v1.2

Better Unit Testing with Microsoft® Fakes

Visual Studio ALM Rangers

Microsoft Visual Studio

Table of Contents

Foreword

For modern development teams, the value of effective and efficient unit testing is something everyone can agree on. Fast, reliable, automated tests that enable developers to verify that their code does what they think it should, add significantly to overall code quality. Creating good, effective unit tests is harder than it seems though. A good unit test is like a good scientific experiment: it isolates as many variables as possible (these are called control variables) and then validates or rejects a specific hypothesis about what happens when the one variable (the independent variable) changes.

Creating code that allows for this kind of isolation puts strain on the design, idioms, and patterns used by developers. In some cases, the code is designed so that isolating one component from another is easy. However, in most other cases, achieving this isolation is very difficult. Often, it's so difficult that, for many developers, it is unachievable.

First included in Visual Studio 2012, Microsoft Fakes helps you — our developers — cross this gap. It makes it easier and faster to create well-isolated unit tests when you do have systems that are "testable," letting you focus on writing good tests and not on test plumbing. It also enables you to isolate and test code that is not traditionally easy to test, by using a technology called Shims. Shims use runtime interception to let you detour around challenging dependencies and replace them with something you can control. As we have mentioned, being able to create this control variable is imperative when creating high-quality, fast-running unit tests.

Shims provide a very powerful capability that will let you circumvent all kinds of roadblocks when unit testing your code. As with all powerful tools, there are a number of patterns, techniques and other "gotchas" that can take time to learn. This guidance document provides you with a jump-start on acquiring that knowledge by sharing a large number of examples and techniques for effectively using Microsoft Fakes in your projects.

We are happy to introduce this excellent guidance document produced by the Visual Studio ALM Rangers. We are sure that it will help you and your team realize the power and capabilities Microsoft Fakes provides you in creating better unit tests and better code.

Peter Provost – Program Manager Lead, Visual Studio ALM Tools
Joshua Weber – Program Manager, Visual Studio ALM Tools

Introduction

Welcome to *Better Unit Testing with Microsoft Fakes*[1] where we, the ALM Rangers, will take you on a fascinating journey to discover an exciting and powerful new feature provided by Visual Studio 2012.

NOTE

Our guidance is based on Visual Studio 2012 Update 1 and to the best of our knowledge is still applicable to all Visual Studio 2012 updates as well as **Visual Studio 2013**.

Intended audience

Developers! Developers! Developers! We expect the majority of our audience to be developers. We will cover some basics concerning unit testing but anticipate most of our reading audience to have had some degree of previous experience with writing unit tests. Having prior experience with other mocking / isolation frameworks will be beneficial. However, if you are looking to adopt Microsoft Fakes as your first mocking solution, we think you will be able to follow our guidance and make a comfortable and confident start in implementing Microsoft Fakes as a mocking solution. If you are new to terms like unit testing and mocking, we still think that this guide will be useful in introducing you to these important concepts that developers need in their tool belt.

What you'll need

The following editions of Visual Studio support Microsoft Fakes and are referenced in this guide as a *'supported edition'* of Visual Studio

	2012	2013
Visual Studio Ultimate	✓	✓
Visual Studio Premium	✓ Update 2 is required to enable Microsoft Fakes support	✓

To write unit tests using Microsoft Fakes you will need a supported edition of Visual Studio. Executing these tests on a build server will require a supported edition. It is possible to execute tests using Team Foundation Server 2010 and perhaps earlier versions. However, for a seamless experience, we recommend using Team Foundation Server 2012 or later. Tests containing Microsoft Fakes can still be shared and viewed with other team members who are not using a supported edition of Visual Studio but they will not be able to execute them.

Visual Studio ALM Rangers

The Visual Studio ALM Rangers are a special group with members from the Visual Studio Product group, Microsoft Services, Microsoft Most Valuable Professionals (MVP) and Visual Studio Community Leads. Their mission is to provide out-of-band solutions to missing features and guidance. A growing Rangers Index is available online[2].

Contributors

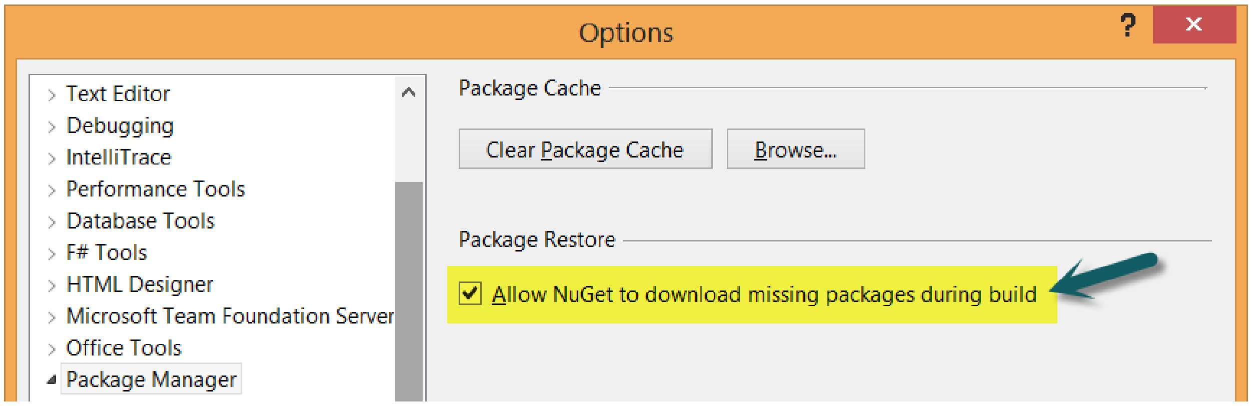
Brian Blackman, Carsten Duellmann, Dan Marzolini, Darren Rich, David V. Corbin, Hamid Shahid, Hosam Kamel, Jakob Ehn, Joshua Weber, Mehmet Aras, Mike Fourie, Patricia Wagner, Richard Albrecht, Richard Fennell, Rob Jarratt, Shawn Cicoria, Waldyr Felix, Willy-Peter Schaub

[1] http://msdn.microsoft.com/en-us/library/hh549175.aspx
[2] http://aka.ms/vsarindex

Using the sample source code, Errata and support

All source code in this guide is available for download via the Visual Studio Test Tooling Guidance[3] home page where we also provide the latest corrections and updates.

Our Hands-on Labs make use of NuGet Package Restore[4]. You will need to enable this via the Visual Studio Options (see **Figure 1**) if you have not already done so.


Options

- Text Editor
- Debugging
- IntelliTrace
- Performance Tools
- Database Tools
- F# Tools
- HTML Designer
- Microsoft Team Foundation Server
- Office Tools
- Package Manager

Package Cache

Clear Package Cache | Browse...

Package Restore

☑ Allow NuGet to download missing packages during build

Figure 1 – NuGet package restore

Lambda expressions

A lambda expression is a concise way to write an anonymous function that you can use to create delegates or expression tree types. By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. We make extensive use of lambda expressions in the sample code. If you are new to the concept of lambda expressions, we recommend you read *Lambda Expressions (C# Programming Guide)* on MSDN[5].

Additional ALM Rangers Resources

Understanding the ALM Rangers – http://aka.ms/vsarunderstand

Visual Studio ALM Ranger Solutions – http://aka.ms/vsarsolutions

[3] http://vsartesttoolingguide.codeplex.com

[4] http://blog.nuget.org/20120518/package-restore-and-consent.html

[5] http://msdn.microsoft.com/en-us/library/bb397687.aspx

Chapter 1: A Brief Theory of Unit Testing

Before we get into the technical details of Microsoft Fakes, let's start with a small amount of theory to refresh our seasoned readers and bring others up to speed on the topic of Unit Testing. There are two major schools of thought among Unit Testing practitioners:

Should you, or should you not, change the design of existing code to make it more testable?

The way you answer that question has a direct impact on how isolation frameworks like Microsoft Fakes are used by the team, the discussion centering on how to isolate the code under test from its dependencies. Some argue that changing the design to make it more testable is a good thing because it helps achieve a more decoupled and cohesive object-oriented design. In this case, special techniques to substitute concrete classes (in Microsoft Fakes this is the Shims feature) are used to test legacy code that was not designed to be testable. Others argue that tests should not compromise the design. In this case, new code tends to make more use of concrete classes; to test the new code requires the special techniques to substitute the concrete classes in unit tests.

Which route you choose is up to you and is outside the scope and intention of this guide. Hence, we make no judgment or recommendation. Regardless of your choice, it is generally accepted that unit tests should be small and fast.

Software testing

Software testing is the art of measuring and maintaining software quality to ensure that user expectations and requirements, business value, non-functional requirements, such as security, reliability and recoverability, and operational policies are all met. Testing is a team effort to achieve the well understood and agreed upon minimum quality bar and definition of "done."

NOTE

Definition of Done – is a team definition and commitment of quality to be applied to the solution during each iteration (this may also occur at the Task or User Story level, too). Consider design, code reviews, refactoring and testing when discussing and defining your definition of done.

Testing strategies

Testing strategies are traditionally divided into black, white, and gray box testing (see **Table 1**).

Strategy	Description
Black	The inside of the box ("solution implementation") is dark. Testers focus only on input and output, typically when performing system and user acceptance testing.
White	The inside of the box is visible and analyzed as part of the testing.
Gray	A combination of black and white box testing typically used to test edge cases, which require an understanding of the internals and expected behavior.

Table 1 – Testing strategies

Testing types

There is no silver bullet when it comes to testing (see **Figure 2**). Sometimes, when human interaction and feedback is needed, automation is required to run targeted feature testing or manual testing repetitively (see **Table 2**).

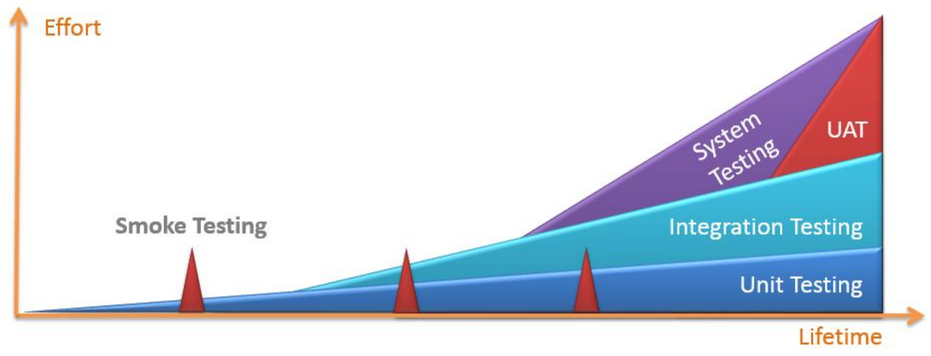


Figure 2 – Testing types and lifetime

Strategy	Description	Visual Studio Tooling
Exploratory Test	Tester tries to think of possible scenarios not covered by other test strategies and tests. Useful when engaging users with the testing and observing their (expected) usage of the system. There are no predefined tests.	Exploratory testing with Microsoft test Manager (MTM)
Integration Test	Testing different solution components working together as one.	Visual Studio Unit Test features
Load Test	Testing under load, in a controlled environment.	Visual Studio Load Test Agent
Regression Test	Regression testing ensures that the system still meets the quality bar after changes such as bug fixes. Uses a mixture of unit tests and system tests.	Automated testing with MTM
Smoke Test	Smoke testing is used to test a new feature or idea before committing the code changes.	
System Test	Testing of the entire system against expected features and non-functional requirements.	Visual Studio Lab Management
Unit Test	A test of the smallest unit of code (method / class, and so on) that can be tested in isolation from other units. See Verifying Code by Using Unit Tests [6] and *The fine line between good and flawed unit testing* in this guide for more information.	Visual Studio Test Explorer Unit Test Frameworks
User Acceptance Test	Toward the end of the product cycles users are invited to perform acceptance testing under real-world scenarios, typically based on test cases.	Automated testing with MTM

Table 2 – Testing types

[6] http://msdn.microsoft.com/en-us/library/dd264975.aspx

The fine line between good and flawed unit testing

Instead of focusing on the flaws of unit testing, we decided to present a concise checklist that will assist you in your pursuit to design, develop, and implement good unit tests.

WARNING

Unit testing focuses on very small bits of code, so used exclusively it probably won't help you improve the overall quality of your product. We recommend using it in conjunction with the other testing strategies described in this guidance. In other words, unit testing is not a magical silver bullet, but an important ingredient in a complete testing strategy.

Why unit testing is important

When someone asks why unit tests are important, we often ask whether they believe it is important that the commercial plane they frequently use to cross vast oceans be tested thoroughly before each flight. Is it important that the plane is flight-tested? Yes. Okay, now is it important that the altimeter is itself well tested? Of course. That is unit testing… testing the altimeter. It does not guarantee the plane will fly, but it really cannot fly safely without it.

The importance of unit testing, which begins at the start of the project and continues throughout the application lifecycle, is important when you are striving for:

- Higher-quality code from the start
- Fewer bugs
- Self-describing code
- Reducing the cost of fixing bugs by fixing them earlier, rather than later
- Solution responsibility and ownership, through pride down to the bits and bytes

NOTE

The core value of test-driven development and unit testing is to ensure that the team **thinks** and even **dreams** about the code, **scrutinizes** the requirements, and **pro-actively** avoids problems and **scope-creep.**

Automate your tests
Consider automating your tests. Automated testing lets you quickly and automatically test code changes on an on-going basis. Automated testing brings its own challenges, however, and we encourage you to investigate this topic[7].

Define a unit test driven development paradigm

We recommend that you consider a **RED** (fail) to **GREEN** (succeed) unit test driven development strategy, especially suitable for agile development teams.

Once you understand the business requirement and intent for the unit test, proceed as follows (see **Figure 3**):

[7] http://blogs.msdn.com/b/steverowe/archive/2008/02/26/when-to-test-manually-and-when-to-automate.aspx

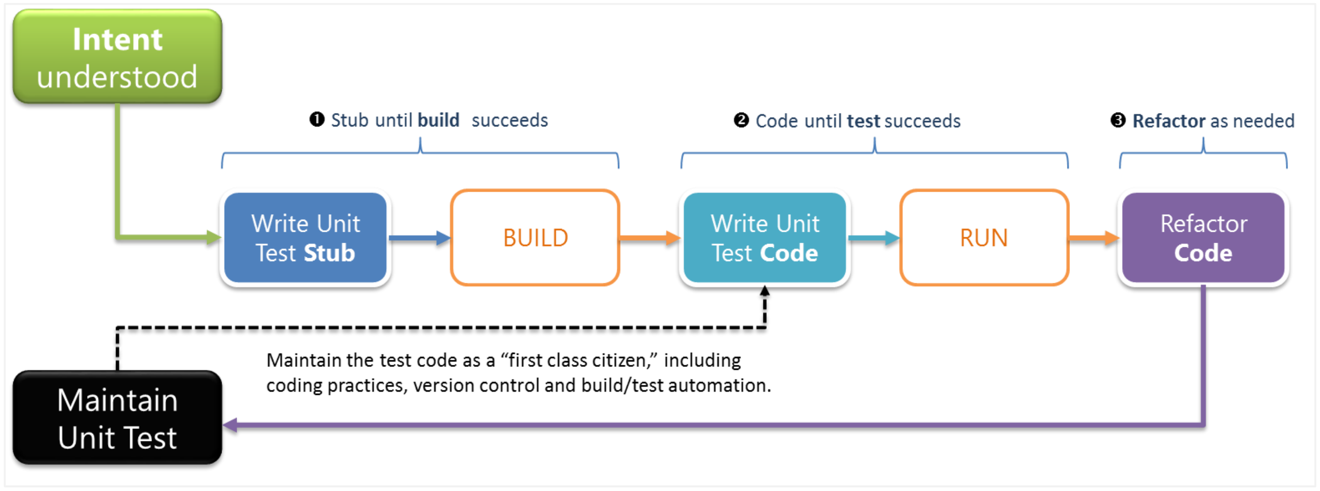

Intent understood
❶ Stub until build succeeds
❷ Code until test succeeds
❸ Refactor as needed
Write Unit Test Stub
BUILD
Write Unit Test Code
RUN
Refactor Code
Maintain the test code as a "first class citizen," including coding practices, version control and build/test automation.
Maintain Unit Test

Figure 3 – Stub versus Shim in your solution

1. **Stub** the test code first from **RED** to **GREEN.**
   - Initially the build will fail **RED** due to missing feature code.
   - Implement just enough code to get the build to succeed **GREEN** (no real implementation yet).
2. **Code** the test code from **RED** to **GREEN.**
   - Initially the test will fail **RED** due to missing functionality.
   - Implement the new requirement defined by the new test code until the **test** succeeds **GREEN.**
3. **Refactor** the test and the code once you are green and as the solution evolves.

NOTE

Refer to Guidelines for Test-Driven Development [8] and Testing for Continuous Delivery with Visual Studio 2012 [9] for more detailed information on good unit test development practices.

[8] http://msdn.microsoft.com/en-us/library/aa730844(v=vs.80).aspx

[9] http://aka.ms/treasure25

Unit test checklist – Top 9

Check	Description	✓
Descriptive naming convention	Consider a naming convention such as Classname_Purpose_ExpectedResult style, which makes the tests consistent in style, and descriptive, with a clear intent. Examples: • *Credentials_UserNameLength_Succeed()* • *Credentials_UserNameLength_TooLong_Fail()*	☐
Document your test code	The unit test tests and validates the feature and business requirements. The code itself is a living documentation of the system. We recommend that you use a combination of the name, concise code, and minimal documentation, to make the unit test readable, maintainable, and self-describing.	☐
Descriptive error and assert messages	Use descriptive messages to improve **readability** of code and the build log. Examples: *Content submission failed due to too many spelling errors*	☐
Adopt an interface driven design	The **interface defines the contract**, allowing the use of stubs and black box testing.	☐
Keep it simple	The unit test should be as simple, clean, and concise as possible. It should be perceived as a valuable asset for testing and **documentation**; otherwise, it might become an ignored maintenance mess.	☐
Keep it focused	The unit test is **focused** at the **smallest unit of code (method/class/etc.)** that **can be tested in isolation**, not at system or integration level. To keep the test and associated infrastructure focused on the unit (class) level, avoid testing across application or service layers.	☐
Have one logical assert	Each test should only have **one logical assertion**. It should validate the unit test, as indicated by its descriptive name. A logical assert can contain one or more assert statements.	☐
Organize and maintain your tests	The code should be organized and maintained as a **first class citizen,** on par with the rest of the solution. Its code should be based on the same best practices, quality, and company coding requirements.	☐
Test the good, the bad and the edge case	The unit test should cover all possible scenarios and strive for high **code coverage**. Testing edge cases and exceptions are the responsibility of the tester, not the end user!	☐

Table 3 – Good unit tests checklist

Chapter 2: Introducing Microsoft Fakes

Microsoft Fakes is a new code isolation framework that can help you isolate code for testing by replacing other parts of the application with stubs or shims. It allows you to test parts of your solution even if other parts of your app haven't been implemented or aren't working yet.

Microsoft Fakes come in two flavors:

- **Stubs** … replace a class with a small substitute ("stub") that implements the same interface.
- **Shims** … modify the compiled code at run time, to inject and run a substitute ("shim").

As shown in Figure 4, **stubs** are typically used for calls within your solution that you can decouple using interfaces; **shims** are used for calls to referenced assemblies for which the code is not under your control.

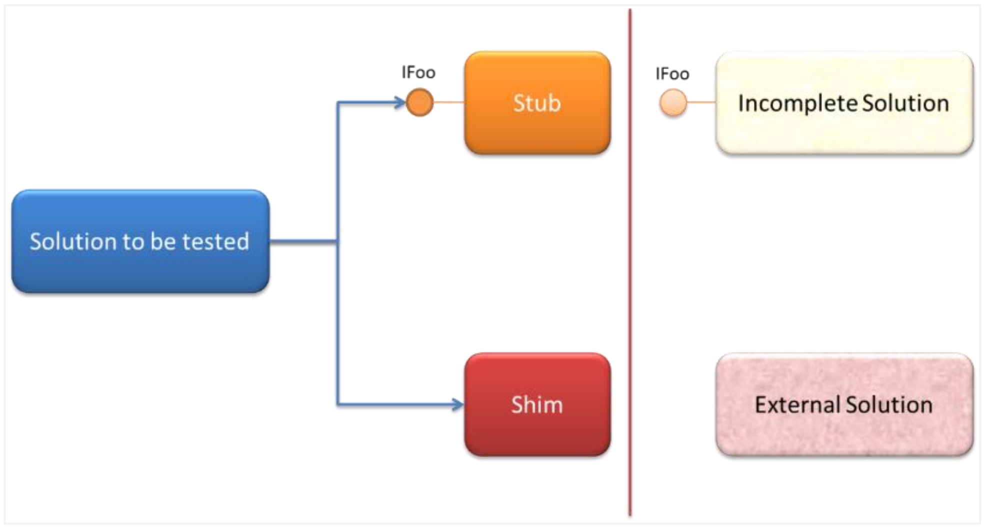


Figure 4 – Stub versus Shim in your solution

Stubs

Let's take an in-depth look at Stubs so you, as a developer, can start integrating them into your software development lifecycle.

What do we test?

In the following diagram (see **Figure 5**), we have a canonical example of a multi-tier solution. The complexity of any solution is usually dependent upon the level of features and capabilities that the solution satisfies, and for this example, we will simplify our scenario. We are focusing on isolating — within reason — the components that comprise the building blocks for our solution.

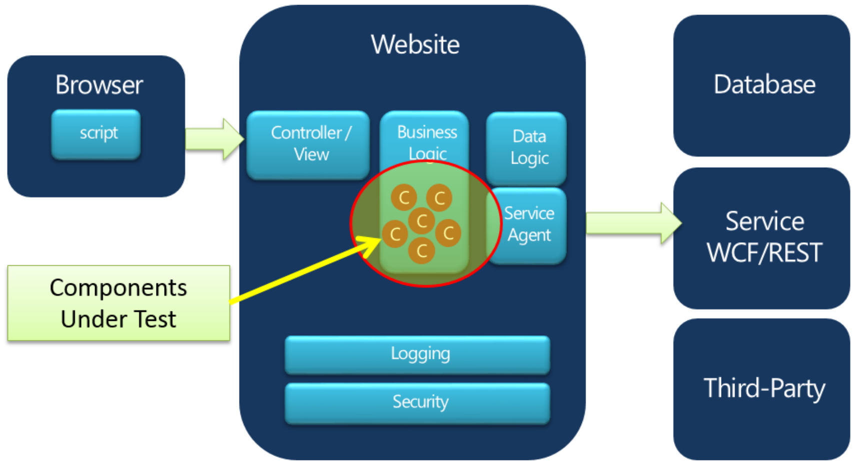


Figure 5 – System under test

For our isolation, we'll look at the individual Components that run in the Business Logic tier (your terminology may vary) for Stubs. Our testing strategy here is NOT testing the logic in the Database, the logic in the WCF/REST services, the Browser, etc. Our focus is reasonable granularity *within* our components (state and behavior).

Test doubles

Test doubles provide ways to isolate code under test from its dependencies. They play an important role in the evolution of your codebase towards quality and greater testability. Stubs, a type of test double, require the code under test to be designed in a specific way that promotes separation and decoupling of dependencies (see **Figure 6**).

It's important to understand what Stubs are versus what they are not. Martin Fowler's article Mocks aren't Stubs[10] compares and contrasts the underlying principles of Stubs and Mocks. As outlined in Martin Fowler's article, a stub provides static canned state which results in state verification[11] of the system under test, whereas a mock provides a behavior verification[12] of the results for the system under test and their indirect outputs as related to any other component dependencies while under test.

For this section, we will focus on Stubs and provide guidance on the value of introducing Stubs into your overall development process. You can check out Martin Fowler's article for more background on the principles and

[10] http://martinfowler.com/articles/mocksArentStubs.html

[11] http://xunitpatterns.com/State%20Verification.html

[12] http://xunitpatterns.com/Behavior%20Verification.html

contrast the two different approaches. With Stubs, we can provide both isolation of the code under test along with specific test cases for validation in our business code – that is, specific "state" of objects or behavior that we can reproduce under test conditions as it executes our code under test. Yet it is important to note that Microsoft Fakes currently does not provide Behavioral Verification, as found in mocking frameworks such as NMoq and Rhino Mocks.

Why use Stubs?

Testing and Development processes that incorporate unit testing along with Code Coverage analysis have the ability to alleviate issues related to lack of coverage in other more "integrated" testing scenarios. Full integration and white-box testing potentially miss significant parts of the code base, thereby increasing the chance of introducing functional defects.

Increasing code coverage

With Stubs we identify specific conditions at the unit level and incorporate this "state" into a single test or set of tests. This increases overall test coverage of the codebase. Defects and conditions are identified during integration tests, production faults, or other means. This represents areas of the codebase not covered by sufficient unit tests. At this point, by adding additional test stubs that cover these conditions or areas of the code, helps affect both the coverage and quality of the coverage for the system under test. Additionally, these stubs become part of the set of unit tests, with the desire that "it won't happen again" – or regression.

During test-driven development (TDD), defects are routinely identified at a point past development, when the initial unit tests are written (system or acceptance testing, or even production). Stubs make it easier to create unit tests by ensuring coverage of specific state conditions from isolated dependencies. In addition, a good TDD approach is to not write the change in the functional code immediately. Instead, write code for a unit test that is able to re-create the state of these isolated components. The unit test that represents the condition for the defect should be introduced first. That test initially fails, and then it is the developer, not the tester, who addresses the defect in the functional code until unit tests are all passing.

Isolation and granularity

As discussed in the *What do we test* section, isolation allows you to focus on the area of the system under test without external dependencies – an isolation of the system under test or the unit. In our diagram above, this was the Business Logic Layer (BLL) components. Isolation reduces the amount of code your test is directed at. In many instances, this can also reduce the amount of code required for test setup as well. Component or Environmental Isolation is required to decouple transient conditions or physical state in order to recreate specific conditions that are not easily reproducible through, for example, a sample database or file that contains test data.

Better components make better systems

The underlying principle of Isolation and Stubs in unit testing is directly tied to concepts related to component or composition approaches to building solutions. In many solutions, we look towards building apps and solutions based upon other parts. As we decompose a solution to its more granular parts, and if we focus on raising the quality of these parts, we can drive the quality of our software upwards. Assuming we all agree that if you build something upon poor quality components you lessen the chance of an overall quality solution; and the corollary is if you build something with quality components you increase the chance of a higher quality solution. Notice that we imply "chance". A successful solution is not necessarily tied to quality components; what we are trying to instill is that we can mitigate risk of failure (which could be poor quality in the eyes of the customer) by applying these techniques.

Shims

Shims are a feature of Microsoft Fakes that allows creating unit tests for code that would otherwise not be testable in isolation. In contrast to Stubs, Shims do not require the code to be tested be designed in a specific way. To be able to use Stubs, they need to be injected[13] into the code under test in some way, so that calls to the dependency will not be handled by a real (production code) component, but instead by the Stub. This way, test values and objects can be passed to the code under test (see **Figure 6**).

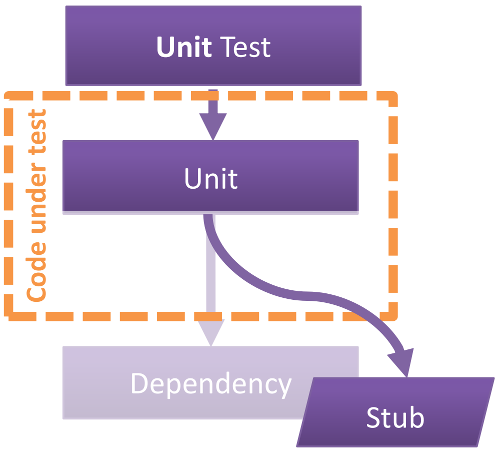


Figure 6 – A Dependency replaced by an injected Stub

Yet there are cases when the code under test is not designed in a way that allows for its dependencies to be replaced, for instance when it makes calls to static methods, has hard coded dependencies or relies on sealed classes of third party frameworks. In such cases, Shims provides a means of not replacing dependencies at design time but instead intercepting the calls to the dependencies at run time and detouring to different code specifically made up to provide the desired test values to the code under test (see **Figure 7**).

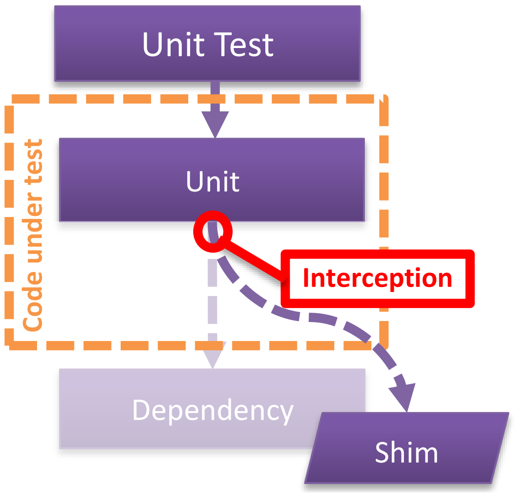


Figure 7 – A method call to a Dependency intercepted and rerouted to a Shim

[13] The technique of "Dependency Injection" (DI) is being used in Object Oriented Programming to decouple classes from their dependencies or at least from the specific concrete implementations of dependent interfaces. This technique can be used to inject stubs for testing purposes as well. DI can either be performed by using a framework for it (e.g. Spring.NET or Unity) or by "manually" injecting concrete implementations to classes,, e.g. by creating an instance of the dependency class and passing it to a constructor of the dependent class ("Constructor Injection"). For Constructor Injection to work, the dependent component obviously needs to have an appropriate constructor. For a class that completely hides its dependencies, DI does not work and stubs cannot be injected as well.

Choosing between a stub and a shim

As outlined in the overview, **Stubs** provide implementations of **interfaces** and classes and are most effective when the code was designed with testability in mind. **Shims** allow support for times when the dependent code, for example legacy or external, cannot be changed and a detour of method calls to Shim implementations is needed (see **Table 4**).

NOTE

Where possible, use **Stubs**. Your unit tests will run faster.

Objective | Consideration	Stub	Shim
Looking for maximum **performance**?	✓	✓ (slower)
Abstract and Virtual methods	✓	
Interfaces	✓	
Internal types	✓	✓
Static methods		✓
Sealed types		✓
Private methods		✓

Table 4 – Choosing between a stub and a shim

Refer to Isolating Code under Test with Microsoft Fakes[14] on MSDN for more information.

[14] http://msdn.microsoft.com/en-us/library/hh549175.aspx

Chapter 3: Migrating to Microsoft Fakes

We expect that many of you reading this guidance might have had some previous experience with Microsoft Moles[15] as well as other commercial and open source isolation frameworks. In this chapter, we'll guide you through some of the steps and hurdles you could face when migrating from some of these to Microsoft Fakes. Keep in mind that you can use Microsoft Fakes with other frameworks so you can migrate at your own pace.

Migrating from Moles to Microsoft Fakes

Moles was the Microsoft Research project foundation for Microsoft Fakes and as such, it has many similarities in syntax. However, there is no automated mechanism to convert a project from using Moles to one using Microsoft Fakes. With the release of Microsoft Fakes, Moles is now legacy and it is recommended that Moles-based test projects be migrated to Microsoft Fakes if possible.

The key reason for this recommendation, beyond that of support, is that installing Visual Studio 2012 installs .NET 4.5 which is a problem for Moles because Moles tries to generate stubs or moles for types that exist only in .NET 4.5. Moles itself is written in .NET 4 so errors result. The only workaround is to use filters in the .moles file to avoid loading those types (and dependent types if needed). This process is detailed on the Microsoft Research site[16]. Because this is a potentially complex process and prone to error, we recommend that you do not attempt to run Moles and Fakes side by side.

Any migration from Moles to Fakes will require changes in code; however, given that Microsoft Fakes' foundation is Moles, the scope of the changes is not too great.

NOTE

Microsoft Fakes does not replace PEX[17] and provides no automatic unit test generation functionality provided by PEX.

Changing references

The first step in any migration from Moles to Fakes is to remove the references to the moled assemblies and to delete the **.moles** files. The next step is to generate new Fake Assemblies. This will add the references to the fake assemblies and add the **.fakes** files to the test project.

Scope of faking shims

A major difference between Moles and Shims is the way that the scope of the faking operation is managed.

In Moles, a **HostType** attribute is placed on the test method:

```
[TestMethod]
[HostType("Moles")]
public void TestMethod()
{
    //...
}
```

In Microsoft Fakes, a **using** statement containing a **ShimsContext** object is used within the test method. This change needs to be made in all moles-based tests that will be using the Shim feature of Microsoft Fakes. Remember that the **using** construct is not required when using Stubs.

[15] http://research.microsoft.com/en-us/projects/moles

[16] http://research.microsoft.com/en-us/projects/moles/molesdev11.aspx

[17] http://research.microsoft.com/en-us/projects/pex

```
[TestMethod]
public void FakesTestMethod()
{
    using (ShimsContext.Create())
    {
        //...
    }
```

Defining behaviors

Within tests, the basic way that behaviors for Stubs or Shims are defined has not changed between Moles and Fakes. However, the way that the moled / faked classes are named has changed. In Moles, the behavior of a moled DateTime call to the Now property would be declared as:

```
MDateTime.NowGet = () =>
{
    return new DateTime(1, 1, 1);
};
```

In Microsoft Fakes, this changes to:

```
System.Fakes.ShimDateTime.NowGet = () =>
{
    return new DateTime(1, 1, 1);
};
```

Because only the namespace changes, this should be a simple global replace within a text editor.

Moles behaviors for Microsoft SharePoint

Microsoft Research produced a behaviors library to aid in the testing of SharePoint[18]. This library leveraged Moles to redirect SharePoint API calls to an in-memory model of SharePoint. When moving to Microsoft Fakes, you can make use of SharePoint Emulators, which are a released version of the Moles SharePoint behaviors. SharePoint Emulators are discussed in the Introducing SharePoint Emulators[19] blog and available via a NuGet package.

[18] http://research.microsoft.com/en-us/projects/pex/pexsharepointbehaviors.pdf

[19] http://blogs.msdn.com/b/visualstudioalm/archive/2012/11/26/introducing-sharepoint-emulators.aspx

Migrating from commercial and open source frameworks

Introduction

A number of open source projects, such as RhinoMocks, MOQ, etc., provide equivalents to the stubbing technologies found within Microsoft Fakes. They do not, however, offer an equivalent to the shimming technologies provided by Microsoft Fakes. Only commercially licensed products provide the ability to mock out objects such as sealed private classes.

Other than Microsoft Fakes, this form of mocking is offered by Telerik in their JustMock[20] product and Typemock in their Isolator[21] product. In both cases, their product ranges offer Microsoft Fakes stub-like functionality, allowing the mocking of interfaces, etc., in the form of a free 'lite' version. They also offer premium products that provide shim-like tools to mock out objects that are not usually mockable.

Differences between these products and Microsoft Fakes

Creating fake assemblies

The most noticeable difference between these products and Microsoft Fakes is the process that the developer needs to go through to generate the shim.

In Microsoft Fakes, the developer must right-click the assembly reference they wish to mock and select **Add Fake Assembly**. This will generate a new assembly that must be referenced to create the shim objects.

In the Telerik and Typemock products, this pre-generation of the fake assembly is not required; it is handled by the framework at runtime.

Using Microsoft Fakes

Within unit testing code both Telerik and Typemock use lambda expressions to define behavior, as does Microsoft Fakes. The format changes slightly between products, though the intention expressed remains the same.

Telerik JustMock

The following example (see **Code 1**) shows the mocking of a call to DateTime.Now using Telerik JustMock:

Code 1 – Example of Telerik JustMock usage[22].

```
[TestClass]
public class MsCorlibFixture
{
    static MsCorlibFixture()
    {
        Mock.Replace(() => DateTime.Now).In<MsCorlibFixture>
                                 (x => x.ShouldAssertCustomValueForDateTime());
    }

    [TestMethod]
    public void DateTimeTest()
    {
        Mock.Arrange(() => DateTime.Now).Returns(new DateTime(2016, 2, 29));

        // Act
        int result = MyCode.DoSomethingSpecialOnALeapYear();
```

[20] http://www.telerik.com/products/mocking.aspx

[21] http://www.typemock.com/isolator-product-page

[22] http://www.telerik.com/help/justmock/advanced-usage-mscorlib-mocking.html

```
        // Assert
        Assert.AreEqual(100, result);
    }
}
```

Typemock isolator

The following example shows the mocking of a call to DateTime.Now using Typemock Isolator:

```
[TestMethod, Isolated]
public void DateTimeTest()
{
    // Arrange
    Isolate.WhenCalled(() => DateTime.Now).WillReturn(new DateTime(2016, 2, 29));

    // Act
    int result = MyCode.DoSomethingSpecialOnALeapYear();

    // Assert
    Assert.AreEqual(100, result);
}
```

Microsoft Fakes

The following example[23] shows the mocking of a call to DateTime.Now using Microsoft Fakes:

```
[TestMethod]
public void DateTimeTes()
{
    using (ShimsContext.Create())
    {
        // Arrange:
        System.Fakes.ShimDateTime.NowGet = () => { return new DateTime(2016, 2, 29); };

        // Act
        int result = MyCode.DoSomethingSpecialOnALeapYear();

        // Assert
        Assert.AreEqual(100, result);
    }
}
```

Migrating from Moq

This section describes how to migrate some of the commonly used features of Moq to Stubs in Microsoft Fakes.

The big difference between Moq and Stubs is that Moq uses generics based on the interfaces and abstract classes that need to be stubbed and Stubs uses code generation to implement classes derived from the interfaces and abstract classes.

The stub classes generated by Microsoft Fakes provide extra members that are called by the properties and methods being stubbed to provide the return values or execute any arbitrary code required.

One of the small differences you will see in the unit test's code itself is that when accessing the actual stubbed object you don't need to use the Object member on the Mock<T> object, because the Microsoft Fakes stub is actually derived directly from the interface or abstract class to be stubbed.

The rest of this section starts by presenting a little bit of example code, and then goes on to provide Moq scenarios based on this example code and then shows how to change it to work with Stubs.

[23] http://msdn.microsoft.com/en-us/library/hh549176.aspx

Example code

In the rest of this section, we'll need to reference some sample code. We'll use a very simple bank account class as our running example (see **Code 2**):

Code 2 – Sample interface dependencies

```
public enum TransactionType { Credit, Debit };

public interface ITransaction
{
    decimal Amount { get; }
    TransactionType TransactionType { get; }
}

public interface ITransactionManager
{
    int GetAccountTransactionCount(DateTime date);
    ITransaction GetTransaction(DateTime date, int transactionNumber);
    void PostTransaction(decimal amount, TransactionType transactionType);
    event TransactionEventHandler TransactionEvent;
}

public delegate void TransactionEventHandler(DateTime date, decimal amount, TransactionType transaction
Type);
```

Migrating setup with returns

In Moq, the Setup method is used to create a stub that will respond to a specific set of parameters with a specific return value. For example, if we want the ITransactionManager object passed to the code under test to return a particular value when passed a date to the GetAccountTransactionCount method, then we might use code like this in Moq:

```
DateTime testDate = new DateTime(2012, 1, 1);
Mock<ITransactionManager> mockTM = new Mock<ITransactionManager>();
mockTM.Setup(tm => tm.GetAccountTransactionCount(testDate)).Returns(8);
```

This can easily be done with Stubs as follows:

```
DateTime testDate = new DateTime(2012, 1, 1);
StubITransactionManager stubTM = new StubITransactionManager();
stubTM.GetAccountTransactionCountDateTime = (date) => (date == testDate) ? 8 : default(int);
```

The Stub class created by Fakes provides a member called GetAccountTransactionCountDateTime, which you can set to a lambda that will execute to provide the return value you want. Note that the lambda checks for the value of the parameter because in Moq it would do the same. If a different value was provided, it would return the default value for the type.

Moq also allows you to call the Setup method multiple times to return different values for different inputs. Here's an example:

```
// txn1 and txn2 previously set up as mock transactions
mockTM.Setup(tm => tm.GetTransaction(testDate, 0)).Returns(txn1.Object);
mockTM.Setup(tm => tm.GetTransaction(testDate, 1)).Returns(txn2.Object);
```

This can be dealt with using a more complex lambda, such as the following:

```
// txn1 and txn2 previously set up as stub transactions
ITransaction[] stubTransactions = new ITransaction[] { txn1, txn2 };
stubTM.GetTransactionDateTimeInt32 = (date, index) => (index >= 0 || index < stubTransactions.Length) ?
 stubTransactions[index] : null;
```

Here we use an array to allow the values to be matched and then looked up by the lambda. In this case, we are actually ignoring the date parameter.

Sometimes, the values to be looked up can't be known in advance and using a collection allows further test code to set up the dictionary dynamically, even after the stub has been passed to the code under test. One scenario for this is where you want to centralize the creation of the code under test into the test initialization code, but have each test run with a different set of values. You could do this as shown in **Code 3** in Stubs:

Code 3 – Setting up code under test and dependencies in initialization

```
private StubITransactionManager stubTM = new StubITransactionManager();
private List<ITransaction> transactions = new List<ITransaction>();
private DateTime testDate = new DateTime(2012, 1, 1);
private Account cut;

[TestInitialize]
public void InitializeTest()
{
    this.stubTM.GetTransactionDateTimeInt32 = (date, index) =>
                                               (date == testDate && index >= 0 || index < this.transac
tions.Count)
                                                   ? this.transactions[index] : null;
    this.stubTM.GetAccountTransactionCountDateTime = (date) =>
                                                       (date == testDate) ? this.transactions.Count :
default(int);
    this.cut = new Account(stubTM);
}

[TestMethod]
public void StubCreditsSumToPositiveBalance()
{
    // Arrange
    this.AddTransaction(10m, TransactionType.Credit);
    this.AddTransaction(20m, TransactionType.Credit);

    // Act
    decimal result = this.cut.CalculateBalance(this.testDate);

    // Assert
    Assert.AreEqual<decimal>(30m, result);
}

[TestMethod]
public void StubDebitsAndCreditsSum()
{
    // Arrange
    this.AddTransaction(10m, TransactionType.Credit);
    this.AddTransaction(20m, TransactionType.Debit);

    // Act
    decimal result = this.cut.CalculateBalance(this.testDate);

    // Assert
    Assert.AreEqual<decimal>(-10m, result);
}

private void AddTransaction(decimal amount, TransactionType transactionType)
{
    this.transactions.Add(new StubITransaction
                                {
                                    AmountGet = () => amount,
                                    TransactionTypeGet = () => transactionType
                                });
}
```

Of course, Moq is also able to deal with setting up methods that take multiple parameters. You deal with this in Stubs using a more complex lambda that checks each of the parameters.

When dealing with multiple input parameters in multiple setups, a simple way to deal with it generically is to use a dictionary that uses a Tuple as the key. An example (see **Code 4**) is shown here:

Code 4 – Stub for multiple parameter values with multiple call setups

```
private StubITransactionManager stubTM = new StubITransactionManager();
private Dictionary<Tuple<DateTime, int>, ITransaction> transactions = new Dictionary<Tuple<DateTime, in
t>, ITransaction>();
private DateTime testDate = new DateTime(2012, 1, 1);
private Account cut;

[TestInitialize]
public void InitializeTest()
{
    this.stubTM.GetTransactionDateTimeInt32 =
        (date, index) =>
        {
            ITransaction txn;
            if (!this.transactions.TryGetValue(new Tuple<DateTime, int>(date, index),
                                                out txn))
            {
                txn = null;
            }

            return txn;
        };
    stubTM.GetAccountTransactionCountDateTime = (date) =>
    this.cut = new Account(stubTM);
}

[TestMethod]
public void StubCreditsSumToPositiveBalance()
{
    // Arrange
    this.AddTransaction(testDate, 0, 10m, TransactionType.Credit);
    this.AddTransaction(testDate, 1, 20m, TransactionType.Credit);

    // Act
    decimal result = this.cut.CalculateBalance(this.testDate);

    // Assert
    Assert.AreEqual<decimal>(30m, result);
}

[TestMethod]
public void StubDebitsAndCreditsSum()
{
    // Arrange
    this.AddTransaction(testDate, 0, 10m, TransactionType.Credit);
    this.AddTransaction(testDate, 1, 20m, TransactionType.Debit);

    // Act
    decimal result = this.cut.CalculateBalance(this.testDate);

    // Assert
    Assert.AreEqual<decimal>(-10m, result);
}

private void AddTransaction(DateTime date, int index, decimal amount, TransactionType transactionType)
{
    ITransaction txn = new StubITransaction
    {
        AmountGet = () => amount,
        TransactionTypeGet = () => transactionType
    };
    this.transactions.Add(new Tuple<DateTime, int>(date, index), txn);
}
```

Moq also provides various ways of matching multiple input values all in one setup call. For example:

```
mockTM.Setup(tm => tm.GetTransaction(It.IsAny<DateTime>(), It.IsAny<int>())).Returns(txn.Object);
```

In this case, a Stub can just ignore the appropriate parameter in the lambda. In the previous case, it would look like this:

```
stubTM.GetTransactionDateTimeInt32 = (date, index) => txn1;
```

Migrating Callbacks

Callbacks allow a method to be registered that will be called whenever some other action takes place. Both Moq and Stubs allow you to specify a callback method within their unit tests.

If for example, we wished to call back to the following method within our test class, it would look like this:

```
bool callBackCalled = false;

public void CallBackMethod(decimal param)
{
    callBackCalled = true;
}
```

In Moq, the .Setup method is used, as in the previous section. However, in place of the .Returns method the .Callback method is used specify the call back method, passing parameters as required in a manner similar to parameter handling for the Returns method:

```
[TestMethod]
public void MoqCallback()
{
    // arrange
    Mock<ITransactionManager> mockTM = new Mock<ITransactionManager>();
    mockTM.Setup(tm => tm.PostTransaction(It.IsAny<decimal>(),
                                          It.IsAny<TransactionType>()))
                                          .Callback<decimal, TransactionType>
                                          ((amount, transType) => CallBackMethod(amount));
    Account cut = new Account(mockTM.Object);

    // act
    cut.AddCredit(9.99m);

    // assert
    Assert.AreEqual(true, callBackCalled);
}
```

With Stubs, the call back is declared as a delegate:

```
[TestMethod]
public void StubCallback()
{
    // arrange
    StubITransactionManager stubTM = new StubITransactionManager();
    stubTM.PostTransactionDecimalTransactionType = (amount, transType) =>
                                                     CallBackMethod(amount);
    Account cut = new Account(stubTM);

    // act
    cut.AddCredit(9.99m);

    // assert
    Assert.AreEqual(true, callBackCalled);
}
```

Migrating verify

Verify is used in Moq for behavioral verification, to make sure that the code under test has made certain calls with particular parameters or that it has made certain calls a certain number of times. However, its behavioral verification abilities are limited. For example, it cannot straightforwardly verify that methods are called in a particular order. Stubs in Microsoft Fakes are not intended to be used in this way; however, they can do behavioral verification if required. In the following example, we want to test that a credit transaction is posted for the opening balance when an account is opened. This could be tested in other ways, although this example shows how to test behavior. In Moq, the test would look like this:

```
[TestMethod]
public void MoqAccountOpenPostsInitialBalanceCreditTransaction()
{
    // Arrange
    Mock<ITransactionManager> mockTM = new Mock<ITransactionManager>();

    Account cut = new Account(mockTM.Object);

    // Act
    cut.Open(10m);

    // Assert
    mockTM.Verify(tm => tm.PostTransaction(10m, TransactionType.Credit), Times.Once());
}
```

Using Stubs in Microsoft Fakes requires a little bit of code in the lambda to record the calls (see **Code 5**):

Code 5 – Stubs Behavioral Verification

```
[TestMethod]
public void StubAccountOpenPostsInitialBalanceCreditTransaction()
{
    // Arrange
    int callCount = 0;
    StubITransactionManager stubTM = new StubITransactionManager
    {
        PostTransactionDecimalTransactionType = (amount, type) =>
        {
            if (amount == 10m && type == TransactionType.Credit)
            {
                callCount++;
            }
        }
    };

    Account cut = new Account(stubTM);

    // Act
    cut.Open(10m);

    // Assert
    Assert.AreEqual<int>(1, callCount);
}
```

With Moq, the developer has to call the correct form of verification method, depending on the item they wish to verify:

- .Verify – for methods
- .VerifyGet – for Get calls to properties
- .VerifySet – for Set calls to properties

Because Stubs does not provide verification methods, the developer has to create their own, then there is no similar distinction between verification of methods and properties; it is all custom code created by the developer.

Clearly, more complex verifications are sometimes needed. For example, different parameter combinations might be necessary. These can be verified by a Tuple and Dictionary technique, similar to the one shown in the section on Setup, to count the number of calls for each parameter combination.

Migrating events

In event driven architectures, it is important to be able to raise events within tests. This can be done using both Moq and Stubs using very similar syntax.

In both cases, the developer first registers a delegate that will be called when the event is raised. In this example, we only set a Boolean flag to signal that the event using our custom event type was raised. However, the verification techniques, as discussed in the previous section, could be used to check any of the passed parameters.

In the Act section, we raise the event we wish to test by passing in the required parameters. Here the syntax does differ. With Stubs the event is raised with a simple method call (see **Code 6**), while in Moq we have to use a lambda expression (see **Code 7**) to provide the parameter values:

Code 6 – Raising events in Stubs

```
[TestMethod]
public void StubsraiseEvent()
{
    // arrange
    bool delegateCalled = false;
    DateTime testDate = new DateTime(2012, 1, 1);
    StubITransactionManager stubTM = new StubITransactionManager();
    stubTM.TransactionEventEvent = (date, amount, transType) => { delegateCalled = true; };

    // act
    // Raise passing the custom arguments expected by the event delegate
    stubTM.TransactionEventEvent(testDate, 9.99m, TransactionType.Credit);

    // assert
    Assert.AreEqual(true, delegateCalled);
}
```

Code 7 – Raising events in Moq

```
[TestMethod]
public void MoqRaiseEvent()
{
    // arrange
    bool delegateCalled = false;
    DateTime testDate = new DateTime(2012, 1, 1);
    Mock<ITransactionManager> mockTM = new Mock<ITransactionManager>();
    mockTM.Object.TransactionEvent += delegate { delegateCalled = true; };

    // act
    // Raise passing the custom arguments expected by the event delegate
    mockTM.Raise(tm => tm.TransactionEvent += null,
                                                testDate, 9.99m, TransactionType.Credit);

    // assert
    Assert.AreEqual(true, delegateCalled);
}
```

Recursive Fakes

When you have a complex object tree that needs to be faked, setting all properties will be time consuming and often unnecessary. In many cases all that is required is that the faked object should not throw any null reference exceptions.

Moq provides a means to set the value for all references/properties in the object tree. This is to use the DefaultValue.Mock setting for mocked objects and the SetupAllProperties method to set all properties. By doing this, tests should not encounter null reference exceptions. The default value for any object should be returned. For example, integers return 0, and string returns String.Empty. If any other specific values are required, the developer should set them explicitly:

```
Mock<ITransaction> mockTr = new Mock<ITransaction>() { DefaultValue = DefaultValue.Mock };
mockTr.SetupAllProperties();
```

With Stubs, a similar syntax is used when a stub is created. The InstanceBehavior can be set to set the required behavior when any sub items or properties are accessed:

```
StubITransaction stubTr = new StubITransaction();
stubTr.InstanceBehavior = StubBehaviors.DefaultValue;
```

Additional sample

For illustration purposes, let's look at another sample to finish migrating from Moq to Microsoft Fakes. In the following sample (see **Code 8**), based on the blog post *Mocking HttpWebRequest using Microsoft Fakes*[24], we have a WebServiceClient object using HttpWebRequest, which we would like to get under test:

Code 8 – WebServiceClient object using HttpWebRequest

```
public class WebServiceClient
{
    /// <summary>
    /// Calls a web service with the given URL
    /// </summary>
    /// <param name="url">The web service's URL</param>
    /// <returns>True if the services responds with an OK status code (200). False Otherwise</returns>
    public bool CallWebService(string url)
    {
        var request = CreateWebRequest(url);
        var isValid = true;
        try
        {
            var response = request.GetResponse() as HttpWebResponse;
            isValid = HttpStatusCode.OK == response.StatusCode;
        }
        catch (Exception ex)
        {
            isValid = false;
        }

        return isValid;
    }

    /// <summary>
    /// Creates an HttpWebRequest object
    /// </summary>
    /// <param name="url">The URL to be called.</param>
    /// <returns>An HttpWebRequest.</returns>
    private static HttpWebRequest CreateWebRequest(string url)
    {
        var request = WebRequest.Create(url) as HttpWebRequest;
        request.ContentType = "text/xml;charset=\"utf-8\"";
        request.Method = "GET";
        request.Timeout = 1000;
        request.Credentials = CredentialCache.DefaultNetworkCredentials;
        return request;
    }
}
```

To use Moq we need to create a CustomWebRequestCreate object that implements the IWebRequestCreate interface (see **Code 9**). This allows us to mock the HttpWebResponse using RegisterPrefix:

[24] http://hamidshahid.blogspot.co.uk/2013/01/mocking-httpwebrequest-using-microsoft.html

Code 9 – CustomWebRequestCreate object

```
/// <summary>
/// A custom implementation of IWebRequestCreate for Web Requests.
/// </summary>
/// <summary>A web request creator for unit testing</summary>
public class CustomWebRequestCreate : IWebRequestCreate
{
    /// <summary>
    /// The web request.
    /// </summary>
    private static WebRequest nextRequest;

    /// <summary>
    /// Internally held lock object for multi-threading support.
    /// </summary>
    private static object lockObject = new object();

    /// <summary>
    /// Gets or sets the next request object.
    /// </summary>
    public static WebRequest NextRequest
    {
        get
        {
            return nextRequest;
        }

        set
        {
            lock (lockObject)
            {
                nextRequest = value;
            }
        }
    }

    /// <summary>
    /// Creates a Mock Http Web request
    /// </summary>
    /// <param name="httpStatusCode"></param>
    /// <returns>The mocked HttpRequest object</returns>
    public static HttpWebRequest CreateMockHttpWebRequestWithGivenResponseCode(HttpStatusCode httpStatu
sCode)
    {
        var response = new Mock<HttpWebResponse>(MockBehavior.Loose);
        response.Setup(c => c.StatusCode).Returns(httpStatusCode);

        var request = new Mock<HttpWebRequest>();
        request.Setup(s => s.GetResponse()).Returns(response.Object);
        NextRequest = request.Object;
        return request.Object;
    }

    /// <summary>
    /// Creates the new instance of the CustomWebRequest.
    /// </summary>
    /// <param name="uri">The given Uri</param>
    /// <returns>An instantiated web request object requesting from the given Uri.</returns>
    public WebRequest Create(Uri uri)
    {
        return nextRequest;
    }
}
```

Our Moq based unit test would look like this:

```
[TestMethod]
public void TestThatServiceReturnsAForbiddenStatuscode()
{
        // Arrange
        var url = "http://testService";
        var expectedResult = false;
        WebRequest.RegisterPrefix(url, new CustomWebRequestCreate());
        CustomWebRequestCreate.CreateMockHttpWebRequestWithGivenResponseCode(HttpStatusCode.Forbidden);

        var client = new WebServiceClient();

        //Act
        bool actualresult = client.CallWebService(url);

        //Assert
        Assert.AreEqual(expectedResult, actualresult);
}
```

Using Microsoft Fakes, we can directly shim the HttpWebRequest object and not rely on creating our own implementation of IWebRequestCreate. Our Microsoft Fakes-based unit test would look like this (see **Code 10**):

Code 10 – Microsoft Fakes-based unit test

```
[TestMethod]
public void TestThatServiceReturnsAForbiddenStatuscode()
{
        using (ShimsContext.Create())
        {
          // Arrange
          var requestShim = new ShimHttpWebRequest();
          ShimWebRequest.CreateString = (uri) => requestShim.Instance;
          requestShim.GetResponse = () => { return new ShimHttpWebResponse() { StatusCodeGet = () => { r
          eturn HttpStatusCode.Forbidden; } }; };
          var client = new WebServiceClient();
          var url = "testService";
          var expectedResult = false;

          // Act
          bool actualresult = client.CallWebService(url);

          // Assert
          Assert.AreEqual(expectedResult, actualresult);
        }
}
```

References

For more Moq samples see http://code.google.com/p/moq/wiki/QuickStart

Migrating from RhinoMocks

This section describes how to migrate some of the commonly used features of RhinoMocks API's to Microsoft Fakes. RhinoMocks[25] is one of many open source software projects that provide stubbing technologies for code under test. RhinoMocks uses an API to stub interfaces and abstract classes via reflection. Microsoft Fakes uses code generation for interfaces and abstract classes. The RhinoMocks syntax used for this document is based on version 3.5[26] or later. For developers to use Microsoft Fakes, they just need to right-click the assembly they wish to create stubs for and select Add Fake Assembly from the menu.

[25] http://ayende.com/blog

[26] http://ayende.com/Wiki/Rhino+Mocks+3.5.ashx

We will only cover migration examples for the most commonly used RhinoMocks APIs. The RhinoMocks API set is large and not within the scope of this document. For details of features not covered here, refer to either the RhinoMocks documents or Microsoft Fakes documents for help in migrating to Microsoft Fakes.

Sample code under test

Most of the migration samples use the following interface code and class as being under test (see **Code 11**):

Code 11 – Sample code under test with interface dependencies

```
public interface IDetermineTempWithWindChill
{
    double WhatisCurrentTemp(double airTemp, double airSpeed);
    double CalcSpecialCurrentTemp(double airTemp, double airSpeed, double aboveSeaLevel);
}

public interface IClassUnderTest
{
    double WhatIsTheTempToday(double currentAirTemp, double currentWindSpeed,
                              double currentFeetAboveSeaLevel);
}

public class ClassUnderTest : IClassUnderTest
{
    private readonly IDetermineTempWithWindChill _determineTempWithWindChill;

    public ClassUnderTest(IDetermineTempWithWindChill determineTempWithWindChill)
    {
        _determineTempWithWindChill = determineTempWithWindChill;
    }

    public double WhatIsTheTempToday(double currentAirTemp, double currentWindSpeed,
                                    double currentFeetAboveSeaLevel)
    {
        return currentFeetAboveSeaLevel >= 5000.0
                                ? _determineTempWithWindChill.WhatisCurrentTemp
                                    (currentAirTemp, currentWindSpeed) * 0.1
                                : _determineTempWithWindChill.WhatisCurrentTemp
                                    (currentAirTemp, currentWindSpeed);
    }
}
```

Migrating stub setups and returns

RhinoMocks uses two types of mocking strict mocking and dynamic mocking. Strict mocking requires the developer to define returns for all stub methods being called for the code under test:

```
IDetermineTempWithWindChill determineTempWithWindChill =
MockRepository.GenerateStrictMock<IDetermineTempWithWindChill>();
```

If a stubbed method does not define a return, an exception will be thrown for that method if called. By default, Microsoft Fakes uses default value mocking. The developer can override this behavior by defining InstanceBehavior in the stub interface or abstract class. If a Stub is not set up with a return and is called, it will throw an exception when the unit test executes:

```
stubIDetermineTempWithWindChill.InstanceBehavior = StubBehaviors.NotImplemented;
```

In using

stubIDetermineTempWithWindChill.`InstanceBehavior = StubBehaviors.NotImplemented;`

, if the developer does not define a return for the `WhatIsCurrentTemp` method an exception would be thrown during unit testing. For most of the samples, we'll use dynamic mocking.

RhinoMocks stub setup and returns are straightforward calls to the API, as shown in the next code sample:

```
[TestMethod]
public void TestSetupAndReturn()
{
    //Arrange
    double airTemp = 35;
    double airSpeed = 5.0;
    IDetermineTempWithWindChill determineTempWithWindChill =
        MockRepository.GenerateStub<IDetermineTempWithWindChill>();
    determineTempWithWindChill.Stub(x => x.WhatisCurrentTemp(airTemp, airSpeed))
                                .Return(airTemp * airSpeed);
    IClassUnderTest classUnderTest = new ClassUnderTest(determineTempWithWindChill);

    //Act
    var results = classUnderTest.WhatIsTheTempToday(airTemp, airSpeed, 2000.0);

    //Assert

    Assert.AreEqual(airTemp * airSpeed, results);
}
```

The Microsoft Fakes stub setup and return is a little more involved with lambda expressions, as shown here:

```
[TestMethod]
public void TestSetupAndReturn()
{
    //Arrange
    double airTemp = 35;
    double airSpeed = 5.0;
    StubIDetermineTempWithWindChill stubIDetermineTempWithWindChill =
                    new StubIDetermineTempWithWindChill();
    stubIDetermineTempWithWindChill.WhatisCurrentTempDoubleDouble = (x, y) => x * y;

    IClassUnderTest classUnderTest = new ClassUnderTest(stubIDetermineTempWithWindChill);

    //Act
    var results = classUnderTest.WhatIsTheTempToday(airTemp, airSpeed, 2000.0);

    //Assert
    Assert.AreEqual(airTemp * airSpeed, results);
}
```

Migrating Expect and AssertWasCalled Statements

RhinoMocks uses Expect, AssertWasCalled or AssertWasNotCalled statements to verify algorithms and method calls. The RhinoMocks API provides many options for managing the level of details of how many times a method was called. During testing, if any of the verify statements do not match the expected calls, an exception is thrown:

```
determineTempWithWindChill.Expect(x => x.WhatisCurrentTemp(airTemp, airSpeed)).Repeat.Times(2);
determineTempWithWindChill.AssertWasCalled(x => x.WhatisCurrentTemp(airTemp, airSpeed),options
=> options.Repeat.Times(1));
determineTempWithWindChill.AssertWasNotCalled(x => x.CalcSpecialCurrentTemp(airTemp, airSpeed,
aboveSeaLevel));
```

With Microsoft Fakes, lambda expressions are used to perform similar call verifications. The following code sample (see **Code 12**) shows simple verification for WhatIsCurrentTemp was called once and CalcSpecialCurrentTemp was called none. The CalcSpecialCurrentTemp shows an example where an exception can be thrown if the call should have never been made. With a little more lambda coding, Microsoft Fakes can provide the same level of options for verify algorithms and methods called as in RhinoMocks.

Code 12 – Microsoft Fakes verify method call samples

```
[TestMethod]
public void TestSetupAndReturn()
{
    //Arrange
    int WhatIsCurrentTempCalled = 0;
    int CalcSpecialCurrentTempCalled = 0;

    double airTemp = 35;
    double airSpeed = 5.0;

    StubIDetermineTempWithWindChill stubIDetermineTempWithWindChill =
        new StubIDetermineTempWithWindChill();

    stubIDetermineTempWithWindChill.WhatisCurrentTempDoubleDouble = (x, y) =>
    {
        WhatIsCurrentTempCalled++;
        return x * y;
    };
    stubIDetermineTempWithWindChill.CalcSpecialCurrentTempDoubleDoubleDouble = (x, y, z) =>
    {
        CalcSpecialCurrentTempCalled++;
        throw new Exception("CalcSpecialCurrentTemp should not have been " +
        "called");
    };

    IClassUnderTest classUnderTest = new ClassUnderTest(stubIDetermineTempWithWindChill);

    //Act
    var results = classUnderTest.WhatIsTheTempToday(airTemp, airSpeed, 2000.0);

    //Assert
    Assert.AreEqual(airTemp * airSpeed, results);
    Assert.AreEqual(1, WhatIsCurrentTempCalled);
    Assert.AreEqual(0, CalcSpecialCurrentTempCalled);
}
```

Conclusion

Any migration of existing tests from the Telerik, RhinoMocks, Moq, or Typemock frameworks will require a lot of reworking. Find and replace-based migration doesn't work well because behavior definitions are too different. The exact details of each of these products are beyond the scope of this document. To see how to migrate these products to Microsoft Fakes, refer to the documentation for each product.

Chapter 4: Miscellaneous Topics

In this chapter, we will cover miscellaneous topics and frequently asked questions, some of which we consider advanced, — perhaps edge cases — although we think they are important enough to cover in this guidance.

Targeting Microsoft .NET Framework 4

Just because Microsoft Fakes is a new feature doesn't mean it's restricted to .NET 4.5. You can use Microsoft Fakes with any .NET version that is supported by Visual Studio 2012. For example, you can use Shim Types to cover unit tests for legacy code written in .NET 2.0.

Adopting Microsoft Fakes in a team

To execute a unit test or build a project that uses Microsoft Fakes requires a supported edition[27] of Visual Studio. This applies to other developers executing your tests as well as any Team Foundation Build agents. This is because when you use Microsoft Fakes a reference is made to Microsoft.QualityTools.Testing.Fakes.dll. This file is not included in Visual Studio versions that do not support Microsoft Fakes.

If you add the Microsoft.QualityTools.Testing.Fakes.dll assembly locally to your project and check it in, then others will be able to compile the project. However, a `NotSupportedException` will be thrown if they are run from an edition of Visual Studio that does not support Microsoft Fakes. To avoid encountering exceptions you should allocate your tests to a Test Category[28] that can be filtered out by other developers and build servers not running a supported edition of Visual Studio. For example:

```
[TestCategory("FakesRequired"), TestMethod()]
public Void DebitTest()
{
}
```

If you choose not to add the Microsoft.QualityTools.Testing.Fakes.dll assembly locally, you could isolate Fakes usage by adding them to a separate project and only compiling that project within a specific build configuration.

It is important to note that even if your build servers are running a supported edition of Visual Studio, you must also have Team Foundation Server 2012 or later so that unit tests execute seamlessly during a Team Build that uses Microsoft Fakes. If you are using Team Foundation Server 2010, you will need to edit your build template to run tests that contain Fakes with vstest.console.exe. If you are using the RTM version of Visual Studio 2012, you will need to generate and publish your TRX file. Visual Studio 2012 Update 1 provides an update to vstest.console.exe to support publishing as part of the call.

You can't Fake everything!

By default, the majority of System classes are not faked and some cannot be faked due to a design decision. System is handled as a special case because it could be used by the detours engine, which could lead to unpredictable behavior. For example, the following namespaces are not supported in projects targeting .NET 4:

- System.Globalization
- System.IO
- System.Security.Principal
- System.Threading

[27] http://www.microsoft.com/visualstudio/eng/products/compare

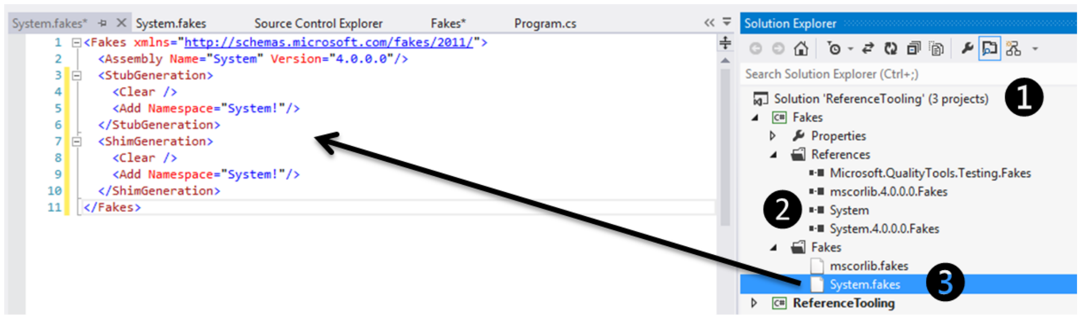
[28] http://msdn.microsoft.com/en-us/library/dd286683.aspx

There is no definitive list of types that are not supported because it depends on the combination of the target framework version of the unit test project and the .NET versions. The list of unsupported types will be different for someone building a .NET 3.0 project with .NET 3.5 Framework installed than for someone building a .NET 3.0 project with the 3.0 version of the Framework.

WARNING

Be cautious if you fake a call that the detours engine uses. It might cause unpredictable behavior.

You can override the default behavior for some of the System classes just like any other assemblies by configuring the generation of stub types and filters in an XML file that has the .fakes extension (see **Figure 8**).



```
<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">
  <Assembly Name="System" Version="4.0.0.0"/>
  <StubGeneration>
    <Clear />
    <Add Namespace="System!"/>
  </StubGeneration>
  <ShimGeneration>
    <Clear />
    <Add Namespace="System!"/>
  </ShimGeneration>
</Fakes>
```

Figure 8 – Faked System classes

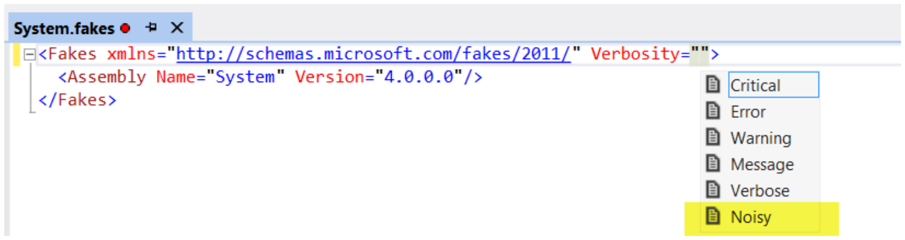
NOTE

To eliminate types that Microsoft Fakes doesn't currently support, such as CancellationToken and CancellationTokenSource, you might have to refactor your code to change interfaces and dependencies of the components you want to test. Warning messages reported while building a **.fakes** assembly will highlight unsupported types.

Refer to Code generation, compilation, and naming conventions in Microsoft Fakes[29] for more information.

Verbose logging

For a number of reasons, Fakes can decide to skip a class when generating shims. With Visual Studio 2012 Update 1, you can get more information on why the generation was skipped by changing the Diagnostic attribute of the Fakes element to true; this will make Fakes report explanations of skipped types as warnings. When Fakes decides to skip individual members of a given type, it writes diagnostic messages to the MSBuild log. You can enable them by setting the Verbosity attribute of the Fakes element to Noisy and increasing MSBuild log verbosity to Detailed (see **Figure 9**).



```
<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/" Verbosity="">
  <Assembly Name="System" Version="4.0.0.0"/>
</Fakes>
```

Figure 9 – Fakes verbosity

[29] http://msdn.microsoft.com/en-us/library/hh708916.aspx

NOTE

While we are on the subject of logging, you may find it necessary in certain scenarios to enable Unit Test Explorer logging. See the following blog post for details - How to enable UTE logs[30].

Working with strong named assemblies

When generating Fakes for strong named[31] assemblies, the strong naming of the Fakes assemblies is automatically handled for you by the framework. By default, the framework will use the same key that was used for the shimmed assembly. You can specify a different public key for the Fakes assembly, such as a key you have created for the shimmed assembly, by specifying the full path to the .snk file that contains the alternate key as the KeyFile attribute value in the Fakes\Compilation element of the .fakes file:

```
<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">
  <Assembly Name="ClassLibrary1" Version="1.0.0.0"/>
  <Compilation KeyFile="MyKeyFile.snk" />
</Fakes>
```

If you have access to the code of the assembly you are faking, you can expose the internal types by using the InternalsVisibleTo[32] attribute. When you do this for strong named assemblies you will need to specify the assembly name *and* the public key for **both** the Fake assembly and the test assembly. For example:

```
[assembly: System.Runtime.CompilerServices.InternalsVisibleTo("SimpleLibrary.Test, PublicKey=002…8b")]
[assembly: System.Runtime.CompilerServices.InternalsVisibleTo("SimpleLibrary.Fakes, PublicKey=002…8b")]
```

Note that you will require the public key and not the assembly's public key *token*, which is often what you see. To acquire the public key of a signed assembly, you will need the "sn.exe" tool that is included with Visual Studio. For example:

```
C:\sn -Tp ClassLibrary1.Fakes.dll
Microsoft (R) .NET Framework Strong Name Utility Version 4.0.30319.17929
Copyright (c) Microsoft Corporation.  All rights reserved.
Public key (hash algorithm: sha1):
0024000004800000940000000602000000240000525341310004000001000100172b76875201e1
5855757bb1d6cbbf8e943367d5d94eb7f2b5e229e90677c649758c6a24186f6a0c79ba23f2221a
6ae7139b8ae3a6e09cb1fde7ce90d1a303a325719c2033e4097fd1aa49bb6e25768fa37bee3954
29883062ab47270f78828d2d2dbe00ae137604808713a28fce85dd7426fded78e1d1675ee3a1e8
0cdcd3be

Public key token is 28030c10971c279e
```

Therefore, the InternalsVisibleTo attribute would be:

```
[assembly: InternalsVisibleTo("ClassLibrary1.Fakes,
PublicKey=0024000004800000940000000602000000240000525341310004000001000100e92decb949446f688ab9f6973436c535bf50acd
1fd580495aae3f875aa4e4f663ca77908c63b7f0996977cb98fcfdb35e05aa2c842002703cad835473caac5ef14107e3a7fae01120a965587
85f48319f66daabc862872b2c53f5ac11fa335c0165e202b4c011334c7bc8f4c4e570cf255190f4e3e2cbc9137ca57cb687947bc")]
```

Optimizing the generation of Fakes

By default, when you add your Fakes assembly, the Fakes framework creates a fakes XML file that attempts to generate Stubs and Shims, thereby introducing Types that you may never use within your unit testing needs and

[30] http://blogs.msdn.com/b/aseemb/archive/2012/03/02/how-to-enable-ute-logs.aspx

[31] http://msdn.microsoft.com/en-us/library/t07a3dye.aspx

[32] http://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.internalsvisibletoattribute.aspx

negatively affecting compilation times. If you have a testable codebase and no need for Shims, turn them off. If you have only a specific area of your solution that needs inclusion, identify it through Type filtering. (See Code generation, compilation, and naming conventions in Microsoft Fakes[33]).

NOTE

Before you specify your type filters, always specify a **<Clear/>**.

Disable generation with a single <Clear/> or with the Disable="true" attribute on the StubGeneration or ShimGeneration element.

The following example turns off **ShimGeneration**, and then generates only Stubs for a *substring* match of types that have **Contoso.MainWeb.Repository** in their name:

```
<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">
  <Assembly Name=" Contoso.MainWeb"/>
  <StubGeneration>
    <Clear/>
    <Add Namespace="Contoso.MainWeb.Repository" />
  </StubGeneration>
  <ShimGeneration Disable="true"/>
</Fakes>
```

You should be aware that the restricted generation has a limited effect on compilation time and by far the biggest optimization you can make is to avoid the re-generation of the fakes altogether. If you are faking an assembly that is unlikely to change then you should compile your fakes assemblies once in a separate project then add those assemblies as 'reference assemblies' to source control. From there, they can be directly referenced by your unit test projects.

Looking under the covers

Let's look at what happens when you alter the configuration of your .fakes file. In this sample (see **Table 5**), we will fake System.dll. This is a prime candidate for generating once and adding to your 'reference assemblies' in version control, because it will not change often. In the next example, we use ILSpy[34] to disassemble the faked assembly and see what types have been generated.

Desc.	Fakes Configuration	Build Time	Assembly Structure and Comments
No Fakes	NA	1s	NA

Default	`<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">`
	`    <Assembly Name="System" Version="4.0.0.0"/>`
	`</Fakes>`
		19.5s	
			(Intentionally small image... you get everything!)
			Note the big jump in compilation time, hence our recommendation to generate once and check in assemblies that are unlikely to change.

[33] http://msdn.microsoft.com/en-us/library/hh708916.aspx

[34] http://www.ilspy.net/

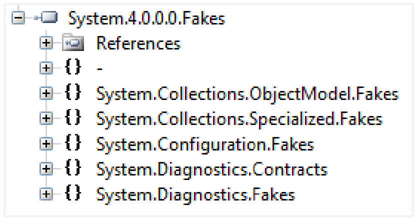
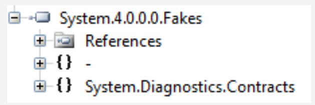
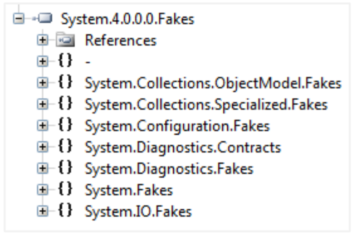
Desc.	Fakes Configuration	Build Time	Assembly Structure and Comments
No Stubs	`<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">` `<Assembly Name="System" Version="4.0.0.0"/>` `<StubGeneration Disable="true"/>` `</Fakes>`	18.6s	 System.4.0.0.0.Fakes; References; {} -; {} System.Collections.ObjectModel.Fakes; {} System.Collections.Specialized.Fakes; {} System.Configuration.Fakes; {} System.Diagnostics.Contracts; {} System.Diagnostics.Fakes Changes to the configuration file have a big impact on the output, but little on compilation time.
No Stubs or Shims	`<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">` `<Assembly Name="System" Version="4.0.0.0"/>` `<StubGeneration Disable="true"/>` `<ShimGeneration Disable="true"/>` `</Fakes>`	18.8s	 System.4.0.0.0.Fakes; References; {} -; {} System.Diagnostics.Contracts This is just an example. You wouldn't want to disable Stubs and Shims!
Limited	`<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">` `<Assembly Name="System" Version="4.0.0.0"/>` `<StubGeneration>` `<Clear />` `<Add Namespace="System!" />` `<Add Namespace="System.IO!"/>` `</StubGeneration>` `</Fakes>`	18.6s	 System.4.0.0.0.Fakes; References; {} -; {} System.Collections.ObjectModel.Fakes; {} System.Collections.Specialized.Fakes; {} System.Configuration.Fakes; {} System.Diagnostics.Contracts; {} System.Diagnostics.Fakes; {} System.Fakes; {} System.IO.Fakes Note the use of <Clear/>. Without the clear, you will get the same output as showin the in the Default behavior.

Table 5 – Looking under the covers

Working with Code Contracts

If you are using Code Contracts[35] you may encounter an error similar to

```
Error 2 Rewrite aborted due to metadata errors. Check output window

…'XYZ' should contain custom argument validation for 'Requires<ArgumentNullException>(action !=
null)' as it implements 'Framework.Testing.IAsserterWhen.When(System.Action)' which suggests it
does. If you don't want to use custom argument validation in this assembly, change the assembly
mode to 'Standard Contract Requires'.
```

In this scenario, you will need to disable Code Contracts for the fake assembly. This is done by using the DisableCodeContracts Compilation attribute in your Fakes configuration file, e.g.

```
<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">
  <Assembly Name="AssemblyName"/>
  <Compilation DisableCodeContracts="true" />
</Fakes>
```

Refactoring code under test

The naming conventions used by Microsoft Fakes can make refactoring code under test somewhat of a challenge. Shim class names prefix "Fakes.Shim" to the original type name. For example, if you were to fake

[35] http://research.microsoft.com/en-us/projects/contracts

System.DateTime, the Shim would be System.***Fakes.Shim***DateTime. The name of a stub class is derived from the name of the interface, with "Fakes.Stub" as a prefix, and the parameter type names appended. The name of the stub type is derived from the names of the method and parameters.

If you refactor your code under test, the unit tests you have written using Shims and Stubs from previously generated Fakes assemblies will no longer compile. At this time, there is no easy solution to this problem other than perhaps using a set of bespoke regular expressions to update your unit tests. Keep this in mind when estimating any refactoring to code which has been extensively unit tested. It may prove a significant cost.

Removing Fakes from a project

Adding and, especially, removing Fakes from your solution can become **non-**trivial. We encourage you to evaluate and perform functional and technical proof-of-concepts before committing to Fakes or any other isolation technology. To remove Fakes from your project, proceed as follows (see **Figures 10, 11,** and **12**):

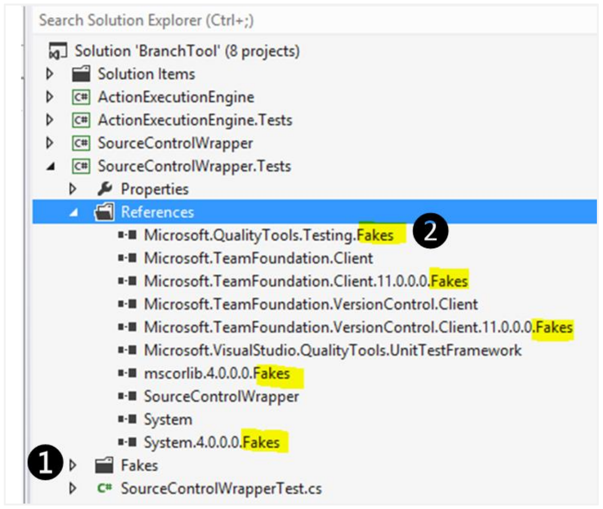

Search Solution Explorer (Ctrl+;)
Solution 'BranchTool' (8 projects)
Solution Items
ActionExecutionEngine
ActionExecutionEngine.Tests
SourceControlWrapper
SourceControlWrapper.Tests
Properties
References
Microsoft.QualityTools.Testing.Fakes
2
Microsoft.TeamFoundation.Client
Microsoft.TeamFoundation.Client.11.0.0.0.Fakes
Microsoft.TeamFoundation.VersionControl.Client
Microsoft.TeamFoundation.VersionControl.Client.11.0.0.0.Fakes
Microsoft.VisualStudio.QualityTools.UnitTestFramework
mscorlib.4.0.0.0.Fakes
SourceControlWrapper
System
System.4.0.0.0.Fakes
1
Fakes
SourceControlWrapperTest.cs

Figure 10 – Removing Fakes from your project

1. Delete the **Fakes** folder and associated files from your project.
2. Delete the **.Fakes** assembly references from your project.
3. Delete the hidden **FakesAssemblies** folder from your project directory.

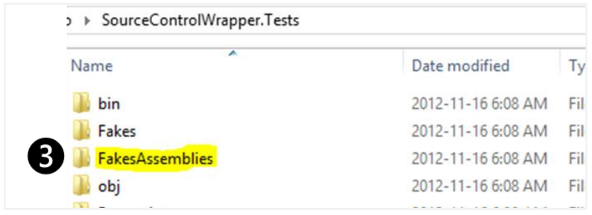
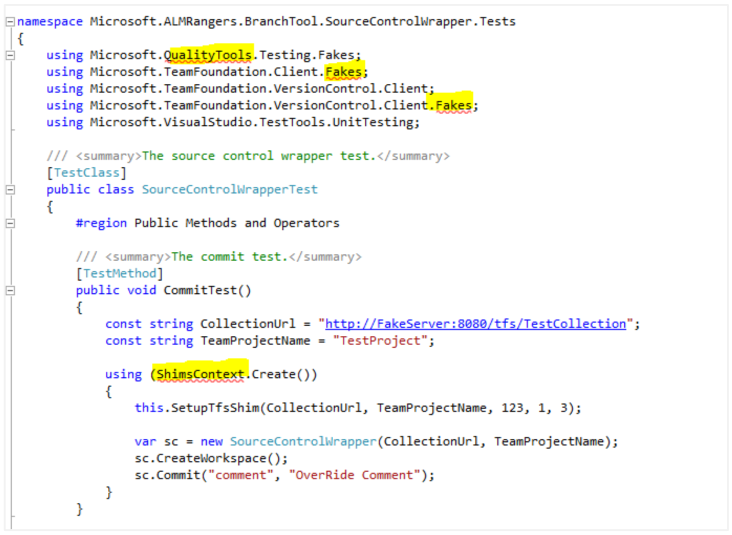


Figure 11 – Removing FakesAssemblies from your project

4. Manually edit your test project file(s). Use the ~~ underline warnings to remove all leftover “using” references and Fakes based code, as shown here:

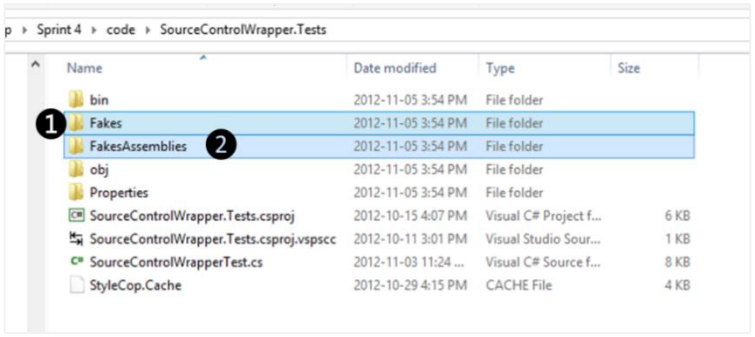


```
namespace Microsoft.ALMRangers.BranchTool.SourceControlWrapper.Tests
{
    using Microsoft.QualityTools.Testing.Fakes;
    using Microsoft.TeamFoundation.Client.Fakes;
    using Microsoft.TeamFoundation.VersionControl.Client;
    using Microsoft.TeamFoundation.VersionControl.Client.Fakes;
    using Microsoft.VisualStudio.TestTools.UnitTesting;

    /// <summary>The source control wrapper test.</summary>
    [TestClass]
    public class SourceControlWrapperTest
    {
        #region Public Methods and Operators

        /// <summary>The commit test.</summary>
        [TestMethod]
        public void CommitTest()
        {
            const string CollectionUrl = "http://FakeServer:8080/tfs/TestCollection";
            const string TeamProjectName = "TestProject";

            using (ShimsContext.Create())
            {
                this.SetupTfsShim(CollectionUrl, TeamProjectName, 123, 1, 3);

                var sc = new SourceControlWrapper(CollectionUrl, TeamProjectName);
                sc.CreateWorkspace();
                sc.Commit("comment", "OverRide Comment");
            }
        }
```

Figure 12 – Removing Fakes code from your project

Using Fakes with Team Foundation Version Control

When adding Fakes, you'll notice that the folders ❶ **Fakes** and ❷ **FakesAssemblies** are created. They contain a number of configuration files and assemblies (see **Figure 13**).

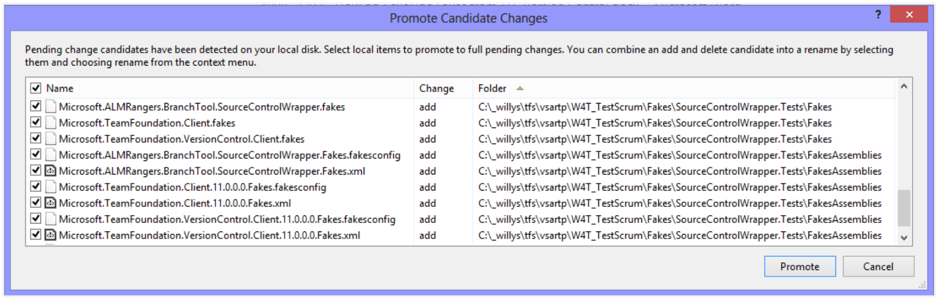


p ▸ Sprint 4 ▸ code ▸ SourceControlWrapper.Tests

Name	Date modified	Type	Size
bin	2012-11-05 3:54 PM	File folder	
❶ Fakes	2012-11-05 3:54 PM	File folder	
FakesAssemblies ❷	2012-11-05 3:54 PM	File folder	
obj	2012-11-05 3:54 PM	File folder	
Properties	2012-11-05 3:54 PM	File folder	
SourceControlWrapper.Tests.csproj	2012-10-15 4:07 PM	Visual C# Project f...	6 KB
SourceControlWrapper.Tests.csproj.vspscc	2012-10-11 3:01 PM	Visual Studio Sour...	1 KB
SourceControlWrapperTest.cs	2012-11-03 11:24 ...	Visual C# Source f...	8 KB
StyleCop.Cache	2012-10-29 4:15 PM	CACHE File	4 KB

Figure 13 – Fakes folders

Team Explorer reports candidate changes when using local workspaces (See **Figure 14**).



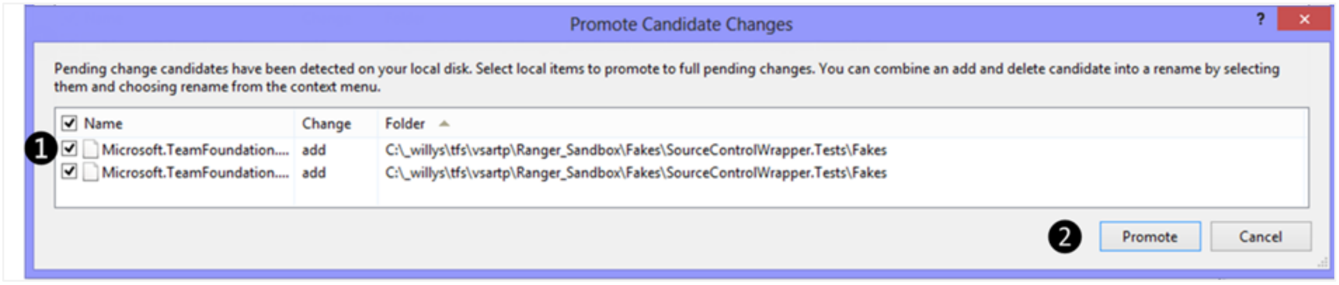
Promote Candidate Changes

Pending change candidates have been detected on your local disk. Select local items to promote to full pending changes. You can combine an add and delete candidate into a rename by selecting them and choosing rename from the context menu.

✓ Name	Change	Folder
✓ Microsoft.ALMRangers.BranchTool.SourceControlWrapper.fakes	add	C:_willys\tfs\vsartp\W4T_TestScrum\Fakes\SourceControlWrapper.Tests\Fakes
✓ Microsoft.TeamFoundation.Client.fakes	add	C:_willys\tfs\vsartp\W4T_TestScrum\Fakes\SourceControlWrapper.Tests\Fakes
✓ Microsoft.TeamFoundation.VersionControl.Client.fakes	add	C:_willys\tfs\vsartp\W4T_TestScrum\Fakes\SourceControlWrapper.Tests\Fakes
✓ Microsoft.ALMRangers.BranchTool.SourceControlWrapper.Fakes.fakesconfig	add	C:_willys\tfs\vsartp\W4T_TestScrum\Fakes\SourceControlWrapper.Tests\FakesAssemblies
✓ Microsoft.ALMRangers.BranchTool.SourceControlWrapper.Fakes.xml	add	C:_willys\tfs\vsartp\W4T_TestScrum\Fakes\SourceControlWrapper.Tests\FakesAssemblies
✓ Microsoft.TeamFoundation.Client.11.0.0.0.Fakes.fakesconfig	add	C:_willys\tfs\vsartp\W4T_TestScrum\Fakes\SourceControlWrapper.Tests\FakesAssemblies
✓ Microsoft.TeamFoundation.Client.11.0.0.0.Fakes.xml	add	C:_willys\tfs\vsartp\W4T_TestScrum\Fakes\SourceControlWrapper.Tests\FakesAssemblies
✓ Microsoft.TeamFoundation.VersionControl.Client.11.0.0.0.Fakes.fakesconfig	add	C:_willys\tfs\vsartp\W4T_TestScrum\Fakes\SourceControlWrapper.Tests\FakesAssemblies
✓ Microsoft.TeamFoundation.VersionControl.Client.11.0.0.0.Fakes.xml	add	C:_willys\tfs\vsartp\W4T_TestScrum\Fakes\SourceControlWrapper.Tests\FakesAssemblies

Figure 14 – Candidate changes

Fakes assemblies are auto-generated and exist within a 'FakesAssemblies' folder under the owning project. These files are re-created with each build. Therefore, they should not be considered as configurable items and should not be checked into version control. Corresponding Fakes configuration files of the format 'assemblyName.fakes' which are created within a 'Fakes' folder under the project **are** configurable items and should be checked into version control. ❶ Select the changes within the Fakes folder and ❷ Promote to a full pending change (see **Figure 15**).


Promote Candidate Changes

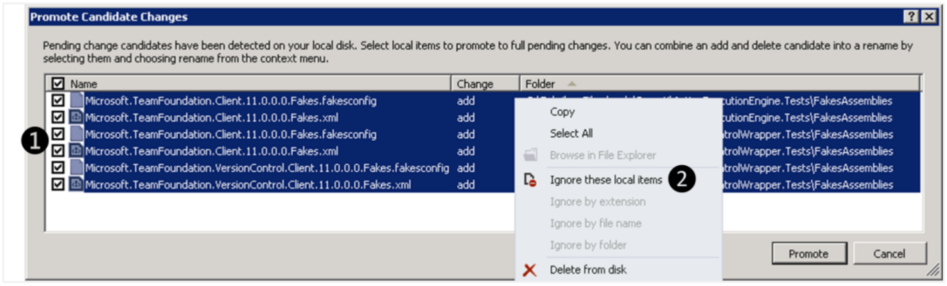
Pending change candidates have been detected on your local disk. Select local items to promote to full pending changes. You can combine an add and delete candidate into a rename by selecting them and choosing rename from the context menu.

Name	Change	Folder
☑ Microsoft.TeamFoundation....	add	C:_willys\tfs\vsartp\Ranger_Sandbox\Fakes\SourceControlWrapper.Tests\Fakes
☑ Microsoft.TeamFoundation....	add	C:_willys\tfs\vsartp\Ranger_Sandbox\Fakes\SourceControlWrapper.Tests\Fakes

Figure 15 – Promote Fakes folder and files to a full pending change

Excluding Fakes – using Team Explorer

To exclude Fakes, ❶ select the changes within the Fakes folder, right-click, and ❷ select Ignore (see **Figure 16**).


Promote Candidate Changes

Pending change candidates have been detected on your local disk. Select local items to promote to full pending changes. You can combine an add and delete candidate into a rename by selecting them and choosing rename from the context menu.

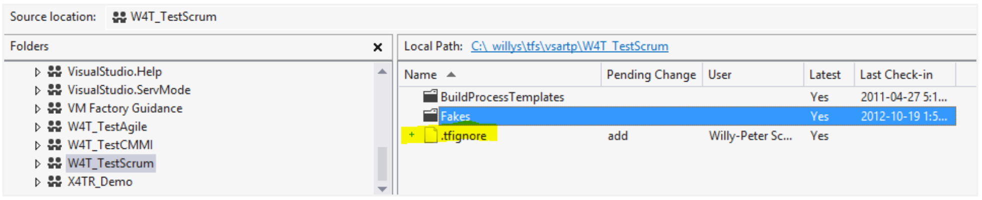
Name	Change	Folder
☑ Microsoft.TeamFoundation.Client.11.0.0.0.Fakes.fakesconfig	add	utionEngine.Tests\FakesAssemblies
☑ Microsoft.TeamFoundation.Client.11.0.0.0.Fakes.xml	add	utionEngine.Tests\FakesAssemblies
☑ Microsoft.TeamFoundation.Client.11.0.0.0.Fakes.fakesconfig	add	trolWrapper.Tests\FakesAssemblies
☑ Microsoft.TeamFoundation.Client.11.0.0.0.Fakes.xml	add	trolWrapper.Tests\FakesAssemblies
☑ Microsoft.TeamFoundation.VersionControl.Client.11.0.0.0.Fakes.fakesconfig	add	trolWrapper.Tests\FakesAssemblies
☑ Microsoft.TeamFoundation.VersionControl.Client.11.0.0.0.Fakes.xml	add	trolWrapper.Tests\FakesAssemblies

Figure 16 – Ignoring Fakes files from pending changes

Alternatively, select each change separately, which enables additional ignore (by extension, file name and folder) options.

Excluding Fakes – using .tfignore

Using Team Explorer, you are indirectly updating the Team Project .tfignore file, which ensures that matching file specifications are excluded from version control (see **Figure 17**).


Source location: W4T_TestScrum

Local Path: C:_willys\tfs\vsartp\W4T_TestScrum

Name	Pending Change	User	Latest	Last Check-in
BuildProcessTemplates			Yes	2011-04-27 5:1...
Fakes			Yes	2012-10-19 1:5...
.tfignore	add	Willy-Peter Sc...	Yes	

Figure 17 – Automatically created .tfignore file

The following rules apply to a .tfignore file:

- **#** begins a comment line.
- The ***** and **?** wildcards are supported
- A filespec is recursive unless prefixed by the \ character.
- **!** negates a filespec (files that match the pattern are not ignored).

The .tfignore file can be edited with any text editor and should be added to version control.

You can configure which kinds of files are ignored by placing text file called **.tfignore** in the folder where you want rules to apply. The effects of the .tfignore file are recursive. However, you can create .tfignore files in sub-folders to override the effects of a .tfignore file in a parent folder." - http://msdn.microsoft.com/en-us/library/ms245454.aspx#tfignore

Using Microsoft Fakes with ASP.NET MVC

ASP.NET MVC was built on top of ASP.NET, which has many high-coupled classes that sometimes make it hard to test. Microsoft Fakes can help you isolate the SUT (System Under Test) from the other ASP.NET MVC components. The main idea is to keep focused on testing what really matters without the influence of dependence.

Using Stubs with ASP.NET MVC

With Microsoft Fakes, we can isolate the MVC controller from the app and test only the functionality that is part of the MVC controller. To do so, you must inject the dependency into the controller, usually through the constructor using interfaces (see **Code 13**):

Code 13 – ASP.NET MVC Stub target code

```
public class CustomersController : Controller
{
    private readonly ICustomerRepository customerRepository;

    public CustomersController(ICustomerRepository customerRepository)
    {
        this.customerRepository = customerRepository;
    }

    [HttpPost]
    public ActionResult Create(Customer customer)
    {
        if (ModelState.IsValid)
        {
            this.customerRepository.InsertOrUpdate(customer);
            this.customerRepository.Save();
            return RedirectToAction("Index");
        }

        return this.View();
    }
}
```

It is possible to create stubs using Microsoft Fakes to isolate that dependency. **Code 14** shows how to create a stub to fake a dependency that will be injected into the controller:

Code 14 – ASP.NET MVS Stub sample

```
[TestClass]
public class CustomersControllerTest
{
    private StubICustomerRepository stubCustomerRepository;
    private CustomersController controller;

    [TestInitialize]
    public void SetupController()
    {
        stubCustomerRepository = new StubICustomerRepository();
        controller = new CustomersController(stubCustomerRepository);
    }

    [TestMethod]
    public void CreateInsertsCustomerAndSaves()
    {
        // arrange
        bool isInsertOrUpdateCalled = false;
        bool isSaveCalled = false;
        stubCustomerRepository.InsertOrUpdateCustomer = customer =>
                                                         isInsertOrUpdateCalled = true;
        stubCustomerRepository.Save = () => isSaveCalled = true;

        // act
        controller.Create(new Customer());

        // assert
        Assert.IsTrue(isInsertOrUpdateCalled);
        Assert.IsTrue(isSaveCalled);
    }
}
```

Using Shims with ASP.NET MVC

Sometimes we cannot inject interfaces or create a new one to make the tests easier. For that scenario, we can use Shims. With Shims, we can change the behavior of an object, setting up the expected result in a method or property. The following code shows one scenario that can be solved with Shims:

```
public class AccountController : Controller
{
    [HttpPost]
    public ActionResult Login(LogOnModel model, string returnUrl)
    {
        if (ModelState.IsValid)
        {
            if (Membership.ValidateUser(model.UserName, model.Password))
            {
                FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);
                return Redirect(returnUrl);
            }
            ModelState.AddModelError("", "The user name or password incorrect.");
        }

        return View(model);
    }
}
```

To test this action, we have to use Shim types to isolate the Membership and FormsAuthentication classes (see **Code 15**):

Code 15 – ASP.NET MVS Shim sample

```
[TestMethod]
public void Login_with_valid_model_and_valid_user_authenticate_and_redirect()
{
    // arrange
    var model=new LogOnModel{Password = "123", UserName = "usrtest", RememberMe = true};
    var returnUrl = "/home/index";
    bool isAuthenticationCalled = false;
    bool isValidateUserCalled = false;
    RedirectResult redirectResult;

    using (ShimsContext.Create())
    {
        ShimMembership.ValidateUserStringString = (usr, pwd) => isValidateUserCalled = true;
        ShimFormsAuthentication.SetAuthCookieStringBoolean =
            (username, rememberme) =>
            {
                Assert.AreEqual(model.UserName, username);
                Assert.AreEqual(model.RememberMe, rememberme);
                isAuthenticationCalled = true;
            };

        // act
        redirectResult = controller.Login(model, returnUrl) as RedirectResult;
    }

    // assert
    Assert.IsTrue(isValidateUserCalled, "Membership.ValidateUser not invoked");
    Assert.IsTrue(isAuthenticationCalled, "FormsAuthentication.SetAuthCookie not invoked");
    Assert.AreEqual(returnUrl, redirectResult.Url);
}
```

Chapter 5: Advanced Techniques

The focus of this chapter is not strictly unit testing, but rather scenarios that can leverage Fakes under specific conditions. Following established principals such as Test-Driven Development is often the best bet when starting a new project. However, when faced with existing codebases that were not designed with testability in mind, a completely different set of challenges might arise. In many cases, the original development team might no longer be available, and details of the implementation, such as documentation, not readily accessible.

In these circumstances, it's often necessary to understand the behavior of the implementation. The unit testing framework provided with Visual Studio, together with Microsoft Fakes, provides an excellent set of tooling for this. In addition, the inherent ability to selectively run these tests to gain additional insight into what is happening under different conditions, speeds up the learning process. As we will also see, the resulting code yields many artifacts, often referred to as "Emulators," which can be leveraged for "conventional" unit testing and applied to future development, which will hopefully include refactoring to improve the testability.

The reference code base is the traffic simulator we used in Exercise 4. It has a number of elements that are tightly coupled and do not expose any means of breaking the complexity chain. Let's work through some of the challenges this app brings.

Dealing with Windows Communication Foundation (WCF) service boundaries

Many apps are implemented as multiple processes that communicate via services. You can set up an environment for the entire system by using Microsoft Test Manager and its Lab Management feature, but this is usually unsuitable for tasks such as unit and behavioral testing.

If the client side is written such that an instance of the interface (contract) is retrieved from a factory class, then it is a simple matter to have the factory return an in process replacement. However, if the WCF client is internally instantiated, there no direct opportunity to replace the calls.

```
private void UpdateRoadwork()
{
    var client = new RoadworkServiceReference.RoadworkServiceClient();
    var locations = new List<RoadworkServiceReference.Block>();

    // Initialization of locations removed for clarity…
    var impediments = client.RetrieveCurrent(locations.ToArray());
```

If you want to test this code, or any code that will invoke the UpdateRoadwork() method, it's necessary to deal with this hard coded situation. Refactoring to eliminate the problem is probably the best bet, but there may be times when this is not desirable or possible.

Breaking the Dependency

The simplest solution is to Shim the generated WCF client and provide our own implementation. This does not require a running instance of the service (see **Code 16**).

Code 16 – Minimal Shim for WCF client

```
using (ShimsContext.Create())
{
    RoadworkServiceReference.Fakes.ShimRoadworkServiceClient.Constructor =
        (real) => { };

    var intercept = new
```

```
        FakesDelegates.Func<RoadworkServiceReference.RoadworkServiceClient,
            RoadworkServiceReference.Block[], RoadworkServiceReference.Impediment[]>(
        (instance, blocks) =>
            {
                // Body of Shim removed for brevity…
            });
    RoadworkServiceReference.Fakes.ShimRoadworkServiceClient.AllInstances.RetrieveCurrentBlockArray = i
ntercept;
```

It's important to note that in addition to creating a shim on the specific operation [method], a shim was also created on the constructor. This is because the constructor of a "real" WCF client will throw an exception due to missing configuration and the lack of a reachable server.

This approach could have significant limitations. First, the logic required to provide the simple implementation might not be so simple, and second, this approach will mask any serialization related problems, such as passing a derived class that is not recognized as a known type.

Improving the Situation

If the actual service implementation assembly is part of the solution, there's an approach that addresses both of these concerns. We'll create an actual instance and use the real logic, but bypass the actual service aspects.

Because the client and service do not share implementation types, it's necessary to transform the parameters and results. By using a DataContractSerializer, we will also reveal any serialization related issues (see **Code 17**).

Code 17 – Improved Shim for WCF client

```
var services = new Dictionary<RoadworkServiceReference.RoadworkServiceClient,
    RoadworkService.RoadworkService>();

using (ShimsContext.Create())
{
    RoadworkServiceReference.Fakes.ShimRoadworkServiceClient.Constructor = real =>
                                  {
                                    services.Add(real, new RoadworkService.RoadworkService());
                                   };

    var intercept = new FakesDelegates.Func<RoadworkServiceReference.RoadworkServiceClient,
        RoadworkServiceReference.Block[], RoadworkServiceReference.Impediment[]>(
        (instance, blocks) =>
            {
                // ====
                // The following (commented out) code uses explicit transforms, see docs for
                // reasons this may rapidly become difficult, and other potential issues..
                // ====
                // var realBlocks = new List<Models.Block>();
                // foreach (RoadworkServiceReference.Block item in blocks)
                // {
                //    var realBlock = Transform(item);
                //    realBlocks.Add(realBlock);
                // }
                Models.Block[] dataContractTransform =
                    DataContractTransform<RoadworkServiceReference.Block[],
                                                               Models.Block[]>(blocks);
                var realBlocks = new List<Models.Block>(dataContractTransform);
                var service = services[instance];
                var results = service.RetrieveCurrent(realBlocks);
                var impediments = new List<RoadworkServiceReference.Impediment>();
                foreach (var result in results)
                {
                    var clientImpediment = new RoadworkServiceReference.Impediment();
                    clientImpediment.location = Transform(result.Location);
                    impediments.Add(clientImpediment);
                }

                return impediments.ToArray();
            });
    RoadworkServiceReference.Fakes.ShimRoadworkServiceClient.AllInstances.RetrieveCurrentBlockArray = i
ntercept;
```

The completed implementation is available in the Hands-on Lab code at

- Exercise 4\Traffic.AdvancedTechniques\Examples\BreakingServiceBoundaryTechniques.cs

Dealing with non-deterministic calculations

In this example (see **Code 18**) the simulator makes calculations based on the time interval between successive calls. Since this can't be accurately controlled, we'll use a shim to intercept the values being presented to the code, and make our test use these values for comparison.

Timer-based operations

Dealing with code elements that are invoked on a timed basis represents a set of challenges.

- If the time is fast, then it might be impossible to know exactly how many invocations have occurred.
- If the time is slow, then the test time necessary to invoke the requisite number of calls might take an excessive amount of time.

To address both these issues, we can generate a shim over the timer and allow for manually invocation of the timed code (see **Code 18** and **Code 19**).

Code 18 – Shim Timer constructor to capture parameters

```
TimerCallback applicationCallback = null;
object state = null;
TimeSpan interval = TimeSpan.Zero;
System.Threading.Fakes.ShimTimer.ConstructorTimerCallbackObjectTimeSpanTimeSpan = (timer, callback, arg
3, arg4, arg5) =>
{
    applicationCallback = callback;
    state = arg3;
    interval = arg5;
};
```

Code 19 – Invoke desired code deterministically

```
const int IterationCount = 10;
for (int i = 1; i <= IterationCount; ++i)
{
    applicationCallback(state);
    Thread.Sleep(interval);
}
```

Non-repeatable data

Another situation found in certain app types is logic that is based on a distribution created by a random number generator. This makes it impossible to know exactly what the generated data will contain. The simplest way to address this is to Shim the Random class, and provide a deterministic set of values.

In this example we're going to ensure that the cars are all "facing west" instead of having the random orientation in the code.

```
System.Fakes.ShimRandom.Constructor = (real) => { };
System.Fakes.ShimRandom.AllInstances.NextDouble = this.NextDouble;
System.Fakes.ShimRandom.AllInstances.NextInt32Int32 = this.NextInt32Int32;

private int NextInt32Int32(Random random, int i, int arg3)
{
   return (i + arg3) / 2;
}
```

```
private double NextDouble(Random random)
{
   return 0.5;
}
```

The completed implementation is available in the Hands-on Lab code at

- Exercise 4\Traffic.AdvancedTechniques\Examples\NonDeterministicBehaviorTechniques.cs

Since we happen to be dealing with the random number generator, there is one element that people often overlook: *Multiple instances with the same seed will generate the same sequence of numbers*! If you're intending to perform operations in independent sets of random numbers and you're using multiple Random instances to accomplish this goal, you must ensure that the instances all have unique seeds. This will be covered in the next section.

Gathering use-case and other analytical information

In this (see **Code 20**) example, the code under test performs a number of mathematical calculations, however the range and combinations of typical input values is not found. We will use unit tests to gather the values being presented under a variety of conditions.

Validating (private) Implementation Details

Here we will deal with a potential issue with the generation of random numbers. Each time an instance of Random is created, it uses a seed; instances with the same seed will generate the same sequence of random numbers on each run.

For our purposes, we are not concerned with the run to run impact, but rather we want to ensure that each instance of Random is running a unique sequence to avoid multiple instances being in lock-step. Additionally, since we want this test to gather data without actually altering the behavior of the code under test, we must ensure that the random number generator continues to work.

NOTE

Note: The default (parameterless) constructor uses Environment.TickCount, which means that multiple instances created within a very short timeframe could also have the same seed.

Code 20 – Using Shims to validate unique seed values for random

```
System.Fakes.ShimRandom.Constructor = delegate(Random random)
{
    ShimsContext.ExecuteWithoutShims(delegate
    {
        var constructor = typeof(Random).GetConstructor(new Type[] { });
        constructor.Invoke(random, new object[] { });
    });
};

System.Fakes.ShimRandom.ConstructorInt32 = delegate(Random random, int i)
{
    ShimsContext.ExecuteWithoutShims(delegate
    {
        var constructor = typeof(Random).GetConstructor(new[] { typeof(int) });
        constructor.Invoke(random, new object[] { i });
    });
    if (this.values.Contains(i))
    {
        passed = false;
        Assert.Fail("Multiple Random instances Created with identical seed Value={0}", i);
    }

    this.values.Add(i);
};
```

The completed implementation is available in the Hands-on Lab code at

- Exercise 4\Traffic.AdvancedTechniques\Examples\DataGatheringTechniques.cs

Analyzing internal state

This example (see **Code 21**) will verify some of the operations of the simulators routing engine. Our specific goal is to verify that all valid routes are being considered when picking the best route for a given car. The logic for this is contained in the ShortestTime and ShortestDistance classes. Unfortunately, the list of valid routes is a local variable.

Code 21 – Shims to capture local variable values

```
List<Route> consideredRoutes = new List<Route>();
MethodInfo mi = typeof(ShortestTime).GetMethod("SelectBestRoute", BindingFlags.Instance | BindingFlags.
NonPublic);
System.Collections.Generic.Fakes.ShimList<Route>.AllInstances.AddT0 =
    (collection, route) =>
    ShimsContext.ExecuteWithoutShims(() =>
    {
        if (this.IsArmed)
        {
            consideredRoutes.Add(route);
        }
        collection.Add(route);
    });

// TODO: We can Shim the protected method, but without using reflection, there is no way to invoke it f
rom within the shim
// FYI: ExecuteWithoutShims disables ALL Shims, thereby breaking the capture of "consideredRoutes", but
 setting the individual shim to null works.
FakesDelegates.Func<ShortestTime, Car, Route> shim = null;

shim = (time, car) =>
{
    Route route = null;
    IsArmed = true;
    ShimShortestTime.AllInstances.SelectBestRouteCar = null;
    var result = mi.Invoke(time, new object[] { car });
    ShimShortestTime.AllInstances.SelectBestRouteCar = shim;
    route = (Route)result;
    IsArmed = false;
    Assert.IsTrue(consideredRoutes.Count > 0, String.Format("Failed to Find Any Considered Routes from
{0} to {1}", car.Routing.StartTrip.Name, car.Routing.EndTrip.Name));
    return route;
};
ShimShortestTime.AllInstances.SelectBestRouteCar = shim;
```

The completed implementation is available in the Hands-on Lab code at

- Exercise 4\Traffic.AdvancedTechniques\Examples\DataGatheringTechniques.cs

Avoiding duplication of testing structures

In many examples, all of the work necessary to stub/shim code has been performed directly inside the unit test. This might lead to significant duplication because different items can require similar or identical infrastructure.

One common approach is to create classes that combine the shim with the associated functionality and then simply activate these within a ShimsContext. While this is helpful in reducing duplication, it might not help in complex scenarios where either the relationships between classes are tightly coupled or you want to invoke "real" implementations for some or all of the functionality.

A more comprehensive approach is to create Doppelgängers [any double or look-alike, or literally "double goer"]. To accomplish this, we extend the emulator concept so that the new class has an actual instance of the real class, along with the necessary shims and helper functions and possibly even implicit conversions between the two. Since Doppelgängerr is long and difficult to spell, we will simply refer to these as "Testable classes" and they'll be identifiable by having the same name as the real class with "Testable" prepended, for example, TestableCar for Car.

A set of "Testable" classes that mirror the classes in the assembly under test are provided. We encourage you to look at these classes. They allow us, for example, to shim a service call and shim Route initialization, thus avoiding long startup times in TestableCity, or to shim a constructor and Random Number to provide movement control in TestableCar.

The completed implementation is available in the Hands-on Lab code at

- Exercise 4\Traffic.AdvancedTechniques\Examples\AvoidingDuplicationTechniques.cs

Chapter 6: Hands-on Lab

NOTE

Please see Using the Sample Source Code before starting this lab.

Exercise 1: Using Stubs to isolate database access (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.

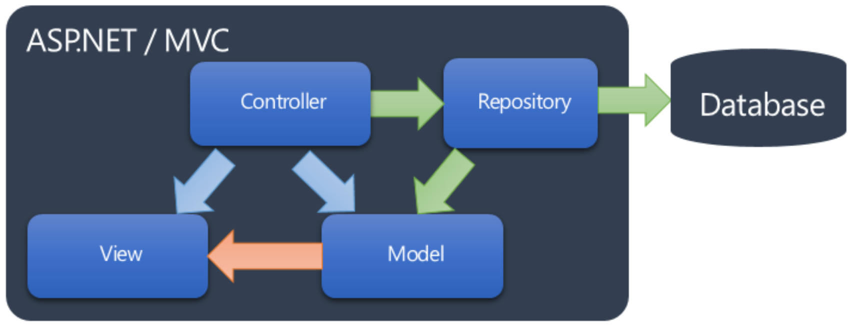


Figure 18 – Environmental dependency

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.

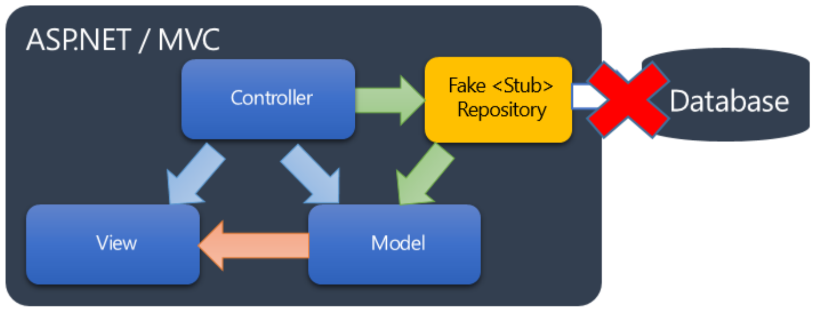


Figure 19 – Dependency isolation

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[36]].
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.

[36] Code generation, compilation, and naming conventions in Microsoft Fakes - http://msdn.microsoft.com/en-us/library/hh708916.aspx

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**).

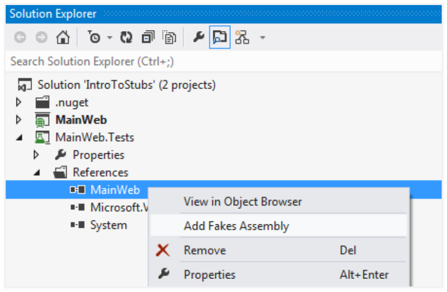


Figure 20 – Adding Fakes assembly

4. At this point review the **MainWeb.Test**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**).

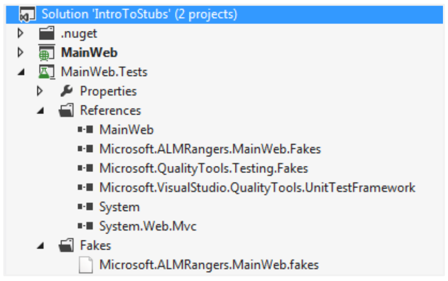

Solution 'IntroToStubs' (2 projects)
.nuget
MainWeb
MainWeb.Tests
Properties
References
MainWeb
Microsoft.ALMRangers.MainWeb.Fakes
Microsoft.QualityTools.Testing.Fakes
Microsoft.VisualStudio.QualityTools.UnitTestFramework
System
System.Web.Mvc
Fakes
Microsoft.ALMRangers.MainWeb.fakes

Figure 21 – View of test project references

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>
```

2. 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*.
3. **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.

3. 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.
4. 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[37]. 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.

5. 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,
```

[37] Parameter Naming Conventions, Code generation, compilation, and naming conventions in Microsoft Fakes -- http://msdn.microsoft.com/en-us/library/hh708916.aspx

```
                                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();
        }
```

6. 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();
}
```

7. 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;

2. 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**.
3. Click **Run All** to execute all tests (for this solution there is 1).
4. 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 (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;
```

3. 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**:

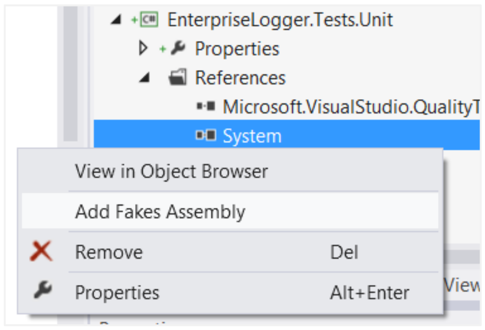


Figure 22 – Add Fakes assembly

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.

2. 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;
```

3. 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.

4. Right-click in the body of the test method and then click **Run Tests**. The test fails.
5. 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.

6. In the using block at the top of "LogAggregatorTests.cs" insert the following line:

```
using Microsoft.QualityTools.Testing.Fakes;
```

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

```
using (ShimsContext.Create())
{
        // Arrange
        …
        // Act
        …
        // Assert
        …
}
```

8. 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;
```

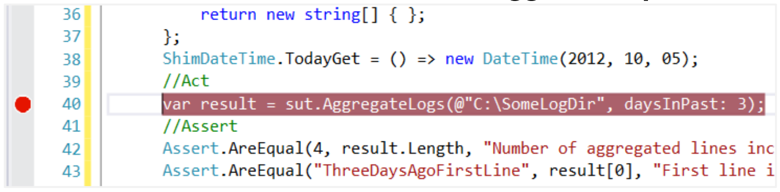
2. Insert the following line above the `//Act` comment:

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

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

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

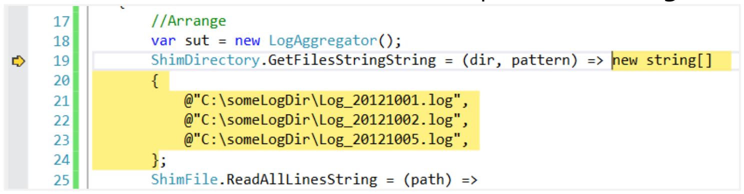
1. Place the cursor on the first line of code below the `//Act` comment (see **Figure 23**).
2. On the main menu, choose **DEBUG**, **Toggle Breakpoint** to add a breakpoint:



```
            return new string[] { };
        };
        ShimDateTime.TodayGet = () => new DateTime(2012, 10, 05);
        //Act
        var result = sut.AggregateLogs(@"C:\SomeLogDir", daysInPast: 3);
        //Assert
        Assert.AreEqual(4, result.Length, "Number of aggregated lines inc
        Assert.AreEqual("ThreeDaysAgoFirstLine", result[0], "First line i
```

Figure 23 – Breakpoint in test method

3. Right-click somewhere in the body of the test method and then click **Debug Tests**.
4. After the breakpoint has been hit, on the main menu, select **DEBUG**, **Step Into** and remember the shortcut (default is **F11**).
5. 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**).



```
        //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) =>
```

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**

Exercise 3: Using Microsoft Fakes with SharePoint (20 – 30 min)

Scenario

SharePoint uses many sealed private classes, which means that it has no externally available constructors for objects that testers need when they write tests. The lack of public interfaces and virtual classes means that it's not possible to use Stub like technologies. This problem is common in any technology that provides a "framework" solution for a developer's bespoke solutions. It's also seen in legacy systems that didn't follow what is now considered good practice — design your system so that it can be easily tested. These legacy systems don't use a form of dependency injection pattern. This leaves a developer with a couple of options if they wish to write tests:

1. Do not do any unit style testing. Rely on UI-driven integration tests with the SharePoint sub-system as a black box.
2. Wrap the whole sub-system that calls SharePoint objects within an implementation of an interface. This can provide a good solution for many projects. Because the implementation details of the SharePoint code are hidden behind an interface, stub-like unit tests can still be used on all the non-SharePoint code. The implementation details of the SharePoint features end up being treated as a black box, so in effect, you are using a black box as described in option 1. These black boxes can, in turn, be tested using the Shim techniques discussed in this document.

However, in the case of SharePoint, this wrapper technique is often not possible because the SharePoint developer is usually creating items such as event receivers and web parts that need to live *within* a SharePoint infrastructure, not on top of it. This makes it impractical to use this form of encapsulation. Neither of these solutions addresses the root issue. We have to be able to substitute our own instances of SharePoint objects for testing purposes. We need to use the functionality provided by Shims.

GOAL

In this exercise, we will see how to use Shims to test the correct functioning of a SharePoint event receiver.

Preparation

This exercise requires that the PC has a supported edition of Visual Studio and SharePoint 2010 Foundation installed (although any later SKU of SharePoint can be used). The SharePoint installation will not be run directly during this exercise, but the assembly files placed in the GAC and Program Files folder will be used in the Faking process. An alternative, if these tools are not available on your PC, is to use the Brian Keller VS2012 ALM demo Hyper-V VM[38] because this has all the required components.

WARNING

SharePoint must be installed on the development PC, not just a copy of the SharePoint DLL, so that all the required assemblies that need to be loaded to create the fakes assembly are available.

Task 1 – Create a sample SharePoint feature

1. In Visual Studio, add a new **.NET 3.5** class library project called **Samples.SharePoint**

NOTE

In production code, you can use any of the SharePoint templates for the feature you are developing, which automatically selects the correct version of the .NET framework. However, to keep this HOL as simple as possible, we're using a basic Class Library. There's no need to create a sandboxed developer SharePoint site. .NET 3.5 has must be selected as the target framework for the project because SharePoint 2010 targets .NET 3.5 SharePoint assembly.

[38] *http://aka.ms/VS11ALMVM*

2. Add a reference to the **Microsoft.SharePoint.dll**. Typically, this file is installed as part of SharePoint Server in **C:\Program Files\Common Files\Microsoft Shared\Web Server Extensions\14\ISAPI**.
3. Rename the file **Class1.cs** to **ContentTypeItemEventReceiver.cs**.
4. In the file **ContentTypeItemEventReceiver.cs**, replace the body of the class with the following code (see **Code 28**), which is our basic sample application. In this sample, we add an event receiver that will be called whenever a new item is added to a SharePoint list. The receiver then edits the title of the new item to match its content type property:

Code 28 – SharePoint ContentTypeItemEventReceiver class

```
using Microsoft.SharePoint;

public class ContentTypeItemEventReceiver : SPItemEventReceiver
{
    public void UpdateTitle(SPItemEventProperties properties)
    {
        using (SPWeb web = new SPSite(properties.WebUrl).OpenWeb())
        {
            SPList list = web.Lists[properties.ListId];
            SPListItem item = list.GetItemById(properties.ListItemId);
            item["Title"] = item["ContentType"];
            item.SystemUpdate(false);
        }
    }

    public override void ItemAdded(SPItemEventProperties properties)
    {
        this.EventFiringEnabled = false;
        this.UpdateTitle(properties);
        this.EventFiringEnabled = true;
    }
}
```

5. Compile the project. It should compile without errors. If there are problems, check for typos.

Task 2 – Create a test

1. Add a new **.NET 4** Unit Test Project **Samples.SharePoint.Tests** to the solution.

NOTE

In this exercise, we use MSTest as our unit-testing framework. However, as long as the Visual Studio Extension Test Adaptors (for example, for nUnit [39] or xUnit [40]) are in place in Visual Studio then any testing framework can be used. Microsoft Fakes does not have a dependency on MSTest.

Unlike the project containing the SharePoint code, which needs to target .NET 3.5, the project containing the tests can target the newer versions of the framework. For SharePoint testing it is recommended to target .NET 4 to avoid problems generating the Fakes.

2. Add a reference to the **Samples.SharePoint** project.
3. Add a reference to the **Microsoft.SharePoint.dll,** like you did for the sample project.

[39] nUnit Test Adaptor http://visualstudiogallery.msdn.microsoft.com/6ab922d0-21c0-4f06-ab5f-4ecd1fe7175d

[40] xUnit Test Adaptor http://visualstudiogallery.msdn.microsoft.com/463c5987-f82b-46c8-a97e-b1cde42b9099

4. In the references section in the test project highlight the **Microsoft.SharePoint** reference and right click. Select **Add Fake Assembly**. This process can take seconds (or even minutes). When it's finished, you should see a file created in the fakes folder and a new reference to **Microsoft.SharePoint.Fakes.dll** (see **Figure 25**).

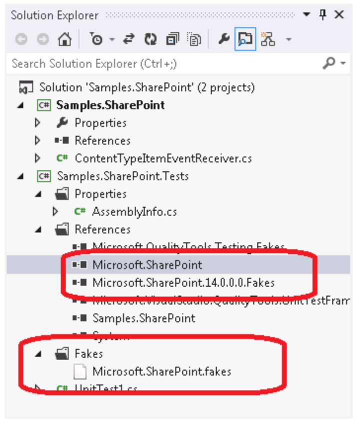

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Samples.SharePoint' (2 projects)
Samples.SharePoint
Properties
References
ContentTypeItemEventReceiver.cs
Samples.SharePoint.Tests
Properties
AssemblyInfo.cs
References
Microsoft.QualityTools.Testing.Fakes
Microsoft.SharePoint
Microsoft.SharePoint.14.0.0.0.Fakes
Microsoft.VisualStudio.QualityTools.UnitTestFram
Samples.SharePoint
System
Fakes
Microsoft.SharePoint.fakes
UnitTest1.cs

Figure 25 **– Added fakes to test project**

WARNING

If for any reason, the creation of this fake fails and you need to repeat the process, make sure you delete the file in the Fakes folder. Otherwise, you'll get an error when you re-run the process.

The most likely reason for a failure is that your test project is targeting .NET 4.5. In this case, Visual Studio will not be able to generate the .NET 3.5-based SharePoint fakes. You have to select .NET 3.5 or .NET 4 as the test project's target framework.

5. Rename the file **UnitTest1.cs** to **SharePointEventTests.cs**.
6. Rename the test **TestMethod1** using a meaningful test name such as **Contributor_AddsNewItem_EventFires**.
7. Add the Using references to the fake SharePoint assembly and faking library.

The code listed at the end of this section shows the completed test (see **Code 29**). The following steps detail the hierarchy of shims being created:

8. Around the whole contents of the test, create a **using** block for the **ShimContext**. This manages the scope of the Shimming operations. The shims operate only in this block.
9. Add two local variables, **systemUpdateHasBeenCalled** and **itemTitleValue**. These are used as flags to signal that the correct code has been called.
10. Create a shim of a **SPItemEventProperties** object and set behaviors for the three properties that we will be calling.
11. Create a shim to intercept the call to the **SPSite** constructor for the site.

NOTE

In the code sample, the parameters for the shimmed constructor start with the @ symbol. This allows reserved words to be used for the variable names that, in this case, can make the intent easier to understand.

12. Inside the **ShimSPSite** block set the behavior on the **OpenWeb** method. This creates a new **SPWeb** shim object.
13. Inside this block, set the behavior on the **List** property to return a shim of a **SPListCollection**.
14. Inside this block, set the behavior for the **Item** property to return a shim of a **SPList**, the passed **GUID** parameter is not used.
15. Inside this block, set the behavior for the **GetItemById** method to returns a shim of a **SPListItem**, the passed **Int** parameter is not used.

16. Inside this block, set the behaviors for **Item** property (get and set) and **SystemUpdate** method. Notice we are setting the value of the local variable created at the top of this test to show that certain calls have been made.
17. Finally, after the nested shim creation, create an instance of the class under test.
18. Add a call to the event receiver we wish to test.
19. Add two asserts to make sure all the correct calls were made when the event method was called.
20. Compile the project. It should compile without errors. If there are problems, check for typos (check against the complete listing at the end of this section).
21. Open the Visual Studio Test Explorer (choose **Test**, **Windows**, **Test Explorer**). You should see the new test listed; if it is not present, rebuild the solution.
22. Choose **Run All**.
23. The test should run and pass.

The complete source for the unit test is shown here (see **Code 29**)

Code 29 – Complete source code for the unit test.

```
namespace Microsoft.ALMRangers.FakesGuide.Sharepoint.Tests
{
    using System;
    using Microsoft.QualityTools.Testing.Fakes;
    using Microsoft.SharePoint.Fakes;
    using Microsoft.VisualStudio.TestTools.UnitTesting;

    [TestClass]
    public class SharePointEventTests
    {
        [TestMethod]
        public void The_item_title_is_set_to_the_content_type_when_event_fires()
        {
            using (ShimsContext.Create())
            {
                // arrange
                // create the local variables we will write into to check that the correct methods are
called
                var systemUpdateHasBeenCalled = false;
                var itemTitleValue = string.Empty;

                // create the fake properties
                var fakeProperties = new ShimSPItemEventProperties()
                {
                    WebUrlGet = () => "http://fake.url",
                    ListIdGet = () => Guid.NewGuid(),
                    ListItemIdGet = () => 1234
                };

                // create the fake site
                ShimSPSite.ConstructorString = (@this, @string) =>
                {
                    new ShimSPSite(@this)
                    {
                        OpenWeb = () => new ShimSPWeb()
                        {
                            ListsGet = () => new ShimSPListCollection()
                            {
                                ItemGetGuid = (guid) => new ShimSPList()
                                {
                                    GetItemByIdInt32 = (id) => new ShimSPListItem()
                                    {
                                        ItemGetString = (name) => string.Format("Field is {0}", name),
                                        SystemUpdateBoolean = (update) => systemUpdateHasBeenCalled =
                                                                                          true,
                                        ItemSetStringObject = (name, value) => itemTitleValue =
                                                                                value.ToString()
                                    }
                                }
                            }
                        }
                    };
                };

                // create the instance of the class under test
                var cut = new ContentTypeItemEventReceiver();

                // act
                cut.ItemAdded(fakeProperties);

                // assert
                Assert.AreEqual(true, systemUpdateHasBeenCalled);
                Assert.AreEqual("Field is ContentType", itemTitleValue);
            }
        }
    }
}
```

NOTE

Microsoft has released SharePoint Emulators, which are a productized version of the Moles SharePoint behaviors. Many of the shim implementations for the core of SharePoint have been packaged and provided as part of the SharePoint Emulators product. SharePoint Emulators are discussed in the Introducing SharePoint Emulators[41] weblog and available via a NuGet package.

REVIEW

In this exercise, we have seen how Microsoft Fakes Shims can be used to enable testing of SharePoint features. You can view the end source code in **Hands-on Lab\Exercise 3\end**.

[41] http://blogs.msdn.com/b/visualstudioalm/archive/2012/11/26/introducing-sharepoint-emulators.aspx

Exercise 4: Bringing a complex codebase under test (20 – 30 min)

Legacy code can pose challenges for refactoring, specifically where the code is tightly-coupled or makes little use of interfaces. To refactor code, it's preferable to have unit tests that confirm the existing behavior of a component before you make any changes.

GOAL

In this exercise, we will use Shims and Stubs to bring a complex codebase under test.

Scenario

The following scenario uses the Traffic Simulator application in the **Exercise 4** folder. Within the **Traffic Core** component, a number of classes act as Models for the Traffic UI component. You will begin the process of creating tests for the City and Car classes to assert their current behavior. The **City** class exposes the layout of the city within the Traffic simulator. This class consumes the **Traffic.RoadworkService** WCF service via a proxy class that is invoked from a private method. That private method is invoked from a callback to a **System.Threading.Timer** instance that is created in the setter of a public property named **Run**. You'll use Shims to produce unit tests against the City class that allow you to shim out references to the WCF service and the Timer instance. The Car class represents a vehicle within the Traffic simulator. This class consumes a number of other objects within the Traffic Core component via a property named **ShouldMove**. You'll use a combination of Stubs and Shims to get this property under test.

Task 1 – Create a test project for the Traffic.Core component

1. In the **ComplexDependencies** solution, add a new Visual C# Unit Test Project called **Traffic.Core.Tests**.
2. Add a class file called **CityTests.cs**.
3. In the newly-created Unit Test project, add a reference to the **Traffic.Core** project.

After we've viewed the code under test, we can add a unit test to test the Run property of the City class.

4. Open the **City.cs** class in the **Model** folder of the **Traffic.Core** project.
5. Navigate to the public **Run** property and view the code. Notice that this property's setter has a dependency on a **System.Threading.Timer** instance, which invokes the **OnTimer** method.
6. Navigate to this method by clicking the **OnTimer** call within the **Run** method. Right-click and select **Go To Definition**.

You'll notice that this method calls the **UpdateRoadwork()** method, which contains the reference to the service client proxy. Therefore, any test method for the Run property will have a dependency upon both a **Timer** instance and the **RoadwordServiceClient**.

Task 2 – Produce a unit test for the City.Run property

Before using Fakes, we'll attempt to produce a unit test that ensures that the Run property can be set to true.

1. In the file **CityTests.cs**, rename **TestMethod1()** to **City_CanSetRunProperty_True()**.
2. Update your references to include System.Threading and Traffic.Core.Models, and add using directives:

```
using System;
using System.Threading;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Microsoft.ALMRangers.FakesGuide.ComplexDependencies.Traffic.Core.Models;
```

3. Add the following code to the method:

```
    [TestMethod]
    public void City_CanSetRunProperty_True()
    {
        City cityUnderTest = new City();
        bool expected = true;

        cityUnderTest.Run = expected;
        Thread.Sleep(TimeSpan.FromSeconds(5));

        Assert.AreEqual<bool>(expected, cityUnderTest.Run, "City.Run property should be set to true.");
    }
```

If you attempt to run this code, the test will fail because the underlying call to the Roadwork service will not have been invoked. Additionally, we can consider this to be brittle test code because a call to Thread.Sleep is required to give the Run property time to create the Timer, register the elapsed event, and invoke the Roadwork service. We'll now try to get this property under test by using Shims to detour calls to external dependencies.

Task 3 – Add a Fakes reference to the Traffic.Core assembly

1. Expand the **References** node within the **Traffic.Core.Tests** project, right-click **Traffic.Core**, and select **Add Fakes Assembly**.

This creates Stub and Shim classes for the Traffic.Core component. We now wish to ensure that when a call is made to the Roadwork service, our Shim implementation will be invoked instead of the actual service and that we will return a Stub of the current return type. To make sure that our implementation has been invoked, we'll set a private test Boolean to true within the method. Additionally, we'll also provide an alternative implementation for the RoadworkServiceClient constructor. This ensures just a basic proxy class is enabled.

Task 4 – Modify the unit test for the Run property

1. Add the following references to the Traffic.Core.Tests project:
    - System.Runtime.Serialization
    - System.ServiceModel
2. Within the CityTests class, add the following Using directives:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.QualityTools.Testing.Fakes;
using
Microsoft.ALMRangers.FakesGuide.ComplexDependencies.Traffic.Core.RoadworkServiceReference.Fakes
```

3. To use the Shim implementation methods, we need to wrap the calls where we invoke the Run property with a ShimsContext call. This will ensure all detours happen only when we are executing the code under test. Wrap the existing method content within this code:

```
using (ShimsContext.Create())
{ }
```

4. Under the line declaring the `expected` Boolean local variable, add another local Boolean variable named `hasServiceBeenInvoked` and set it to false.
5. We do not want the actual RoadworkServiceClient constructor to be invoked so we'll use Shims to create an alternative implementation. Continuing under the Boolean you just created, add the following code:

```
ShimRoadworkServiceClient.Constructor = (x) => { }
```

6. Next, provide an implementation of the RetrieveCurrentBlock service operation call via the ShimRoadworkServiceClient class. This method returns an array of Block types; we'll use our implementation

to set our local test variable `hasServiceBeenInvoked` to true and return a Stub class. Immediately below the code you entered in step 4, add the following code:

```
ShimRoadworkServiceClient.AllInstances.RetrieveCurrentBlockArray =
    (instance, blocks) =>
    {
        hasServiceBeenInvoked = true;
        return new List<StubImpediment>
        {
            new StubImpediment
            {
                description = string.Empty, location = blocks.FirstOrDefault(),
                                                        relativeSpeed = double.MinValue
            }
        }.ToArray();
    };
```

7. Add another assert to ensure the hasServiceBeenInvoked variable matches our expectation:

```
Assert.IsTrue(hasServiceBeenInvoked, "City.Run should invoke the Roadwork service");
```

Code 30 shows the entire CityRun test class:

Code 30 – entire test method implementation

```
[TestMethod]
public void City_CanSetRunProperty_True()
{
    using (ShimsContext.Create())
    {
        City cityUnderTest = new City();
        bool expected = true;
        bool hasServiceBeenInvoked = false;

        ShimRoadworkServiceClient.Constructor = (x) => { };
        ShimRoadworkServiceClient.AllInstances.RetrieveCurrentBlockArray =
            (instance, blocks) =>
            {
                hasServiceBeenInvoked = true;
                return new List<StubImpediment>
                        {
                            new StubImpediment
                            {
                                description = string.Empty, location = blocks.FirstOrDefault(), relativ
eSpeed = double.MinValue
                            }
                        }.ToArray();
            };

        cityUnderTest.Run = expected;
        Thread.Sleep(TimeSpan.FromSeconds(5));

        Assert.AreEqual<bool>(expected, cityUnderTest.Run, "City.Run property should be set to true.");
        Assert.IsTrue(hasServiceBeenInvoked, "City.Run should invoke the Roadwork service");
    }
}
```

You can now run this test from the Test Explorer and the test will pass. We have used Shims to provide a detour to the RoadworkServiceClient class, enabling us to isolate the code under test. Our unit test is still brittle because we still need the Thread.Sleep call to give the Run property enough time to initialize the Timer class.

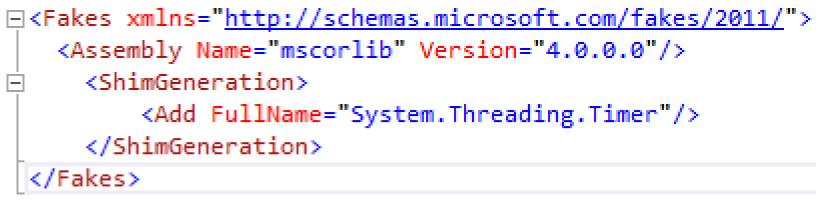
Task 5 – Add a Fakes reference for the System.Timer class

At this point, we now wish to remove the call to Thread.Sleep from our test for the City Run property so that we are not dependent upon a set period of time for the Run property to be initialized. To do this, we will remove the dependency on the Timer constructor by providing an alternative constructor via the ShimTimer class.

1. Expand the **References** node within the **Traffic.Core.Tests** project, right-click **System** and select **Add Fakes Assembly**.

Notice that a System.4.0.0.0.Fakes and a mscorlib.4.0.0.0.Fakes reference are created. This is because the System namespace exists within the mscorlib assembly. Expand the **Fakes** folder in the Test project and you'll see that two files corresponding to the newly added references have been created — **mscorlib.fakes** and **System.fakes**.

2. This step (see **Figure 26**) is necessary because Shims would not be created for the System.Threading namespace by default. Click the **mscorlib.fakes** file and amend the contents as follows:



```
<Fakes xmlns="http://schemas.microsoft.com/fakes/2011/">
  <Assembly Name="mscorlib" Version="4.0.0.0"/>
    <ShimGeneration>
        <Add FullName="System.Threading.Timer"/>
    </ShimGeneration>
</Fakes>
```

Figure 26 **– System.Threading.Timer configuration entry**

3. In the CityTests.cs class, add a Using directive to System.Threading.Timer.Fakes.

Next, modify our existing test to replace the call to the Timer constructor with our own implementation. The Timer constructor used is one that takes a number of parameters to initialize; we need to find the specific one that matches the signature in our ShimTimer class. Again, we will use a local variable to allow us to assert that our implementation was called.

4. Rename the local variable `hasServiceBeenInvoked` to `hasTimerBeenInvoked` and ensure that the corresponding reference to this in the Assert is also renamed. Delete the code that shims the RoadworkServiceClient constructor and the RetrieveCurrentBlockArray calls. Next, add an implementation for the Timer constructor that takes a four parameters – a callback, an object, and two timespans. In this implementation, set the `hasTimerBeenInvoked` property to true. The code should look like this:

```
ShimTimer.ConstructorTimerCallbackObjectTimeSpanTimeSpan = (timer, callback, state, dueTime, period)
=>
{
    // Do nothing else but confirm that our implementation was called
    hasTimerBeenInvoked = true;
};
```

Our refactored City.Run unit test should now look like this (see **Code 31**):

Code 31 – City_CanSetRunProperty_True test

```
/// <summary>
/// Test to ensure that the City Run property can be set to true.
/// </summary>
[TestMethod]
public void City_CanSetRunProperty_True()
{
    using (ShimsContext.Create())
```

```
        {
            City cityUnderTest = new City();
            bool expected = true;
            bool hasTimerBeenInvoked = false;

            ShimTimer.ConstructorTimerCallbackObjectTimeSpanTimeSpan = (timer, callback, state, dueTime,
    period) =>
            {
                // Do nothing else but confirm that our implementation was called here.
                hasTimerBeenInvoked = true;
            };

            cityUnderTest.Run = expected;

            Assert.AreEqual<bool>(expected, cityUnderTest.Run, "City.Run property should be set to true.
    ");
            Assert.IsTrue(hasTimerBeenInvoked, "City.Run should invoke instantiate a Timer instance.");
        }
    }
```

Run the modified test from Test Explorer and the test will pass. The City.Run property is now considered to be under test.

Task 6 – Create a unit test for the Car constructor

Open the **Car.cs** class in the **Model** folder of the **Traffic.Core** project. Note that the constructor takes two parameters – a **Traffic.Core.Algorithms.RoutingAlgorithm** instance and a **System.Windows.Media.Brush** implementation. Both of these input parameters are used to initialize the state of the **Car** instance by setting the state of the **Car.VehicleColor** and **Car.Routing** properties. Additionally, a public **System.Random** property named **RandomGenerator** with a private setter is initialized here. The first unit test to create is one that asserts the current behavior of the constructor in initializing the state of a Car instance.

1. Add a new unit test class called '**CarTests.cs**' to the existing **Traffic.Core.Tests** project.
2. Add the following Using directives:

```
using System.Windows.Media;
using Microsoft.ALMRangers.FakesGuide.ComplexDependencies.Traffic.Core.Algorithms.Fakes;
using Microsoft.ALMRangers.FakesGuide.ComplexDependencies.Traffic.Core.Models;
using Microsoft.ALMRangers.FakesGuide.ComplexDependencies.Traffic.Core.Models.Fakes;
using Microsoft.QualityTools.Testing.Fakes;
using Microsoft.VisualStudio.TestTools.UnitTesting;
```

3. Rename **TestMethod1()** to **Car_Constructor_ShouldInitializeDependentPropertiesSuccessfully()**

In place of an actual **Traffic.Core.Algorithms.RoutingAlgorithm** instance, use a **StubRoutingAlgorithm** type to create a local variable named `expectedAlgorithm` and select a **System.Windows.Media.Brushes** value to assign to a local variable named `expectedColor`.

4. Within the test method, add the following lines of code:

```
var expectedAlgorithm = new StubRoutingAlgorithm();
var expectedColor = Brushes.Aqua;
```

5. Next, create a Car instance using the two test local variables as input parameters to the constructor:

```
Car codeUnderTest = new Car(expectedAlgorithm, expectedColor);
```

6. Finally, assert the state we are expecting the Car instance to be in after the constructor call:

```
    Assert.AreSame(expectedAlgorithm, codeUnderTest.Routing, "The Car constructor should initialize the
    routing algorithm correctly.");
    Assert.AreEqual<Brush>(expectedColor, codeUnderTest.VehicleColor, "The Car constructor should initia
    lize the vehicle color correctly.");
    Assert.IsNotNull(codeUnderTest.RandomGenerator, "The Car constructor should initialize the random ge
    nerator correctly.");
```

7. Run all unit tests within Test Explorer to ensure the test passes.

Task 7 – Add unit tests for the Car.ShouldMove property

Review the **ShouldMove** property in the **Car.cs** class. The property getter has a number of conditional statements that determine what Boolean value should be returned, depending upon the state of the property named **Location**. This property is not initialized by the constructor and within the **ShouldMove** getter, there are several checks for null on both the **Location** and its child properties. The actual logic block is dependent upon a call to the **Location.Road.IsFree** method returning true. This invokes a call to the **DiscoveredRoutes.ToRoutePart** method and then interacts with a local **System.Random** property. Continuing the exercise of getting this code under test, we'll produce some simple unit tests to assert the state of this property. The first test will test the condition where the **Location** property is null.

1. Add a new unit test method entitled **Car_ShouldMoveProperty_ReturnsFalseIfLocationIsNull** to the **CarTests.cs** unit test class. It must be decorated with the TestMethod attribute.
2. In the method body, repeat steps 4 and 5 from Task 6 to arrange the test data.
3. Next, set the **Location** property of the `codeUnderTest` variable to null.
4. Now assert that the `codeUnderTest.ShouldMove` property is false. Provide an error message to tell the developer what to do if the property value is not valid.

The completed unit test is here:

```
/// <summary>
/// Test to ensure that the Car.ShouldMove property returns false where Location is null.
/// </summary>
[TestMethod]
public void Car_ShouldMoveProperty_ReturnsFalseIfLocationIsNull()
{
    var stubAlgorithm = new StubRoutingAlgorithm();
    var testBrush = Brushes.AliceBlue;
    Car codeUnderTest = new Car(stubAlgorithm, testBrush);
    codeUnderTest.Location = null;

    Assert.IsFalse(codeUnderTest.ShouldMove, "The Car.ShouldMove property should return false where Car
.Location is null.");
}
```

The next unit test will test the **ShouldMove** property getter, where the **Location.Road** property is null.

5. Add a new unit test method entitled **Car_ShouldMoveProperty_ReturnsFalseIfLocationRoadIsNull** and repeat the steps 4 and 5 from Task 6.
6. Use an instance of a **StubElementLocation** type with its **Road** property to null. Assign this to the `codeUnderTest.Location` property as follows:

```
        codeUnderTest.Location = new StubElementLocation { Road = null };
```

7. Next, assert that the `codeUnderTest.ShouldMove` property is false. Again, provide a helpful error message. The completed test should look like this:

```
    /// <summary>
    /// Test to ensure that the Car.ShouldMove property returns false where Location is null.
    /// </summary>
    [TestMethod]
    public void Car_ShouldMoveProperty_ReturnsFalseIfLocationRoadPropertyIsNull()
    {
        var stubAlgorithm = new StubRoutingAlgorithm();
        var testBrush = Brushes.AliceBlue;
        Car codeUnderTest = new Car(stubAlgorithm, testBrush);
        codeUnderTest.Location = new StubElementLocation { Road = null };

        Assert.IsFalse(codeUnderTest.ShouldMove, "The Car.ShouldMove property should return false where
    Car.Location.Road is null.");
    }
```

We're now dependent upon the result of the **Location.Result.IsFree** method. Because the ShouldMove property implementation is dependent upon this call returning true, we'll produce a unit test that asserts the state where this property returns false. Because the **Location.Result** property is of type **Block**, we'll need to use a **ShimBlock** instance.

8. Add a new unit test method entitled **Car_ShouldMoveProperty_ReturnsFalseIfLocationRoadIsFreeReturnsFalse** to the **CarTests.cs** class.
9. In the method body, create the local variables `stubAlgorithm` and `testBrush` as per the previous unit tests.
10. Next, add a `using ShimsContex.Create()` statement to detour calls to the **Block** class.
11. Using a ShimBlock class, provide an implementation to ensure any calls to the IsFree method return false. Here's the code:

```
ShimBlock.AllInstances.IsFreeInt32 = (block, position) => { return false; };
```

12. Create a **Car** instance using the `StubAlgorithm` and `testBrush` variables.
13. Create an instance of the **StubElementLocation** class (see step 6) but instead of assigning null to the **Road** property, use an instance of a **StubBlock**.
14. Finally, add an assert statement to ensure that the `codeUnderTest.ShouldMove` property returns false.

The completed unit test is here (see **Code 33**):

Code 32 – Car_ShouldMoveProperty_ReturnsFalseIfLocationRoadIsFreeReturnsFalse implementation

```
/// <summary>
/// Test to ensure that the Car.ShouldMove property returns false where Location.Road.IsFree returns fa
lse.
/// </summary>
[TestMethod]
public void Car_ShouldMoveProperty_ReturnsFalseIfLocationRoadIsFreeReturnsFalse()
{
         var stubAlgorithm = new StubRoutingAlgorithm();
         var testBrush = Brushes.AliceBlue;
         using (ShimsContext.Create())
         {
                  // Ensure any calls to Block.IsFree return false.
                  ShimBlock.AllInstances.IsFreeInt32 = (block, position) => { return false; };

                  Car codeUnderTest = new Car(stubAlgorithm, testBrush);
                  codeUnderTest.Location = new StubElementLocation
                  {
                           Road = new StubBlock()
                  };

                  Assert.IsFalse(codeUnderTest.ShouldMove, "The Car.ShouldMove property should
                                    return false where Car.Location.Road is null.");
         }
}
```

Task 8 – Attempting to Shim the DiscoveredRoutes class

The ShouldMove method makes a call to the DiscoveredRoutes class as shown here:

```
if (this.Location.Road.IsFree(this.Location.Position + 1))
{
    var routePart = DiscoveredRoutes.ToRoutePart(this.Location.Road);
    if (routePart == null)
    {
        return false;
    }

    var probability = routePart.Probability;
    return this.RandomGenerator.NextDouble() < probability;
}
```

To get this part of the method under test, we'd like to Shim the **DiscoveredRoutes** class. However, looking at this class, you'll notice that it is internal. For this exercise, we have decided that the code under test is immutable; this means that we cannot add an InternalsVisibleTo attribute to the Traffic.Core assembly. *Does this mean that our attempts to get the code under test have been futile*? Not entirely. We have increased our code coverage from zero to a certain percentage, thus providing value to our system.

At this point we could choose to adopt a certain level of integration tests to mitigate areas of the system which we cannot cover using unit tests.

REVIEW

In this exercise, we have demonstrated how to start bringing a complex system under test and thus increase code coverage.

In Conclusion

This concludes our adventure into unit testing with Microsoft Fakes. We've touched on theory, introduced you to Shims and Stubs, and hopefully illustrated when to use which. We've also covered various exercises in the Hands-on Lab, guided you through several migration scenarios, and illustrated several advanced topics and techniques.

In the final pages of this guide, you'll find our **Quick Reference** posters. You might find it useful to print these and hang them up at your desk until you are proficient in Microsoft Fakes.

We're at the beginning of the Microsoft Fakes' lifecycle with more to come in future updates to and versions of Visual Studio. We hope you find it a valuable technology to invest in and that you've found this guide useful.

Sincerely

The Microsoft Visual Studio ALM Rangers

Visual Studio
ALM Rangers

Microsoft Fakes – Migration Quick Reference

Visual Studio ALM Rangers

How to Migrate

Generating a Fake assembly

The most noticeable difference between other mocking frameworks and Microsoft Fakes is the process that the developer needs to go through to generate the shim or stub.

In Microsoft Fakes, the developer must right-click the assembly reference they wish to mock and select 'Add Fake Assembly'. This will generate a new assembly that must be referenced to create the stub and shim objects.

Stubs

Stubs will be used when migrating from open source frameworks that mock out interfaces, abstract and virtual classes such as RhinoMocks and Moq.

The changes required to migrate to stubs will usually be straightforward syntactic changes. The basic logical flow of the test will usually remain the same.

For example a test using Moq can be converted to Stubs as follows:

```
DateTime testDate = new DateTime(2012, 1, 1);
Mock<ITransactionManager> mockTM =
   new Mock<ITransactionManager>();
mockTM.Setup(
   tm => tm.GetAccountTransactionCount(testDate))
       .Returns(8);
```

becomes

```
DateTime testDate = new DateTime(2012, 1, 1);
StubITransactionManager stubTM =
    new StubITransactionManager();
stubTM.GetAccountTransactionCountDateTime =
    (date) => (date == testDate) ? 8 : default(int);
```

Shims

Shims will be used when migrating from Microsoft Moles (which is not supported in Visual Studio 2012) or commercial tools such as Typemock Isolator or Telerik JustMock.

Moles

The migration from Moles is usually fairly straightforward. The key change is way that the scope of the faking operation is managed. The HostType attribute is replaced by a ShimsContext using statement:

```
[TestMethod]
[HostType("Moles")]
public void TestMethod()
{
    //...
}
```

becomes

```
public void FakesTestMethod()
{
    using (ShimsContext.Create())
    {
        //...
    }
}
```

A common use for Moles was in the testing of SharePoint solutions. Moles used 'behaviours' to provide prebuilt blocks of faked functionality to assist in this this process. Unfortunately these do not exist for Microsoft Fake Shims so you will need to create these constructs yourself.

Commercial frameworks

Migration from commercial tools will usually be similar to the migration for Stubs once the logic is wrapped in a ShimsContext using statement, as these tools tend to have consistent syntax across their features whether Stub'ing or Shim'ing.

Visual Studio
ALM Rangers

Microsoft Fakes – Tips & Tricks **Quick Reference**

Software Testing

Software testing is the art of measuring and maintaining software quality to ensure that user expectations and requirements, business value, non-functional requirements, such as security, reliability and recoverability, and operational policies are all met.

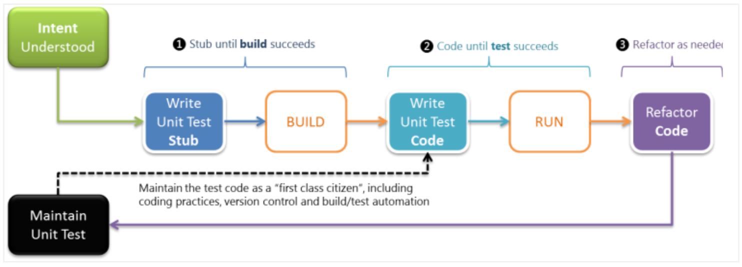
Testing strategies

Testing strategies are traditionally divided into black, white and gray box testing.

Testing types

Typical testing types include, load, regression, smoke, system, **unit** and user acceptance test.

Unit test-driven development paradigm


Intent Understood
❶ Stub until build succeeds
❷ Code until test succeeds
❸ Refactor as neede
Write Unit Test Stub
BUILD
Write Unit Test Code
RUN
Refactor Code
Maintain the test code as a "first class citizen", including coding practices, version control and build/test automation
Maintain Unit Test

Unit test checklist

- ☐ Descriptive naming convention *Classname_Purpose_ExpectedResult*
- ☐ Document your test code
- ☐ Descriptive error and assert messages
- ☐ Adopt an interface driven design
- ☐ Keep it simple
- ☐ Keep it focused
- ☐ Have one logical assert
- ☐ Organize and maintain your tests
- ☐ Test the good, the bad and the edge case

Microsoft Fakes

Stubs … replace a class with a small substitute ("stub") that implements the same interface.

Shims … modify the compiled code at run time, to inject and run a substitute ("shim").

Stub or Shim?

Objective | Consideration	Stub	Shim
Maximum **performance**?	✓	✓
Abstract and Virtual methods	✓	
Interfaces	✓	
Internal types	✓	✓
Static methods		✓
Sealed types		✓
Private methods		✓

Working with strong named assemblies

Default uses the same key used for the shimmed assembly, which can be overridden by specifying the key in the associated .fakes file.

```
<Fakes
xmlns="http://schemas.microsoft.com/fakes/2011/">
  <Assembly Name="ClassLibrary1" Version="1.0.0.0"/>
  <Compilation KeyFile="MyKeyFile.snk" />
</Fakes>
```

Optimizing the generation of Fakes

Default generates Stubs and Shims.

```
<Fakes
xmlns="http://schemas.microsoft.com/fakes/2011/">
  <Assembly Name="Contoso.MainWeb"/>
  <StubGeneration Disable="false">
    <Clear/>
    <Add Namespace="Contoso.MainWeb.Repository" />
  </StubGeneration>
  <ShimGeneration Disable="true"/>
</Fakes>
```

Removing Fakes from project

- ☐ Delete the **Fakes** folder and associated files
- ☐ Delete the **.Fakes** assembly references
- ☐ Delete the hidden **FakesAssemblies** folder
- ☐ Manually edit test project file(s).