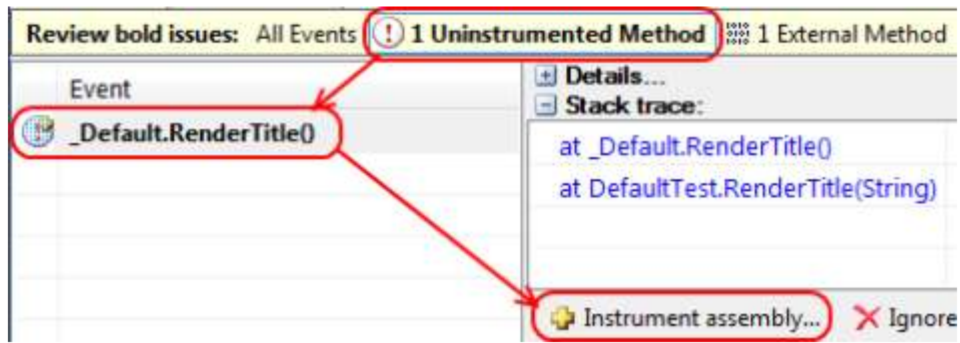
To tune Pex runtime instrumentation

1. In the **Pex Exploration Results** view, click **1 Uninstrumented Method** in the yellow issue strip.



2. Click in the row of the table, and then click **Instrument assembly…** at the lower right of the pane.

© 2010 Microsoft Corporation. All rights reserved.



3. In the **Preview and Apply updates** dialog box, click **Apply**.
4. Open the PexAssemblyInfo.cs file that Pex created in the HelloWeb.Tests test project under the Properties folder and replace the PexInstrumentAssembly attribute with the PexAssemblyUnderTest attribute. This attribute tells Pex that HelloWeb.Tests is the test project of HelloWeb — valuable information that Pex uses in various ways.

```
using Microsoft.Pex.Framework.Instrumentation;
[assembly: PexAssemblyUnderTest("HelloWeb")]
```

5. Run Pex Explorations again and observe that the uninstrumented method message is gone. Also notice that Pex reports the basic block coverage it achieved during the exploration.

Exercise 5 Summary
In this exercise you have used Pex to generate a unit test suite with high code coverage.

Exercise 6: Falling off Behaviors, Landing on Moles

The behaved types provide a simple model of Asp.NET. However, they are not complete. At some point, you are likely to invoke an API that has not been hooked up.

In that case, you will probably receive a MoleNotImplementedException from the Moles Framework. It notifies you that you attempted to invoke a method that was not moled.

You have two options to solve these issues:

- Go into the behavior sources and add support for the missing API.

© 2010 Microsoft Corporation. All rights reserved.

- Use Moles to fix the issue locally in the test. In this exercise, we will show how to fall back on Moles to fix an issue in the context of a unit test.

Task 1: Investigate a Missing Mole Issue

You will add a call to an unsupported API in the implementation and identify which methods need to be moled. To perform this task, start with a version of the HelloWeb project that you had at the end of Exercise 3.

To identify methods that need moles

1. Modify the RenderTitle method and add the code below. This piece of code has no real relevance beyond showcasing an incomplete behaved type.

```
public void RenderTitle()
{
    if(HttpContext.Current.Error != null)
        return; // give up
```

2. Run the tests and observe that they fail with a MoleNotImplementedException. This exception specifies that the method HttpContext.get_Error() was not moled yet.

Error Message

Test method HelloWeb.Tests.DefaultTest.RenderTitle894 threw exception:
Microsoft.Moles.Framework.Moles.MoleNotImplementedException:
HttpContext.get_Error() was not moled.

3. Use the stack trace to take the editor to the line that had the issue. This usually helps you to understand how a method needs to be replaced. Double-click the run result to go to that line.

```
public void RenderTitle()
{
    if(HttpContext.Current.Error != null)
        return; // give up
    if(this.User.IsInRole("friends"))
    {
```

Task 2: Use Moles to Fix the Test

You will instantiate a mole of HttpContext in order to mole the getter of the Error property.

To instantiate a mole to fix a parameterized unit test

1. Open the **DefaultTest.cs** file and update with the following:

```
BHttpContext context = BHttpContext.SetCurrent();
new MHttpContext(context)
{
    ErrorGet = () => null
};
```


© 2010 Microsoft Corporation. All rights reserved.

Note that you need to import the System.Web.Moles namespace.

2. Run the test and observe that it passes again.

Exercise 7 Summary

In this exercise you used Microsoft Moles to work around a limitation of the behaved types.

© 2010 Microsoft Corporation. All rights reserved.

Resources and References

Pex Resources, Publications, and Channel 9 Videos

Pex and Moles at Microsoft Research
http://research.microsoft.com/pex/

Pex Documentation Site

Pex and Moles Tutorials

	Technical Level:
Getting Started with Microsoft Pex and Moles	200
Getting Started with Microsoft Code Contracts and Pex	200
Unit Testing with Microsoft Moles	200
Exploring Code with Microsoft Pex	200
Unit Testing Asp.NET applications with Microsoft Pex and Moles	300
Unit Testing SharePoint Foundation with Microsoft Pex and Moles	300
Unit Testing SharePoint Foundation with Microsoft Pex and Moles (II)	300
Parameterized Unit Testing with Microsoft Pex	400

Pex and Moles Technical References

Microsoft Moles Reference Manual	400
Microsoft Pex Reference Manual	400
Microsoft Pex Online Documentation	400
Parameterized Test Patterns for Microsoft Pex	400
Advanced Concepts: Parameterized Unit Testing with Microsoft Pex	500

Community

Pex Forum on MSDN DevLabs
Pex Community Resources
Nikolai Tillmann's Blog on MSDN
Peli de Halleux's Blog

© 2010 Microsoft Corporation. All rights reserved.

Terminology

This list summarizes common terms introduced in "Exploring Code with Microsoft Pex" and "Unit Testing with Microsoft Moles."

behaved types
: Behaved types are wrappers around mole types that provide a way to specify a state and a behavior for the environment in a reusable way, rather than specifying a hard-coded sequence of low level API calls with fixed return values.

delegate
: A delegate is a reference type that can be used to encapsulate a named or an anonymous method. Delegates are similar to function pointers in C++; however, delegates are type-safe and secure. For applications of delegates, see Delegates in the C# Programming Library on MSDN.

explorations
: Pex executes repeated runs through the code-under-test, using different test inputs and exploring code execution paths encountered during successive runs, until it executes every path in the code. This phase of Pex execution is referred to as "Pex explorations."

integration test
: An integration test exercises multiple test units at one time, working together. In an extreme case, an integration test tests the entire system as a whole.

mock
: A mock is an object that provides limited simulation of another object for testing a specific scenario. For example, a mock can be created that returns specific error codes that might take too long to occur naturally.

mole type
: The Moles framework provides strongly typed wrappers that allow you to redirect any .NET method to a user defined delegate. These wrappers are called mole types, after the framework that generates them. A method that has been wrapped like this is referred to as moled.

stub type
: Usually a stub type is a trivial implementation of an object that does nothing. In the Moles framework, it is specifically a type generated for interfaces and non-sealed classes, which allows you to redefine the behavior of methods by attaching delegates.

unit test
: A unit test takes the smallest piece of testable software in the application, isolates it from the remainder of the code, and determines whether it behaves exactly as you expect. Each unit is tested separately. Units are then integrated into modules to test the interfaces between modules. The most common approach to unit testing requires drivers and stubs to be written, which is simplified when using the Moles framework.

© 2010 Microsoft Corporation. All rights reserved.