Labs: Unit Testing with Microsoft Fakes

Contents

[Exercise 1: Using Stubs to isolate database access 3](#_Toc503935962)

[Exercise 2: Using Shims to isolate from file system and date 12](#_Toc503935963)

# Exercise 1: Using Stubs to isolate database access

**Estimated Completion Time (20 - 30 min)**

For the following walkthrough, we’re going to use a simplistic ASP.NET MVC4 application. The IntroToStubs.sln solution in the **Hands-on Lab\Exercise 1\start** folder contains only one Controller class. It has no Views (it’s set up to use Razor), and, for this exercise, it won’t require any views. Our job is to implement a simple functional aspect: Provide an Order Summary and Calculation of Total Order Cost. For this task, it’s important to note we do not have any Database defined, nor do we have a need to create one to establish our unit tests and validate our components under test.

Without Stubs, our initial approach would have been:

1. Create Sample Database.
2. Populate with sample data.
3. Create Tests – incorporating sample data queries as needed.

|  |  |
| --- | --- |
| GOAL | In this exercise, we will see how to use Microsoft Fakes Stubs to isolate a database dependency from our Controller class to test proper functional implementation. |

#### Environmental dependencies

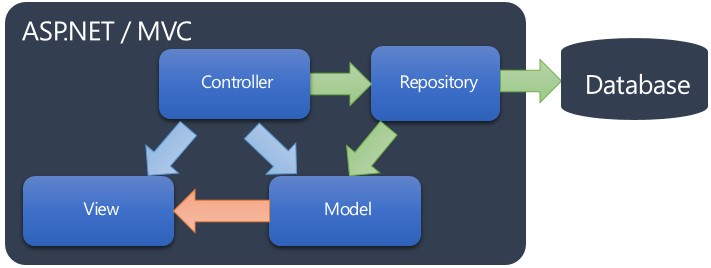
What’s wrong with that approach? Well, what happens if the database technology is a server based relational engine? Remember, unit testing should be small and fast. Additionally, the need for all members of the team to run your unit tests on their machines will require some instance of that RDBMS available to them.

Compounding the issue is that mature development teams leverage Build Services (machines or instances configured with known components, libraries, compilers, scripts.) These build machines may not have access to all of the external dependencies – such as a database or web service.

Environmental dependencies can be a significant blocker and bane to productive development. This is a reason for isolation and, along with the focus on what we need to test, is why Microsoft Fakes provides value.

#### Implementation pattern

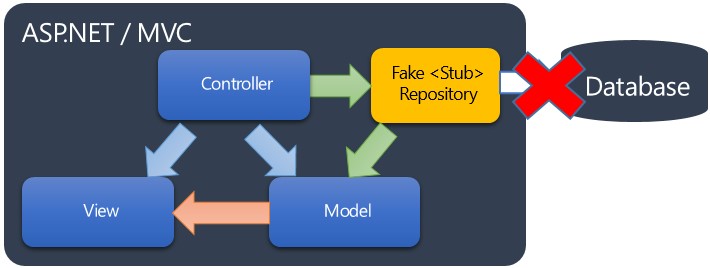
In **Figure 18** you can see the normal interaction of the various classes. In addition, you can see that the coupling of the Repository to the Database is the area we want to isolate. Our intent is to focus our tests on the business logic that, for this example, will reside in the Controller class Action methods.



##### Figure 18 – Environmental dependency

When we isolate, we decouple our implementation of the Repository from the database and, at the same time, we provide known test state and behavior that is then leveraged by the components under test.

In **Figure 19**, the **Fake Stub** is used in place of the real repository, and the test code itself will supply data as required for testing.



##### Figure 19 – Dependency isolation

|  |  |
| --- | --- |
| NOTE | The example is a common approach that leverages default and parameterized constructors in the controller class, along with a Repository pattern. |

#### Overview of steps

1. Add fake assembly for each assembly to be faked.
2. Review and adjust Fakes configuration file(s) [advanced[[1]](#footnote-1)].
3. Set using (C#) or Import (VB) statements as desired to respective Stub namespaces.
4. Provide stub implementation for those classes and methods needed for test method (Arrange) that object or method under test is dependent upon.
5. Provide code to Act upon the object or method under test.
6. Provide code to Assert that expected results occurred.

#### Task 1 – Review starter solution

First, take a quick review of the starter solution, IntroToStubs.sln, which is comprised of two projects.

1. MainWeb – main MVC4 Web Project
2. MainWeb.Tests – Microsoft Unit Testing project

Currently, we have no Test classes defined. Within MainWeb, we will be working with the following classes:

* 1. Controller -> OrderController
  2. Model -> IOrderRepository
  3. Model -> Order
  4. Model -> OrderRepository (implementation of IOrderRepository)

|  |  |
| --- | --- |
| NOTE | For this example, **OrderRepository** represents the concrete implementation with the responsibility of retrieving data from the physical database; however, in this sample, we have left each method as “Not Implemented” – because we’ll be providing Stub implementations for any of the needs for our tests. |

#### Task 2 – Prepare test project for Microsoft Fakes Stubs

We’ll start by setting up our Test project.

1. Select the **MainWeb.Tests** project and then **Add a project Reference** to MainWeb
2. At this point, we need to make sure that the solution compiles. Press F6 to compile the complete solution. This allows the Test Project to pick up the reference and all the types within the **MainWeb** assembly as we move onto the Fake generation.

#### Task 3 – Add the Fakes assembly to the test project

1. Now that the project compiles and we have a reference, we can generate the Fakes assembly for our System Under Test – which is the **MainWeb** Controller classes.
2. In **Solution Explorer**, navigate to the **MainWeb.Tests** project and open the **References** node.
3. Right-click **MainWeb** and choose **Add Fakes Assembly** (see **Figure 20**).

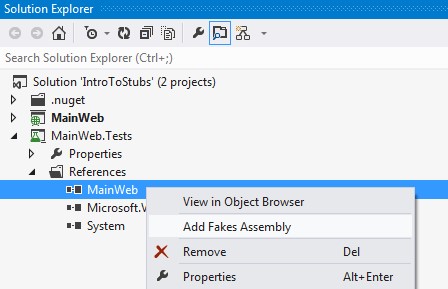
**Figure**

**20**

**–**

**Adding Fakes a**

**ssembly**



1. At this point review the **MainWeb.Tes**t project and folder structure from within Solution Explorer; you should see the following additional **Fakes** node added to the **MainWeb.Tests** project structure with the full name of the **MainWeb** assembly and a **“.fakes”** file extension (see **Figure 21**).

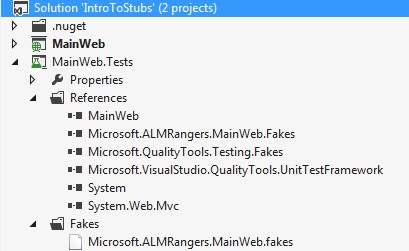
**Figure**

**21**

**–**

**View of test project r**

**eferences**



|  |  |
| --- | --- |
| NOTE | The Fakes framework has generated Stubs and Shims for your Assembly and those types are present in the Microsoft.ALMRangers.MainWeb.Fakes. |

#### Task 4 – Review and update Fakes XML definition file

Let’s take a quick look at the XML file that was generated by adding the Fakes assembly to the Tests project. The contents are sparse, although we will be changing that shortly. 1. Open and review the Microsoft.ALMRangers.MainWeb.fakes file:

<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">

<Assembly Name="Microsoft.ALMRangers.FakesGuide.MainWeb"/> </Fakes>

1. In Solution Explorer, select the Microsoft.ALMRangers.MainWeb.fakes file and then examine the properties (F4) for the file. You’ll notice that the **Build Action** is *Fakes.*
2. **Optional:** Modify the generated file as follows to only create Stubs (no Shims) and to filter on the types we require:

<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">

<Assembly Name="Microsoft.ALMRangers.FakesGuide.MainWeb"/>

<StubGeneration>

<Clear/>

<Add Namespace="Microsoft.ALMRangers.FakesGuide.MainWeb.Models" />

</StubGeneration>

<ShimGeneration Disable="true"/>

</Fakes>

|  |  |
| --- | --- |
| NOTE | The settings shown above illustrate how to slim down the generated assembly by filtering specific types. When you compile, the Microsoft Fakes framework will generate an assembly for your project based on these settings. We do it here to illustrate the narrowed down values that appear in IntelliSense when in the code editor. |

#### Task 5 – Review current model and controller classes in MainWeb

Review the Model classes in the **MainWeb** Models folder. Notice that we’ve used a *Testable* implementation, in which **OrderController** uses an Interface (**IOrderRepository**); this interface allows us to provide a Stub implementation of **IOrderRepository** to **OrderController** as well as provide behavior specific to our isolated testing needs. Beyond that, these are basic CLR classes represent business objects for use by our Business Components during test (see **Code 22**):

##### Code 22 – Starting MainWeb classes public interface IOrderRepository

{

IQueryable<Order> All { get; }

IQueryable<OrderLines> OrderLines(int id);

Order Find(int id);

}

public class Order

{

public Order()

{

this.OrderLines = new HashSet<OrderLines>();

}

public int Id { get; set; } public string CustomerName { get; set; } public double TaxRate { get; set; }

public ICollection<OrderLines> OrderLines { get; set; }

}

public class OrderLines

{

public int Id { get; set; } public string ProductName { get; set; } public double UnitCost { get; set; } public bool IsTaxable { get; set; } public int Quantity { get; set; }

} public class OrderSummaryViewModel

{

public Order Order { get; set; }

public List<OrderLines> OrderLines { get; set; } public double Total { get; set; }

} public class OrderRepository : IOrderRepository

{

public IQueryable<Order> All

{

get { throw new **NotImplementedException**(); }

}

public IQueryable<OrderLines> OrderLines(int id)

{

throw new **NotImplementedException**();

}

public Order Find(int id)

{

throw new **NotImplementedException**();

}

}

#### Task 6 – Create unit test method

We’re ready to start creating our unit tests. For our programming task, we’re going to implement an order line item listing that will simply summarize the total amount of the order.

1. Create a Test Class. Highlight the test project, then from the **Project** menu, choose **Add, Unit Test**.
2. In Solution Explorer, rename the class file. First, select the **OrderControllerTests.cs** file. Then press **F2** or use the context menu. Finally, enter **OrderControllerTests**. This should prompt you to rename the class. Choose **yes**.
3. In the editor, rename **TestMethod1** to **OrderController\_orderSummaryTotalCheck\_equalsSum()**
4. Your test class should look something like this:

using System;

using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Microsoft.ALMRangers.MainWeb.Tests

{

[TestClass]

public class OrderControllerTests

{

[TestMethod]

public void OrderController\_orderSummaryTotalCheck\_equalsSum()

{

} }

}

#### Task 7 – Arrange the test method and create Stub for Repository Interface

We’re ready to start coding our unit test. Remember that we’re testing the **OrderController** Action Method on the controller and we are isolating our **OrderController** logic from the **Repository** implementation. We’ll stub the Repository.

1. Replace the using statements you have with the following statements at the top of your test class:

using System.Collections.Generic; using System.Linq; using System.Web.Mvc;

using Microsoft.ALMRangers.FakesGuide.MainWeb.Controllers; using Microsoft.ALMRangers.FakesGuide.MainWeb.Models; using Microsoft.VisualStudio.TestTools.UnitTesting;

using ModelFakes = Microsoft.ALMRangers.FakesGuide.MainWeb.Models.Fakes;

These using statements include the **Microsoft.ALMRangers.MainWeb.Models.*Fakes*** namespace. They are the

Types (Stubs and Shims) generated by the Fakes framework during compilation for the Types present in the Assembly and Namespaces (**Microsoft.ALMRangers.MainWeb.Models**). Assembly and Namespaces is the target of the generation. We’ve provided the **Using** alias for **ModelFakes** to make it easier to read the code. You don’t have to use this approach; instead, you can use the full namespace as required either in the using or within the declaration inline.

|  |  |
| --- | --- |
| NOTE | The above **Using** alias for **ModelFakes** is provided to make it easier to read the code; you don’t have to utilize this approach and can use the full namespace as required either in the using or within the declaration inline. |

2. Create an instance of **IOrderRepository.** It will be set to a Stub implementation that you’ll define within the context of this test method:

[TestMethod]

public void OrderController\_orderSummaryTotalCheck\_equalsSum()

{

// arrange

const int TestOrderId = 10;

IOrderRepository repository = new ModelFakes.StubIOrderRepository

{

// lambda code

}

This sets up an instance of an **IOrderRepository** that is a Stub (Fake). Not the real thing. This is where, as required for our unit test, we must now provide an implementation of any methods required for our test. The Stub implementation, as generated by the Microsoft Fakes framework is a standard CLR type – absent of all behavior. That is where you must inject specific code to satisfy your test.

1. At this point, we’ve established an instance of our Stub version of our repository – but we’re not finished. We need to implement two methods on our Stub (Fake) **IOrderRepository** as required for this test.
2. Enter the following code, where we had the // lambda code placeholder to define the Stub for the **IOrderRepository.Find(int)** method:

FindInt32 = id =>

{

Order testOrder = new Order

{

Id = 1,

CustomerName = "smith",

TaxRate = 5

};

return testOrder;

},

|  |  |
| --- | --- |
| NOTE | The property name on the **StubIOrderRepository** type has a signature and name of **FakesDelegates.Func<int, Order> FindInt32**. The Microsoft Fakes framework names each method by appending the type of the Parameter to the method name. Here, since Find on **IOrderRepository** had an Int32 parameter, the Stub name is FindInt32. This is how each property is made unique within the generated Stub type[[2]](#footnote-2). The lambda expressions (path) => { … } and (dir,pattern) => { … } in the code above represents a convenient way to set the delegates to detour to. Instead of using lambdas, we could also use delegates pointing to regular methods. |

1. Provide a fake data generator static method to be used by our test method.

private static IQueryable<OrderLines> GetOrderLines()

{

var OrderLines = new List<OrderLines>

{

new OrderLines { Id = 10, IsTaxable = true,

ProductName = "widget1", Quantity = 10, UnitCost = 10 }, new OrderLines { Id = 10, IsTaxable = false,

ProductName = "widget2", Quantity = 20, UnitCost = 20 }, new OrderLines { Id = 10, IsTaxable = true,

ProductName = "widget3", Quantity = 30, UnitCost = 30 }, new OrderLines { Id = 10, IsTaxable = false,

ProductName = "widget4", Quantity = 40, UnitCost = 40 }, new OrderLines { Id = 10, IsTaxable = true,

ProductName = "widget5", Quantity = 50, UnitCost = 50 },

};

return OrderLines.AsQueryable(); }

1. Enter the following code to provide a Stub for **IOrderRepository.OrderLines(int)** method. It uses the static method **GetOrderLines**.

OrderLinesInt32 = id =>

{

var OrderLines = GetOrderLines();

return OrderLines.AsQueryable();

}

1. Immediately below the closing brace, enter the following code to create an instance of **OrderController** using the parameterized constructor:

var controller = new OrderController(repository);

|  |  |
| --- | --- |
| NOTE | The testability of the solution and components under test influences our choice of Stubs or Shims. Our example works well with Stubs because it uses Interfaces. Interfaces let us inject a different concrete implementation for our test, which is our Fake. Testable implementations use interfaces, abstract, and virtual members that permit generation of Stubs from the Microsoft Fakes Framework. See the Shims exercise for testing the “untestable.” |

#### Task 8 – Call the controller action and assert the results

1. Enter the following code to complete the **OrderController\_orderSummaryTotalCheck\_equalsSum** method:

// act

var result = controller.OrderLines(TestOrderId) as ViewResult; var data = result.Model as OrderSummaryViewModel;

// assert

Assert.AreEqual(5675, data.Total, "Order summary total not correct");

**Code 23**, below, is the complete **OrderController\_orderSummaryTotalCheck\_equalsSum** test:

##### Code 23 – Complete OrderController\_orderSummaryTotalCheck\_equalsSum test method

[TestMethod]

public void OrderController\_orderSummaryTotalCheck\_equalsSum()

{

// arrange

const int TestOrderId = 10;

IOrderRepository repository = new ModelFakes.StubIOrderRepository

{

FindInt32 = id =>

{

Order testOrder = new Order

{

Id = 1, CustomerName = "smith",

TaxRate = 5

};

return testOrder; },

OrderLinesInt32 = id =>

{

var OrderLines = GetOrderLines();

return OrderLines.AsQueryable();

}

};

var controller = new OrderController(repository);

// act

var result = controller.OrderLines(TestOrderId) as ViewResult; var data = result.Model as OrderSummaryViewModel;

// assert

Assert.AreEqual(5675, data.Total, "Order summary total not correct");

}

private static IQueryable<OrderLines> GetOrderLines()

{

var orderLines = new List<OrderLines>

{

new OrderLines { Id = 10, IsTaxable = true, ProductName = "widget1",

Quantity = 10, UnitCost = 10 },

new OrderLines { Id = 10, IsTaxable = false, ProductName = "widget2",

Quantity = 20, UnitCost = 20 },

new OrderLines { Id = 10, IsTaxable = true, ProductName = "widget3",

Quantity = 30, UnitCost = 30 },

new OrderLines { Id = 10, IsTaxable = false, ProductName = "widget4",

Quantity = 40, UnitCost = 40 },

new OrderLines { Id = 10, IsTaxable = true, ProductName = "widget5",

Quantity = 50, UnitCost = 50 },

};

return orderLines.AsQueryable();

}

|  |  |
| --- | --- |
| NOTE | At this point, you can run the test from Test Explorer and you’ll have a Test Failure. Our next task will address the failing test with a working implementation. |

#### Task 9 – Complete the implementation of the controller action

1. Add the following using statement to the OrderController class: using System.Linq;

1. The following code (see **Code 24**) can be pasted into the OrderLines Action for the OrderController:

##### Code 24 – MainWeb OrderController OrderLines action public ActionResult OrderLines(int id)

{

// locate the order by ID via repository var order = this.repository.Find(id);

// get the corresponding orderlines

var orderLines = this.repository.OrderLines(order.Id);

// initialize the calculation values double total = 0d;

double taxRate = order.TaxRate / 100; double taxMultiplier = 1 + taxRate;

// run through the list and just summarize conditionally if taxable or not foreach (var lineItem in orderLines)

{

if (lineItem.IsTaxable)

{

total += lineItem.Quantity \* lineItem.UnitCost \* taxMultiplier;

} else {

total += lineItem.Quantity \* lineItem.UnitCost; }

}

// make the view model and set its properties var viewModel = new OrderSummaryViewModel(); viewModel.Order = order;

viewModel.OrderLines = orderLines.ToList(); viewModel.Total = total;

return this.View(viewModel);

}

#### Task 10 – Run unit test

1. Open Test Explorer and **Build All (F6)** the solution.
2. Once the solution is compiled, Test Explorer should show the single test in the solution:

**OrderController\_orderSummaryTotalCheck\_equalsSum** under the category of **Not Run Tests**.

1. Click **Run All** to execute all tests (for this solution there is 1).
2. After building if necessary, and running the tests, you should see passing indication in **Test Explorer.**

At this point, we’ve validated that the action method of **OrderController** (**OrderLines**) returns a model with a member property **Total** that correctly equals our test data, based upon the expected tax calculation.

|  |  |
| --- | --- |
| REVIEW | In this exercise, we removed the dependency on the physical SQL database implementation and have seen how Microsoft Fakes Stubs can be used to enable testing of components through isolation of dependent components.  You can view the end source code in **Hands-on Lab\Exercise 1\end**. |

# Exercise 2: Using Shims to isolate from file system and date

**Estimated Completion Time (20 - 30 min)**

|  |  |
| --- | --- |
| GOAL | In this exercise, we will see how to use Shims to isolate code under test from dependencies on the file system and the system date. |

#### Scenario

You are a developer in a team within a software development department of an Enterprise. Your team is in charge of maintaining a common logger assembly that is being used throughout all applications of the department.

You have been assigned the task of adding a new feature to a central class of the logger assembly: The LogAggregator. This class can aggregate log files in a given folder and filter the files to only a given number of days in the past.

There are no unit tests in place for that component right now. Yet before changing anything in this central piece of code, you want to make sure not to break anything. Unfortunately, the LogAggregator class has not been designed in a way that would allow you to easily stub out the calls to the file system and system date and pass in test values to it. The code does not provide any means to let a test inject a stub; it’s hiding its implementation.

Therefore, you now want to create your first Shims to get the LogAggregator under test.

#### Task 1 – Review the LogAggregator class

1. Open the **EnterpriseLogger.sln** in **Hands-on Lab\Exercise 2\start** and open the **LogAggregator.cs** class file. You should now see the following code (see **Code 25**) in the code editor:

**Code 25 – LogAggregator.cs, the code to be tested** namespace Microsoft.ALMRangers.FakesGuide.EnterpriseLogger

{

using System;

using System.Collections.Generic; using System.Globalization; using System.IO;

public class LogAggregator

{

public string[] AggregateLogs(string logDirPath, int daysInPast)

{

var mergedLines = new List<string>();

var filePaths = Directory.GetFiles(logDirPath, "\*.log"); foreach (var filePath in filePaths)

{

if (this.IsInDateRange(filePath, daysInPast))

{

mergedLines.AddRange(File.ReadAllLines(filePath));

}

}

return mergedLines.ToArray();

}

private bool IsInDateRange(string filePath, int daysInPast)

{

string logName = Path.GetFileNameWithoutExtension(filePath); if (logName.Length < 8)

{

return false;

}

string logDayString = logName.Substring(logName.Length - 8, 8);

DateTime logDay;

DateTime today = DateTime.Today;

if (DateTime.TryParseExact(logDayString, "yyyyMMdd", CultureInfo.InvariantCulture, DateTim eStyles.None, out logDay))

{

return logDay.AddDays(daysInPast) >= today;

}

return false;

}

}

}

|  |  |
| --- | --- |
| NOTE | The code provided here is only one class for brevity and keeping the lab focused. You can perform all steps in this lab based on this single class. If you happen not to have access to the prepared solution mentioned above you can quickly generate such a solution by creating a new class library project and copying and pasting the above code in. |

#### Task 2 – Create a test project

1. **Add** a new **Visual C#** **Unit Test Project** called “EnterpriseLogger.Tests.Unit” to the **EnterpriseLogger solution.**
2. In “EnterpriseLogger.Tests.Unit” add a reference to the project “EnterpriseLogger.”

#### Task 3 – Create a first test

1. Rename “UnitTest1.cs” to “LogAggregatorTests.cs.”
2. Open “LogAggregatorTests.cs” and add the following line to the using-Block:

using Microsoft.ALMRangers.FakesGuide.EnterpriseLogger;

1. Replace the method “TestMethod1” with the following code:

[TestMethod] public void AggregateLogs\_PastThreeDays\_ReturnsAllLinesFromPastThreeDaysAndToday()

{

// Arrange var sut = new LogAggregator();

// Act var result = sut.AggregateLogs(@"C:\SomeLogDir", daysInPast: 3);

// Assert

Assert.AreEqual(4, result.Length, "Number of aggregated lines incorrect.");

Assert.AreEqual("ThreeDaysAgoFirstLine", result[0], "First line incorrect."); Assert.AreEqual("TodayLastLine", result[3], "Last line incorrect.");

}

4. Right-click somewhere in the test method and choose **Run Tests**. The test is being executed and fails.

The preceding unit test tests the method “AggregateLogs” and states its test goal in the name already, which is to verify that “AggregateLogs” — when called with the path of a directory containing log files and daysInPast 3 as parameters — should return all lines from only the log files from three days ago until today included. However, this test would only work if the directory “C:\SomeLogDir” existed and magically had files in it that matched the requirements of this test. This could be accomplished by using some setup code. However, the resulting test would be an integration test rather than a real unit test, because it would actually use the file system.

To make it a real unit test, we will have to isolate the test from the static methods it calls to access system date and the file system. Let’s review the code under test:

public string[] AggregateLogs(string logDirPath, int daysInPast)

{

var mergedLines = new List<string>();

var filePaths = Directory.GetFiles(logDirPath, "\*.log"); foreach (var filePath in filePaths)

{

if (this.IsInDateRange(filePath, daysInPast))

{

mergedLines.AddRange(File.ReadAllLines(filePath));

}

}

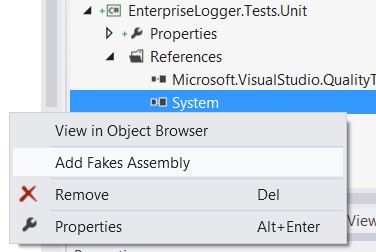
return mergedLines.ToArray();

}

The static method calls highlighted above represent a convenient usage pattern of the File and Directory classes from the System.IO namespace. There might be reasons to use the file system in a different way, which we will discuss later on in the Exercise. Now we’ll shim Directory.GetFiles() and File.ReadAllLines().

#### Task 4 – Add Shims to Fake the file system

1. First, we need to tell Visual Studio for which dependencies we want to have Fakes generated (see **Figure 22**). In **Solution Explorer**, under “EnterpriseLogger.Tests.Unit,” expand **References**, right-click **System** and then choose **Add Fakes assembly**:



##### Figure 22 – Add Fakes assembly

|  |  |
| --- | --- |
| NOTE | Visual Studio created a new folder named “Fakes” containing two XML files and added references to two newly generated assemblies. |

The files in the “Fakes” folder tell Visual Studio which types to shim. You can use these files to customize for which types Shims and Stubs are being generated. The reason why there were two files generated is that the System namespace spans more than one assembly. Yet because mscorlib.dll can’t be referenced directly, to still use Fakes for it, a Fakes assembly for mscorlib.dll is always added when a Fakes assembly for System.dll is added.

1. By convention, the Fakes types for the System.IO namespace are contained in the System.IO.Fakes namespace. To use them conveniently, we’ll work with a using statement:

In **Solution Explorer** double-click “LogAggregatorTests.cs” and insert the following statement in the using section at the top of the file:

using System.IO.Fakes;

1. Change the test method in “LogAggregatorTests.cs” as follows (see **Code 26**). Changes are highlighted in **bold**.

##### Code 26 – Shimming code for isolating the file system

// Arrange

var sut = new LogAggregator();

**ShimDirectory.GetFilesStringString = (dir, pattern) => new string[]**

**{**

**@"C:\someLogDir\Log\_20121001.log",**

**@"C:\someLogDir\Log\_20121002.log",**

**@"C:\someLogDir\Log\_20121005.log",**

**};**

**ShimFile.ReadAllLinesString = (path) =>**

**{**

**switch (path)**

**{ case @"C:\someLogDir\Log\_20121001.log":**

**return new string[] {"OctFirstLine1", "OctFirstLine2"}; case @"C:\someLogDir\Log\_20121002.log":**

**return new string[] {"ThreeDaysAgoFirstLine", "OctSecondLine2"}; case @"C:\someLogDir\Log\_20121005.log": return new string[] {"OctFifthLine1", "TodayLastLine"};**

**}**

**return new string[] {};**

**};**

// Act

var result = sut.AggregateLogs(@"C:\SomeLogDir", daysInPast: 3);

// Assert

Assert.AreEqual(4, result.Length, "Number of aggregated lines incorrect.");

CollectionAssert.Contains(result, "ThreeDaysAgoFirstLine", "Expected line missing from aggregated lo g.");

CollectionAssert.Contains(result, "TodayLastLine", "Expected line missing from aggregated log.");

Let’s review the inserted code. We inserted two statements:

ShimDirectory.GetFilesStringString = [some delegate];

ShimFile.ReadAllLinesString = [some delegate];

These statements tell the Microsoft Fakes framework which methods should be intercepted and which code to detour the calls to instead of the real code. The names are determined by convention again. The name of the class used to access the Shim for a certain type is that type’s name, prefixed with “Shim.” The name of the property used to set the delegate for intercepting calls to a certain method is the name of that method suffixed by the names of the parameter types of the method. This convention enables setting different delegates for the different overloads of a method.

The code we assign to these properties in the example (see **Code 26**) makes the shimmed method

GetFiles(string, string) return three file paths (C:\someLogDir\Log\_20121001.log,

C:\someLogDir\Log\_20121002.log, and C:\someLogDir\Log\_20121005.log) that have dates encoded in their names. (We don’t care about the dir and pattern params here). The shimmed ReadAllLines(string) method returns made up string arrays that represent the lines of the imaginary log files, based on the path parameter.

1. Right-click in the body of the test method and then click **Run Tests**. The test fails.
2. In **Test Explorer** under **Failed Tests**, select the “AggregateLogs\_...” test and review the error message.

The error message tells us to use a ShimsContext. This is needed to scope the usage of shims. The shimming will only take place within this scope. Without this scope, the shims would stay in place for subsequent tests, possibly causing side effects. So, let’s do as advised…

|  |  |
| --- | --- |
| NOTE | Setting up the ShimsContext should always be done in a using statement, although **never** in setup/initialize or teardown/cleanup methods. This would let shims stay in place after the test method itself has been left, potentially affecting the test runner and the desire for granular isolation of each unit test. It would also make all tests in the class slower, even if they do not need the ShimsContext. Finally, it would limit the ability to fine-scope the lifecycle of shims. |

1. In the using block at the top of “LogAggregatorTests.cs” insert the following line:

using Microsoft.QualityTools.Testing.Fakes;

1. Change the test method as follows. (The changes are in **bold**).

using (ShimsContext.Create())

{

// Arrange

…

// Act

…

// Assert

…

}

1. In **Test Explorer** click **Run…**, **Failed Tests**.

The test fails once again with the message, “Assert.AreEqual failed. Expected:<4>. Actual:<0>. Number of aggregated lines incorrect.” This is because the dates encoded in the file names that we provided through the shimmed method GetFiles(string, string) are more than three days old and none of them met the filter. Let’s review the method LogAggregator.IsInDateRange(string,int) that is responsible for filtering the dates (see **Code 27**).

##### Code 27 – Static property to be shimmed in method “IsInDateRange” private bool IsInDateRange(string filePath, int daysInPast)

{ string logName = Path.GetFileNameWithoutExtension(filePath); if (logName.Length < 8)

{

return false;

} string logDayString = logName.Substring(logName.Length - 8, 8); DateTime logDay;

DateTime today = DateTime.Today;

if (DateTime.TryParseExact(logDayString, "yyyyMMdd", CultureInfo.InvariantCulture, DateTimeStyles.N one, out logDay))

{

return logDay.AddDays(daysInPast) >= today;

}

return false;

}

This method relies on a call to the static property Today of the DateTime class. This is what makes the test fail when it’s run on any day but the fifth of October 2012. To make the test pass, we will now shim this property as well.

#### Task 5 – Add Shims to Isolate from the system date

1. In the using block at the top of “LogAggregatorTests.cs” insert the following line:

using System.Fakes;

1. Insert the following line above the //Act comment:

ShimDateTime.TodayGet = () => new DateTime(2012, 10, 05);

1. In **Test Explorer**, click **Run…**, **Failed Tests**. …The test now passes.

#### Task 6 – (Optional) Run test with debugger to understand execution flow

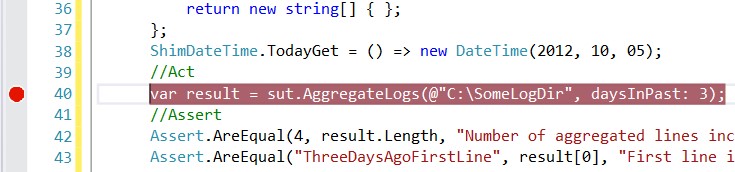
1. Place the cursor on the first line of code below the //Act comment (see **Figure 23**).
2. On the main menu, choose **DEBUG**, **Toggle Breakpoint** to add a breakpoint:

**Figure**

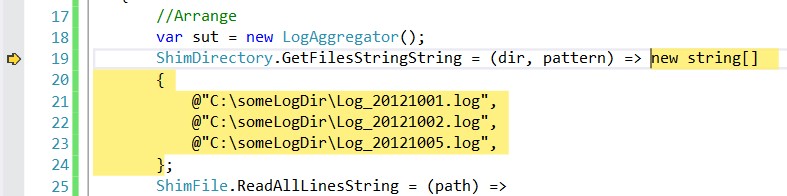
**23**

**–**

**Breakpoint in test method**



1. Right-click somewhere in the body of the test method and then click **Debug Tests**.
2. After the breakpoint has been hit, on the main menu, select **DEBUG**, **Step Into** and remember the shortcut (default is **F11**).
3. Continue stepping through the code (by using the remembered shortcut for **Step Into**) until code execution reaches the first detoured lambda expression (see **Figure 24**).



##### Figure 24 – Code at detour

|  |  |  |
| --- | --- | --- |
| NOTE |  | When using a Shim, everything within the application domain is routed through the shim context; so if you make a call to the shim object from the debug or watch window, you’ll see your shimmed result value rather than the real value. |
| 6. | Continue until all lambdas were passed, then on the main menu, choose **DEBUG**, **Continue**. | |
| REVIEW |  | We’ve now successfully isolated our LogAggregator production code from its dependencies on the file system and the system date — without having to change the production code. You can view the end source code in **Hands-on**  **Lab\Exercise 2\end** |

1. Code generation, compilation, and naming conventions in Microsoft Fakes [- http://msdn.microsoft.com/en-us/library/hh708916.aspx](http://msdn.microsoft.com/en-us/library/hh708916.aspx)  [↑](#footnote-ref-1)
2. Parameter Naming Conventions, Code generation, compilation, and naming conventions in Microsoft Fakes --<http://msdn.microsoft.com/en-us/library/hh708916.aspx> [↑](#footnote-ref-2)