FakeItEasy Succinctly

by Michael McCarthy

Syncfusion | Technology Resource Portal

FakeItEasy Succinctly

By

Mike McCarthy

Foreword by Daniel Jebaraj

Technical Reviewer: Pedro Gomes
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Michael McCarthy is a .NET developer with almost 20 years of experience building web-based, business applications. He’s worked in the loan industry, the financial industry, and the pharmaceutical industry. He currently works as a Senior C# Developer at ALMAC, where he just finished a two year, green-field project re-write of their clinical trial software using SOA, DDD, MVC, and TDD.

Over the past five years, he’s switched his efforts from “n-tier” development to utilizing an SOA-based, asynchronous, durable approach to building business software powered by NServiceBus. In those years, he’s transformed into a TDD advocate, has embraced the SOLID principles, and has come to rely on FakeItEasy for its ease of use and flexible API while writing his unit tests.

He lives in Philadelphia with his lovely wife Jessica, and enjoys cooking, playing guitar, and keeping up-to-date on all things SOA, DDD, and TDD.

To stay in touch, you can contact him on LinkedIn (https://www.linkedin.com/in/michaelgmccarthy), read his blog (https://mgmccarthy.wordpress.com), or check out his GitHub projects (https://github.com/mgmccarthy).

Introduction

This book is about FakeItEasy, an open-source mocking framework written by Patrik Hägne (https://github.com/patrik-hagne). FakeItEasy is used for mocking dependencies while unit testing.

Maybe you've used other mocking frameworks and you're wondering what makes FakeItEasy different. Or maybe you've never used a mocking framework before, but you've been writing "stub" classes that implement interfaces for the purpose of your unit tests, and you're interested in finding out how a mocking framework can help.

What sets FakeItEasy apart from other mocking frameworks is that it's much easier to use. In fact, some people don't even use the word "mock" to describe the framework—they say it's a "faking" framework. The fluent interface combined with the easy-to-understand API make it the quickest mocking framework to use in your unit tests.

This book assumes the reader is familiar with concepts such as IoC (Inversion of Control), DI (Dependency Injection), and Unit Testing (using NUnit). Without a basic understanding of these concepts, the rest of this book might not make much sense. Understanding Microsoft's MVC framework is required to get the most from Chapter 10, "MVC and FakeItEasy." Understanding C# delegates (Action<T> and Func<T>) is recommended to make sense of some of FakeItEasy's API, but is not required to get value from this book.

The book also assumes the reader has no knowledge of nor has used FakeItEasy before, but is interested in how FakeItEasy can improve productivity and reduce wasted code written during unit testing.

What do I mean by wasted code?

Have you ever had to unit test a class that takes a dependency and had to write a class that implemented that interface in order to "stub out" behavior for the explicit purpose of your test? If you've done this, you know that these "stubbed out" classes represent a lot of "throw-away" code. They also represent friction when refactoring an interface that the stub class is implementing, because in addition to changing the implementation of the real class that implements the interface, now you have a bunch of "stub" classes that need to be updated as well.

If you answered "yes" to any of the items in the preceding paragraph, then FakeItEasy is the mocking framework for you. You will learn to use FakeItEasy in your unit tests WITHOUT writing those time-wasting, throw-away stub classes.

For those interested in migrating from MOQ (another popular open-source mocking framework) to FakeItEasy after reading this book, I highly recommend Daniel Marbach's blog post (http://www.planetgeek.ch/2013/07/18/migration-from-moq-to-fakeiteasy-with-resharper-search-patterns/) about Migration from Moq to FakeItEasy with ReSharper Search Patterns.

All examples in this book use NUnit as the testing framework. If you are unfamiliar with NUnit, or unit testing in general, I'd recommend Unit Testing Succinctly by Marc Clifton (http://www.syncfusion.com/resources/techportal/ebooks/unittesting).

Hopefully, by the end of this book, you will have a solid understanding of how to use FakeItEasy in your unit tests during setup, configuration, and assertions for all your testing needs.

Conventions Used in This Book

The "SUT"

We'll be exploring FakeItEasy in the context of unit tests. When I set up unit tests, there will always be the Subject Under Test (SUT), which is the class I'm testing that takes a dependency I'm faking using FakeItEasy.

For example, given this interface:

```
public interface IDoSomething
{
    string DoIt();
}
```

Code Listing 1: The IDoSomething interface

If I'm trying to test this class:

```
public class ClassToBeTested
{
    private readonly IDoSomething doSomething;

    public ClassToBeTested(IDoSomething doSomething)
    {
        this.doSomething = doSomething;
    }

    public string GoAheadAndDoIt()
    {
        return doSomething.DoIt();
    }s
}
```

Code Listing 2: The ClassToBeTested class

The setup for the unit test could look like this:

```
[TestFixture]
public class WhenTheClassToBeTestedIsDoingSomething
{
    [SetUp]
    public void Given()
    {
        var sut = new ClassToBeTested(A.Fake<IDoSomething>());
    }
}
```

Code Listing 3: Using SUT to represent the Subject Under Test

The SUT represents the Subject Under Test, which in this case is the **`ClassToBeTested`** class.

"Given" is always the name of the [SetUp] method

In my unit test, I will always represent the setup of the test in method called **`Given`**, which will be decorated by **`[SetUp]`**.

"new up" or "newing up"

Throughout this book I'll use the terms "new up" or "newing up." This means to create an instance of something. An example of this in usage would be "we're going to new up the Customer class."

Source Code

The source code for this book can be found by downloading it from my GitHub account: https://github.com/mgmccarthy/FakeItEasySuccinctly. Although downloading this is not required to understand the book, it can provide deeper insight by allowing you to run/debug the unit tests.

A Note on the Text

I want to give special thanks to Blair Conrad, a member of the FakeItEasy team, whose feedback on the finished text was invaluable in putting the final touches on this book. Blair lent clarity where there was obfuscation, technical editing where corrections were needed, and general grammar and sentence structure recommendations to ground technical explanations.

You can read his blog at http://blairconrad.com/ and contact him directly at blair@blairconrad.com.

Thanks Blair!

Chapter 1 Installing FakeItEasy

Introduction

FakeItEasy does not require the installation of any software of your machine; you just need to add the proper assemblies as references to your project and you're good to go.

You can get these assemblies in two ways:

- NuGet package, installed using Manage NuGet Packages from Solution
- NuGet package, using Package Manager Console (preferred)

In this book, we'll be using FakeItEasy version 1.25.2, NUnit version 2.6.4, ReSharper version 8.2 (for running NUnit tests in Visual Studio), .NET 4.5, and Visual Studio 2013.

Although ReSharper is not required for running NUnit unit tests in Visual Studio, it is the most robust tool for doing the job. You can get a 30-day free trial of ReSharper here: http://www.jetbrains.com/resharper/download/.

Using Manage NuGet Packages from Solution for installing FakeItEasy and NUnit

For those that prefer working with a GUI, this is the easiest way to get the FakeItEasy and NUnit packages in your Visual Studio project. Start with a New Project in Visual Studio and pick "Class Library" for the project type.

Note: Installing packages using Manage NuGet Packages from Solution does not allow you to pick the version of either FakeItEasy or NUnit that is being installed. Manage NuGet Packages from Solution will always take the newest version of the package being installed. If the version numbers vary greatly between what Manage NuGet Packages from Solution will install and the version numbers pointed out in this book, please skip to the Using Package Manager Console for installing FakeItEasy and NUnit section later.

Once the new solution is ready, right-click on the solution in Solution Explorer and choose **Manage NuGet Packages for Solution**.

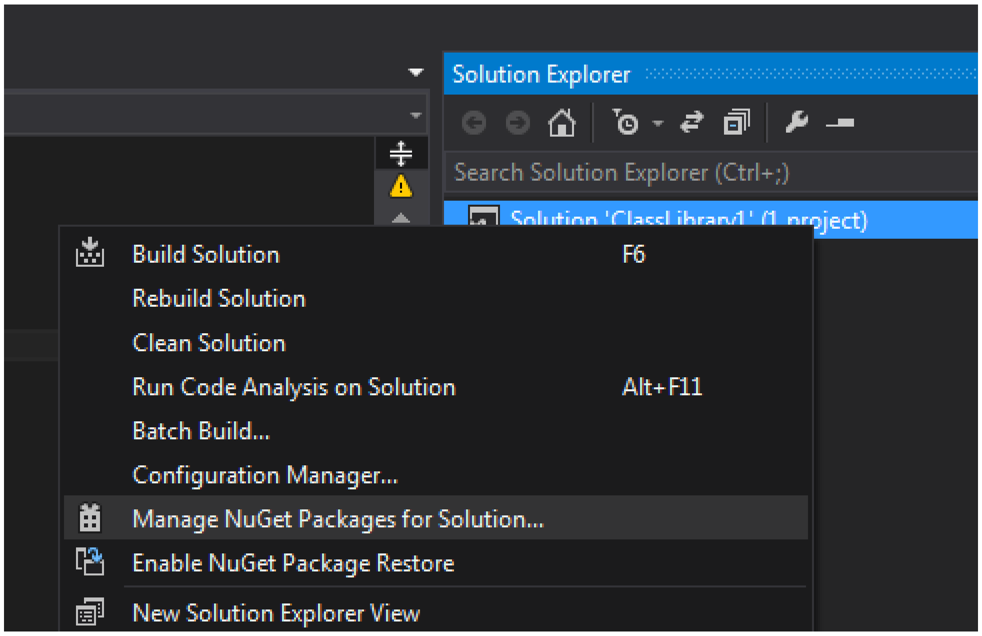
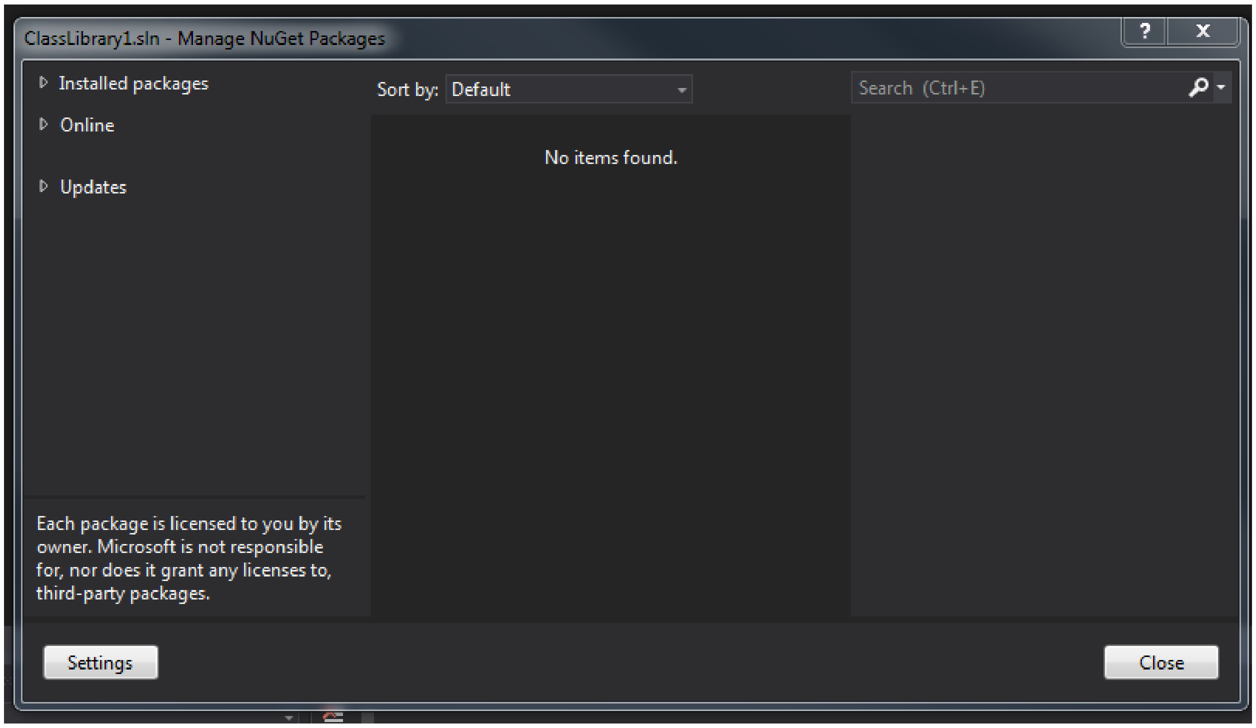

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ClassLibrary1' (1 project)
Build Solution F6
Rebuild Solution
Clean Solution
Run Code Analysis on Solution Alt+F11
Batch Build...
Configuration Manager...
Manage NuGet Packages for Solution...
Enable NuGet Package Restore
New Solution Explorer View

Figure 1: Manage NuGet Packages for Solution

You will see the Manage NuGet Packages window appear.

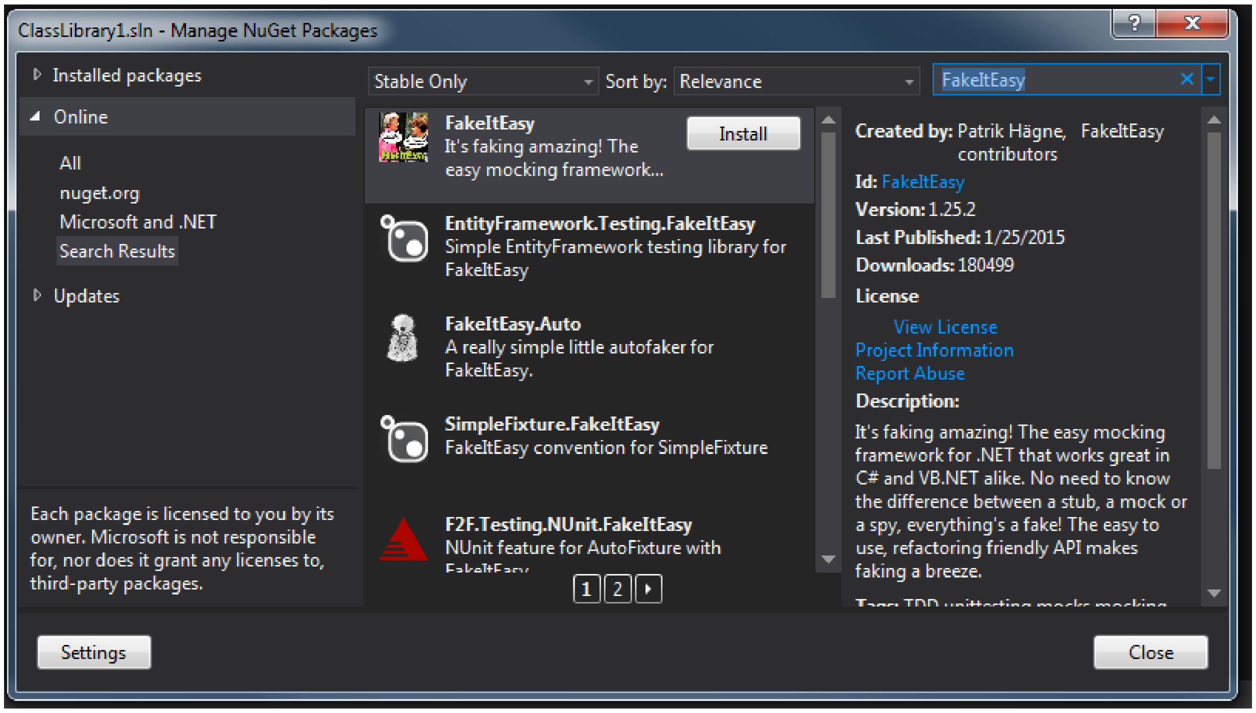


No items found.

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Figure 2: Manage NuGetPackages window

If it's not already open, click the **Online** menu item on the left and then type **FakeItEasy** into the **Search Online** text box at the top-right corner of the window. You should see FakeItEasy appear first in the list; click **Install** to install the package.


ClassLibrary1.sln - Manage NuGet Packages

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

FakeItEasy It's faking amazing! The easy mocking framework...

EntityFramework.Testing.FakeItEasy Simple EntityFramework testing library for FakeItEasy

FakeItEasy.Auto A really simple little autofaker for FakeItEasy.

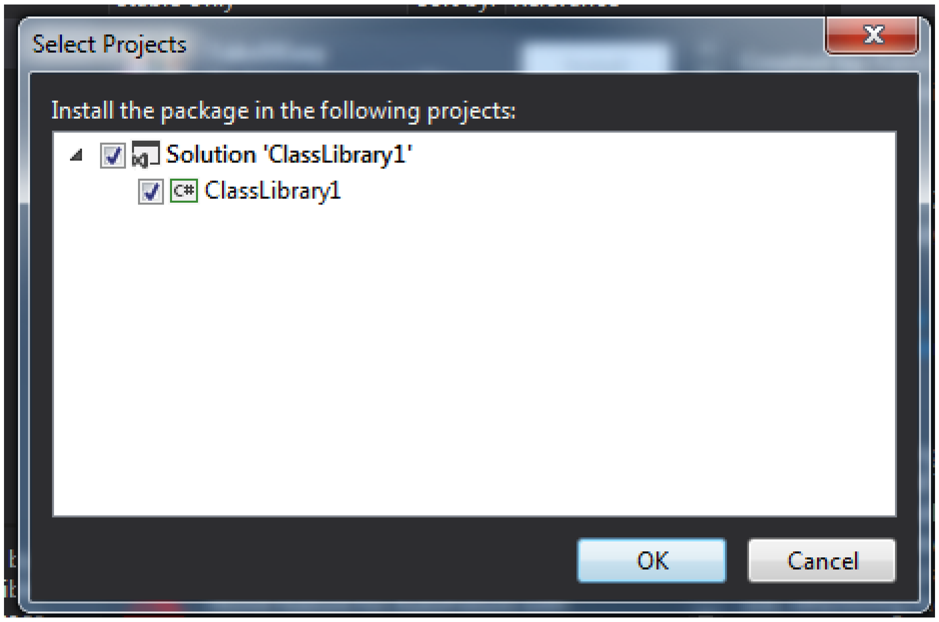
SimpleFixture.FakeItEasy FakeItEasy convention for SimpleFixture

F2F.Testing.NUnit.FakeItEasy NUnit feature for AutoFixture with FakeItEasy

Created by: Patrik Hägne, FakeItEasy contributors
Id: FakeItEasy
Version: 1.25.2
Last Published: 1/25/2015
Downloads: 180499
License
View License
Project Information
Report Abuse
Description:
It's faking amazing! The easy mocking framework for .NET that works great in C# and VB.NET alike. No need to know the difference between a stub, a mock or a spy, everything's a fake! The easy to use, refactoring friendly API makes faking a breeze.

Figure 3: Search for FakeItEasy Online

You'll see the Selected Projects window appear that shows the projects in your solution in which you are installing FakeItEasy. Click **OK**.


Select Projects

Install the package in the following projects:

☑ Solution 'ClassLibrary1'
☑ ClassLibrary1

Figure 4: Selected Projects window

You should see the installation was successful by the addition of the FakeItEasy assembly in the References folder of your project.

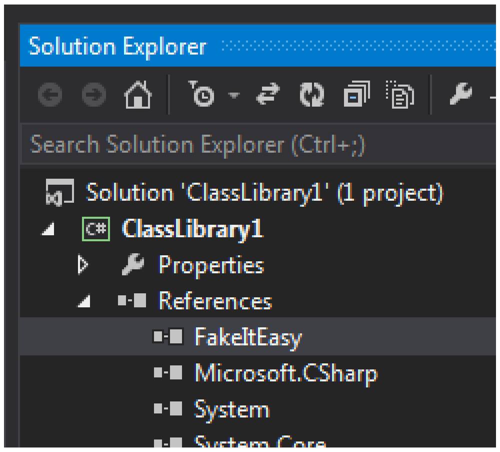


Figure 5: FakeItEasy assembly under References

You'll also see the addition of a packages.config file in the project.

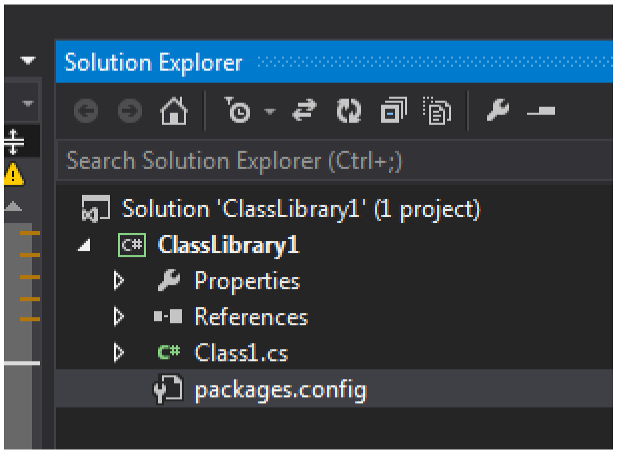
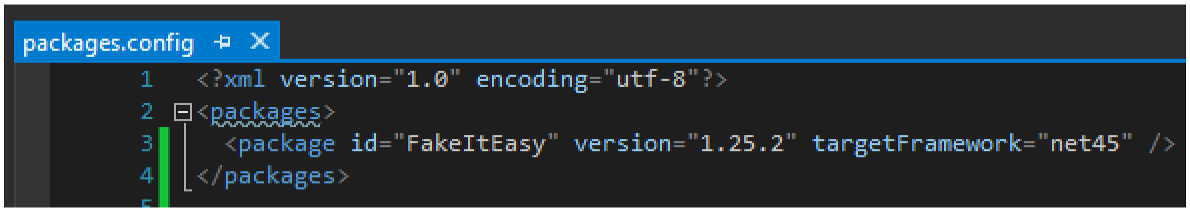


Figure 6: Packages.config file

The contents of the added packages.config file should look like this:



```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="FakeItEasy" version="1.25.2" targetFramework="net45" />
</packages>
```

Figure 7: FakeItEasy added to packages.config

Repeat the process starting at Figure 3, and instead of searching for and installing FakeItEasy, search for and install NUnit.

Using Package Manager Console for installing FakeItEasy and NUnit

If it's not already open, open the Package Manager window by going to the **View** menu in Visual Studio. Select **Other Windows** and choose **Package Manager Console**.

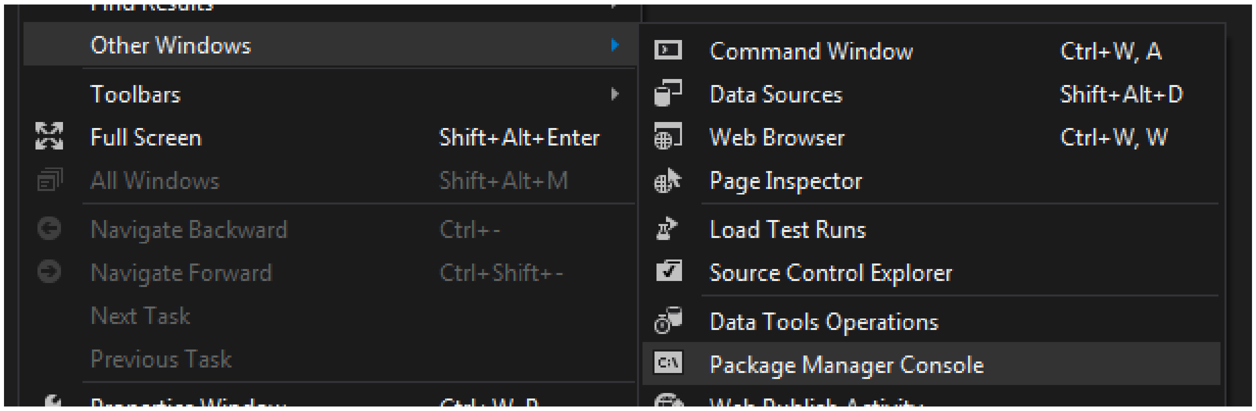
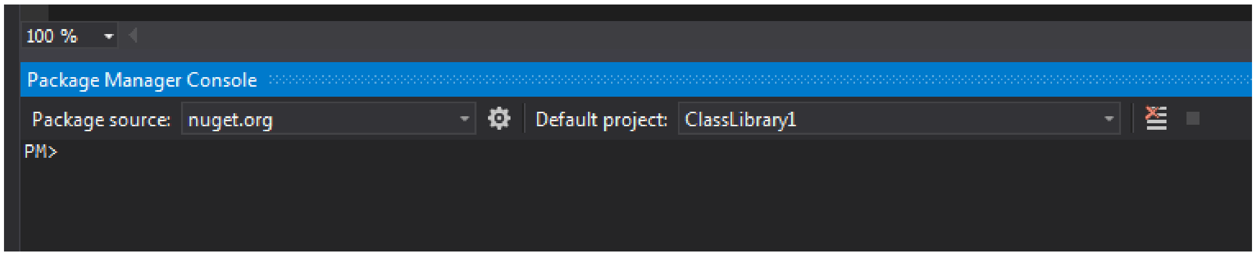


Figure 8: Opening the Package Manager Console

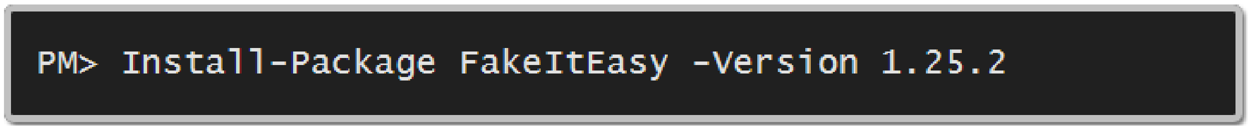
You should now see the Package Manager Console window. Make sure the **ClassLibrary1** project is selected as the Default project in the Default project drop-down menu at the top of the Package Manager Console window.



```
PM>
```

Figure 9: The Package Manager Console window

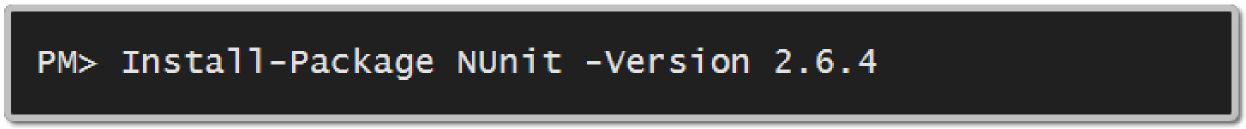
To install FakeItEasy, type the following into the Package Manager Console window:



```
PM> Install-Package FakeItEasy -Version 1.25.2
```

Figure 10: Installing FakeItEasy via the Package Manager Console

To install NUnit, type the following into the Package Manager Console window:



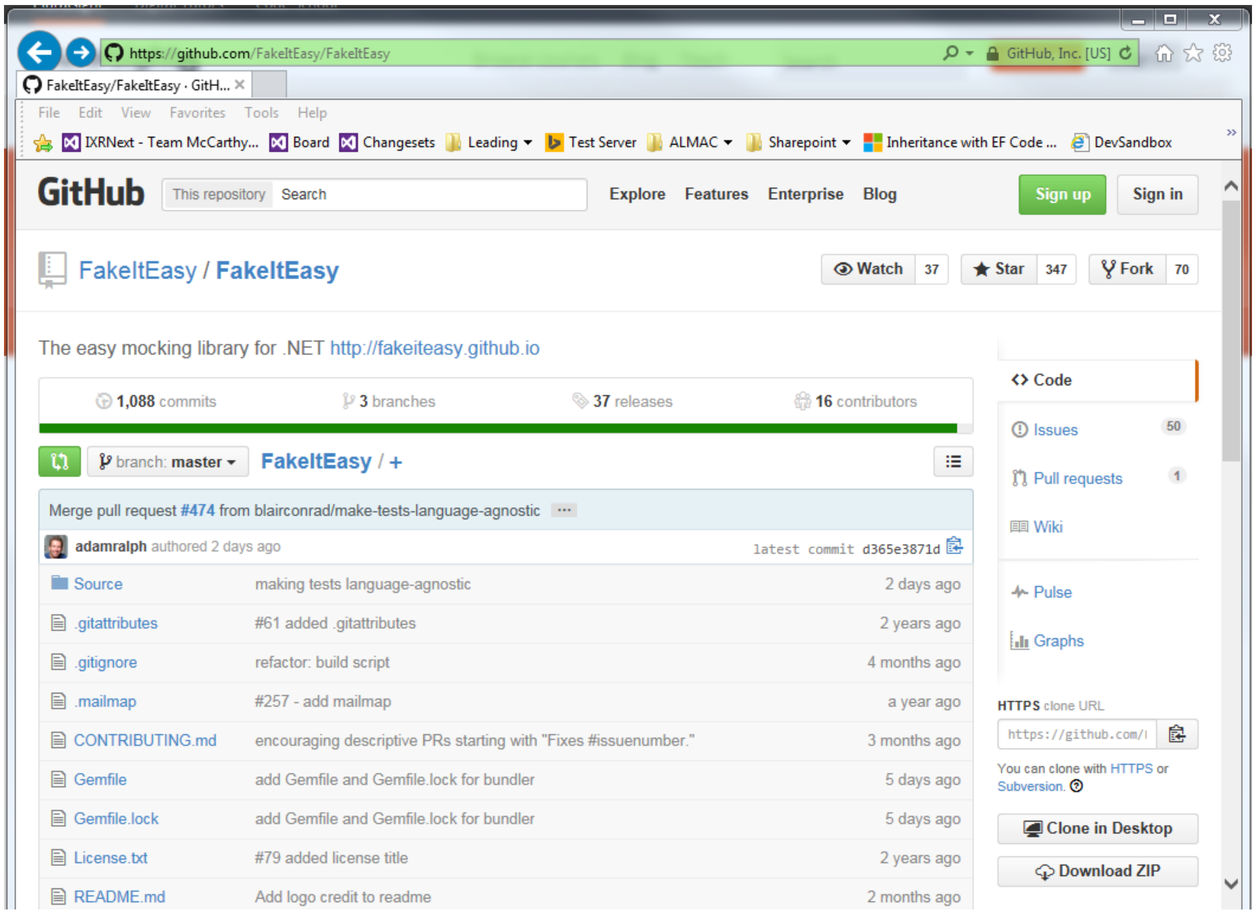
```
PM> Install-Package NUnit -Version 2.6.4
```

Figure 11: Installing NUnit via the Package Manager Console

You can verify installation by checking that both FakeItEasy and NUnit are in the project's References folder and that both references have been added to the packages.config file that was created in the project. This is outlined in Figures 5, 6, and 7 of the previous section.

FakeItEasy Source Code

For those of you who are curious about how exactly FakeItEasy works, and what's really going on behind the scenes, you can browse the source code or do a Git clone from GitHub here: https://github.com/FakeItEasy/FakeItEasy.


https://github.com/FakeItEasy/FakeItEasy
GitHub, Inc. [US]
FakeItEasy/FakeItEasy · GitH...
File Edit View Favorites Tools Help
IXRNext - Team McCarthy... Board Changesets Leading Test Server ALMAC Sharepoint Inheritance with EF Code ... DevSandbox
GitHub
This repository Search
Explore Features Enterprise Blog
Sign up
Sign in
FakeItEasy / FakeItEasy
Watch 37
Star 347
Fork 70
Code
Issues 50
Pull requests 1
Wiki
Pulse
Graphs
HTTPS clone URL
https://github.com/
Clone in Desktop
Download ZIP

The easy mocking library for .NET http://fakeiteasy.github.io

1,088 commits · 3 branches · 37 releases · 16 contributors

branch: master · FakeItEasy / +

Merge pull request #474 from blairconrad/make-tests-language-agnostic

adamralph authored 2 days ago · latest commit d365e3871d

Source	making tests language-agnostic	2 days ago
.gitattributes	#61 added .gitattributes	2 years ago
.gitignore	refactor: build script	4 months ago
.mailmap	#257 - add mailmap	a year ago
CONTRIBUTING.md	encouraging descriptive PRs starting with "Fixes #issuenumber."	3 months ago
Gemfile	add Gemfile and Gemfile.lock for bundler	5 days ago
Gemfile.lock	add Gemfile and Gemfile.lock for bundler	5 days ago
License.txt	#79 added license title	2 years ago
README.md	Add logo credit to readme	2 months ago

You can clone with HTTPS or Subversion.

Figure 12: FakeItEasy on GitHub

Chapter 2 Unit Testing, IoC, and Stubs

Unit Testing Without FakeItEasy

This chapter will walk us through trying to test a class that is untestable, refactoring to allow that class to be tested, creating a "stub" class for the sake of the test, and finally, using FakeItEasy instead of stub classes.

Setting the Stage

Let's explore unit testing first without FakeItEasy. Let's say we have to write a unit test for a class that currently has a dependency on a Repository class.

If you're unfamiliar with the Repository pattern, you can read about it at https://lostechies.com/jimmybogard/2009/09/03/ddd-repository-implementation-patterns/ and at http://martinfowler.com/eaaCatalog/repository.html before continuing. At a very high level, the Repository pattern abstracts database queries and database code away from your classes that consume it.

The following class, **`CustomerService`**, has a dependency on a **`CustomerRepository`** class.

```
public class CustomerService
{
    public Customer GetCustomerByCustomerId(int customerId)
    {
        var customerRepository = new CustomerRepository();
        return customerRepository.GetCustomerBy(customerId);
    }
}
```

Code Listing 4: The CustomerService class with a dependency on the CustomerRepository class

You can see that we're creating a new **`CustomerRepository`** in the **`GetCustomerByCustomerId`** method, and the repository uses the **`GetCustomerBy`** method to query the database for a **`Customer`** by a **`customerId`**.

Unit Testing the CustomerService class

Here's what our unit test looks like:

```
[TestFixture]
public class WhenGettingCustomerById
{
    private const int customerId = 1;
    private Customer result;

    [SetUp]
    public void Given()
    {
        var sut = new CustomerService();
        result = sut.GetCustomerByCustomerId(customerId);
    }

    [Test]
    public void ReturnsTheCorrectId()
    {
        Assert.That(result.Id, Is.EqualTo(customerId));
    }
}
```

Code Listing 5: The unit test for CustomerService

When we go to run this unit test, we're most likely going to get a failure. Why? Because the **`CustomerRepository`** class probably depends on runtime environment (for instance, access to a real database).

Note: I am not showing the implementation of the CustomerRepository class here because it's not important. The way the CustomerRepository communicates with the database (ADO.NET, Entity Framework, NHibernate, RavenDb, etc.) does not matter. Just know that the class will be trying to reach out to some type of database to query for a "Customer" entity.

We know from the SOLID Principles (https://lostechies.com/chadmyers/2008/03/08/pablo-s-topic-of-the-month-march-solid-principles) that our **`CustomerService`** class should NOT be creating (aka, "newing up") a **`CustomerRepository`** inline. We're violating the Dependency Inversion Principle (https://lostechies.com/jimmybogard/2008/03/31/ptom-the-dependency-inversion-principle/).

We also know from good unit testing practices that we should never rely on infrastructure-related items like databases, web servers, and web services to be up and running for our unit tests to run.

Let's fix this problem this problem by introducing Inversion of Control (http://stackoverflow.com/questions/3058/) and see how it changes our unit tests.

Removing the CustomerRepository Dependency using Inversion of Control

Here's our new **`CustomerService`** class:

```
public class CustomerService
{
    private readonly ICustomerRepository customerRepository;
    public CustomerService(ICustomerRepository customerRepository)
    {
        this.customerRepository = customerRepository;
    }

    public Customer GetCustomerByCustomerId(int customerId)
    {
        return customerRepository.GetCustomerBy(customerId);
    }
}
```

Code Listing 6: The CustomerService class now takes a dependency via its constructor

Note the following changes:

- We've added a constructor to the class that takes an interface called **`ICustomerRepository`**. We're injecting that dependency into the class. At runtime, we'll use a container like Microsoft's Unity or StructureMap to resolve the dependency.
- The **`GetCustomerByCustomerId`** method now uses the injected interface to invoke a call to **`GetCustomerBy(customerId)`**.

Note: For more information on how to use Microsoft Unity as a dependency injection container, please see Microsoft Unity Succinctly by Ricardo Peres.

Here is our **`ICustomerRepository`** interface:

```
public interface ICustomerRepository
{
    Customer GetCustomerBy(int customerId);
}
```

Code Listing 7: The ICustomerRepository interface

But We Still Need an Implementation

We hit a problem when we go to refactor the unit test:

```
[TestFixture]
public class WhenGettingCustomerById
{
    private const int customerId = 1;
    private Customer result;

    [SetUp]
    public void Given()
    {
        var sut = new CustomerService(???);
        result = sut.GetCustomerByCustomerId(customerId);
    }
    [Test]
    public void ReturnsTheCorrectId()
    {
        Assert.That(result.Id, Is.EqualTo(customerId));
    }
}
```

Code Listing 8: We need to pass an implementation to Customer Service's constructor

You can see where I have inserted the "???" into the **`CustomerService`**'s constructor. I need to pass an implementation of **`ICustomerRepository`**, and since I can't use the real **`CustomerRepository`**, I have to write a class that implements the interface.

I'll have to write something called a stub.

Stubs to the Rescue (sort of)

Writing a stub class is simple enough; I need to write a new class that implements **`ICustomerRepository`**:

```
public class CustomerRepositoryStub : ICustomerRepository
{
    public Customer GetCustomerBy(int customerId)
    {
        return new Customer { Id = customerId };
    }
}
```

Code Listing 9: The stub class to impelment ICustomerRepository for our unit test

Now that the stub is written, we can use it in our unit test:

```
[TestFixture]
public class WhenGettingCustomerById
{
    private const int customerId = 1;
    private Customer result;

    [SetUp]
    public void Given()
    {
        var sut = new CustomerService(new CustomerRepositoryStub());
        result = sut.GetCustomerByCustomerId(customerId);
    }

    [Test]
    public void ReturnsTheCorrectId()
    {
        Assert.That(result.Id, Is.EqualTo(customerId));
    }
}
```

Code Listing 10: Providing a new CustomerRepositoryStub class to CustomerService's constructor

Note how I create a new **`CustomerRepositoryStub`** class and pass it into the constructor of **`CustomerService.`**

Also note that the implementation of **`CustomerRepositoryStub`** returns a new **`Customer`** (like the real **`CustomerRepository`** would do) based on the **`Id`** I pass into the **`GetCustomerBy`** method.

When we run this test it passes.

We've taken an untestable class, made it testable, and refactored our unit test to correctly take an implementation of an interface and pass it to **`CustomerService`**'s constructor.

But we're not done yet…

Is This a Good Solution?

Let's take a step back at this point and have another look at the **`CustomerRepositoryStub`** class. That's one class I wrote for one very simple interface that was only used in one unit test.

What if I expect a different **`Id`**, **`FirstName`**, and **`LastName`** back on different tests for the same SUT? I have to write more stub classes.

What if I have to implement an interface with 20 methods on it, and those need to change based on the different setup of the SUT? My stub classes get larger.

What if I have to test multiple scenarios driven by conditional statements added to the **`CustomerService`** class dictated by the data returned on the **`Customer`**? I have to write more stub classes.

In my current project at work, we have almost 14,000 unit tests in our system. If I'm writing stub classes for most of, if not all of those tests, we're talking about **14,000** extra classes **at a minimum** written just for the sake of unit testing.

Now change comes. An interface definition changes. Not only do you have to change the real class implementing interface (in our example, **`CustomerRepository`**), but ALL the stub classes you wrote for your unit tests!

There is a better way.

Unit Testing With FakeItEasy

Let's revisit the **`CustomerServiceTests`** class, but this time, instead of writing a stub class to implement an interface for the sake of the unit test, we'll use FakeItEasy.

Here is the unit test using FakeItEasy:

```
[TestFixture]
public class WhenGettingCustomerById
{
    private const int customerId = 1;
    private Customer result;
    private Customer customer;

    [SetUp]
    public void Given()
    {
        customer = new Customer { Id = customerId };

        //create our Fake
        var aFakeCustomerRepository = A.Fake<ICustomerRepository>();

        //set expectations for the call to .GetCustomerBy
        A.CallTo(() => aFakeCustomerRepository.GetCustomerBy(customerId))
            .Returns(customer);

        var sut = new CustomerService(aFakeCustomerRepository);
        result = sut.GetCustomerByCustomerId(customerId);
    }

    [Test]
    public void ReturnsTheCorrectId()
    {
        Assert.That(result.Id, Is.EqualTo(customerId));
    }
}
```

Code Listing 11: The unit test for CustomerService using FakeItEasy

There were a good number of changes made to the unit test, but the biggest changes are the introduction of FakeItEasy. Let's take each line, one at a time.

```
var aFakeCustomerRepository = A.Fake<ICustomerRepository>();
```

Code Listing 12: Creating a fake

This is how you create a "fake" in FakeItEasy. Unlike other mocking frameworks, where you have to differentiate between creating "stubs," "spies," or "mocks," FakeItEasy treats everything as a fake. In this case, creating our fake is very straightforward; we ask FakeItEasy to create a fake of **ICustomerRepository** for us.

Now that we have our fake created, we configure calls on the fake that allow us to test a certain scenario in our SUT. For this simple example, there are no conditional statements in our SUT; we're just asserting that we expect a customer with a matching ID to be returned for the ID we're passing into the **GetCustomerBy** method:

```
A.CallTo(() => aFakeCustomerRepository.GetCustomerBy(customerId)).Returns(customer);
```

Code Listing 13: Configuring a call to the fake and specifying a return value

A.CallTo allows us to configure the fake so we can start telling it what to do. We use the fake to call the **GetCustomerBy** method passing it the **customerId** that we set up in our unit test, and return a specific customer (which also was set up in our unit test). **A.CallTo** allows us to set up our expectations of what should happen in this unit test when this call is made.

If we were to try to pass a different **customerId** to the **GetCustomerBy** method, when we went to assert matching customer IDs in the test method, our unit test would fail.

Finally, we specify the fake's behavior by using **Returns**. Using **Returns** allows us to control what our configured fake will return for the given configuration. In this case, we're specifying that the customer set up in our unit test should be returned by the configured call. If we were to return a different customer in the **Returns** portion of the FakeItEasy call here, our unit test would fail.

As you can see, FakeItEasy allows us to not only assert that the call to **GetCustomerBy** takes a specific **customerId**, but it also allows us to assert that a specific customer is returned.

And finally, we pass the fake to the **CustomerService**'s constructor:

```
var sut = new CustomerService(aFakeCustomerRepository);
```

Code Listing 14: Passing the faked CustomerRepository to CustomerService

The constructor takes the fake, because it sees **aFakeCustomerRepository** as an implementation of **ICustomerRepository**.

The rest of the unit test flows as before, and our test passes.

By adding a couple lines to our unit test, we've introduced FakeItEasy, set expectations on calls made to the fake's method, arguments, and return value, and eliminated the need to write a stub class.

Summary

In this chapter, we started with untestable code, refactored to using stubs after introducing dependency injection to allow our class to be unit tested, and ended with the final refactoring to FakeItEasy, which removed the need to write a unit testing stub class. Next, we'll take a look at the rules that govern how FakeItEasy can be used.

Chapter 3 Introducing FakeItEasy

Before we start writing unit tests and using FakeItEasy, let's start with a solid understanding of the framework. This chapter will focus on what can be faked, how different fakes can be created, how to feed constructor arguments to fakes, what members can be overridden by FakeItEasy, and default behavior "out of the box" when creating a fake.

What Can Be Faked?

Interfaces

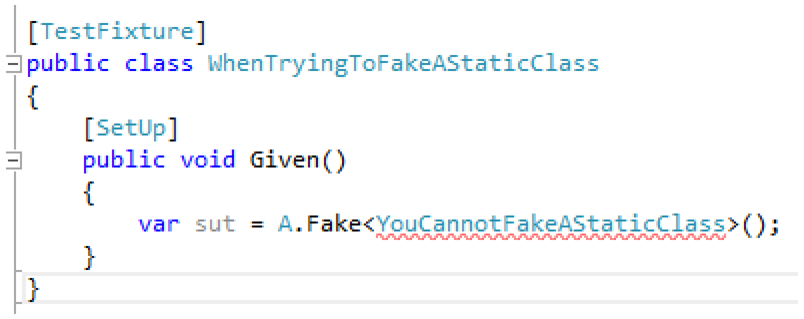
Since many of the dependencies passed into classes are done so via interfaces, most of the time, you'll be creating a fake for an interface.

Classes

There are times when you want to fake a class's functionality. Classes can be injected into other classes as dependencies (although the preferred method is to use an interface). These are the types of classes that can be faked:

- Classes that are not sealed
- Classes that are not static
- Classes that have at least one public or protected constructor whose arguments FakeItEasy can construct or obtain.

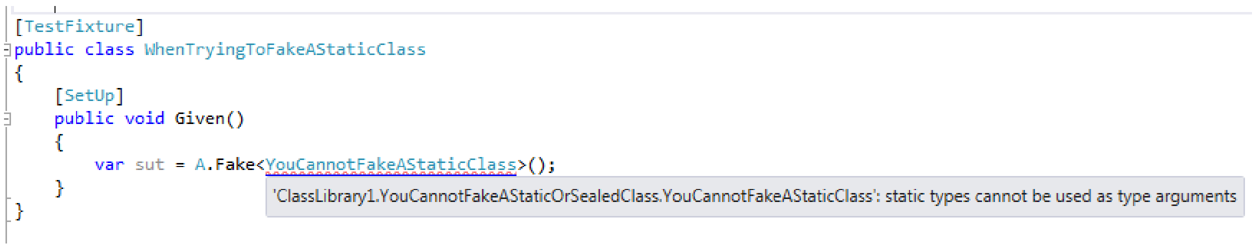
If we do try to fake a static class, this is what we'll see:



```
[TestFixture]
public class WhenTryingToFakeAStaticClass
{
    [SetUp]
    public void Given()
    {
        var sut = A.Fake<YouCannotFakeAStaticClass>();
    }
}
```

Figure 13: You cannot fake a static class

Note that the compiler is already showing us an error. If you hover over the red underlined class name, you'll see this error from the compiler:



```
[TestFixture]
public class WhenTryingToFakeAStaticClass
{
    [SetUp]
    public void Given()
    {
        var sut = A.Fake<YouCannotFakeAStaticClass>();
    }
}
```

'ClassLibrary1.YouCannotFakeAStaticOrSealedClass.YouCannotFakeAStaticClass': static types cannot be used as type arguments

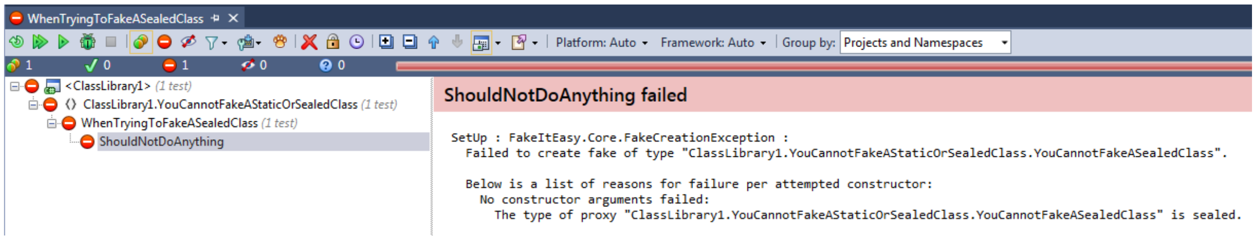
Figure 14: The compiler won't let you fake a static class

If we try to fake a sealed class, unlike the static class, the compiler will not give us an error and will allow the code to compile. However, the unit test will still fail. Here is the class and the test for a static class:

```
public sealed class YouCannotFakeASealedClass
{
    public void DoSomething()
    {
        //some implementation
    }
}

[TestFixture]
public class WhenTryingToFakeASealedClass
{
    private YouCannotFakeASealedClass sut;

    [SetUp]
    public void Given()
    {
        sut = A.Fake<YouCannotFakeASealedClass>();
        sut.DoSomething();
    }

    [Test]
    public void ShouldNotDoAnything()
    {
        A.CallTo(() => sut.DoSomething()).MustHaveHappened(Repeated.Exactly.Once);
    }
}
```

Code Listing 15: A unit test for a sealed class

And here is the unit test failure:


WhenTryingToFakeASealedClass
Platform: Auto
Framework: Auto
Group by: Projects and Namespaces
1
0
1
0
0
<ClassLibrary1> (1 test)
ClassLibrary1.YouCannotFakeAStaticOrSealedClass (1 test)
WhenTryingToFakeASealedClass (1 test)
ShouldNotDoAnything

ShouldNotDoAnything failed

```
SetUp : FakeItEasy.Core.FakeCreationException :
  Failed to create fake of type "ClassLibrary1.YouCannotFakeAStaticOrSealedClass.YouCannotFakeASealedClass".

  Below is a list of reasons for failure per attempted constructor:
    No constructor arguments failed:
      The type of proxy "ClassLibrary1.YouCannotFakeAStaticOrSealedClass.YouCannotFakeASealedClass" is sealed.
```

Figure 15: Unit test failure trying to fake a sealed class

Where do the constructor arguments come from?

WithArgumentsForConstructor

Here is how to pass arguments to your fake using **`WithArgumentsForConstructor`**:

```
public class AClassWithArguments
{
    private readonly string thisIsAnArgument;

    public AClassWithArguments(string thisIsAnArgument)
    {
        this.thisIsAnArgument = thisIsAnArgument;
    }

    public virtual void AFakeableMethod()
    {
        Console.Write(thisIsAnArgument);
    }
}

[TestFixture]
public class WhenFakingAClassWithArgumentsUsingWithArgumentsForConstructor
{
    private AClassWithArguments sut;

    [SetUp]
    public void Given()
    {
        sut = A.Fake<AClassWithArguments>(x => x.WithArgumentsForConstructor(
        () => new AClassWithArguments(A<string>.Ignored)));
        sut.AFakeableMethod();
    }

    [Test]
    public void ACallToAFakeableMethodMustHaveHappened()
    {
        A.CallTo(() => sut.AFakeableMethod()).MustHaveHappened();
    }
}
```

Code Listing 16: Passing arguments to a faked SUT

Note how we use **`A.Fake<T>`** to construct the fake using the class, but our lambda uses **`WithArgumentsForConstructor`** in order for FakeItEasy to get a "hook" into the values passed to the fake's constructor.

Note: Don't worry about `A.string<Ignored>` or MustHaveHappened yet. We'll get to passing values to methods later in the book. For now, know that passing A<string>.Ignored into the fake's method means we either don't care about the value, or won't be asserting against anything that the ignored value might affect for our unit tests. MustHaveHappened is a way to assert

something has, or hasn't happened in the configured fake. WithArgumentsForConstructor will be covered in-depth in Chapter 9, "Faking the Sut."

What members can be overridden?

- Virtual members
- Abstract members
- Interface method when an interface is being faked

Note that static members are missing from this list; they are not overridable.

Default Fake Behavior

This is behavior we get "out of the box" when we create a fake. We will be learning more about behavior of a fake later in the book, starting with Chapter 5, "Configuring Calls to a Fake." For now, the following two sections will outline the behavior you get when simply creating a new fake.

Non-overridable members cannot be faked

What happens when we try to fake a non-overridable method on a class? Given the following class:

```
public class AClassWithAnUnfakeableMethod
{
    public void YouCantFakeMe()
    {
        //some implementation
    }
}
```

Code Listing 17: A class with an unfakeable method

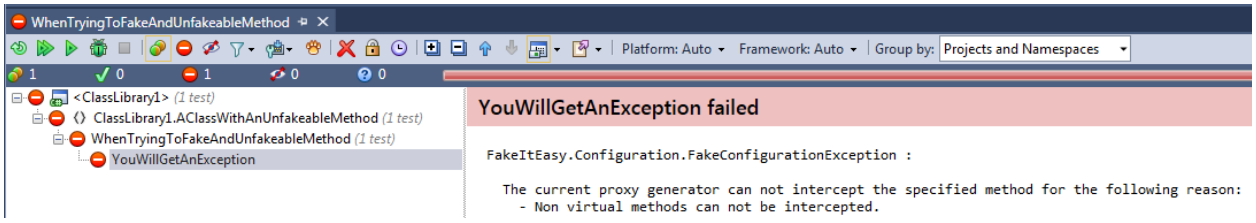
And the following unit test for the class:

```
[TestFixture]
public class WhenTryingToFakeAndUnfakeableMethod
{
    private AClassWithAnUnfakeableMethod sut;

    [SetUp]
    public void Given()
    {
        sut = A.Fake<AClassWithAnUnfakeableMethod>();
        sut.YouCantFakeMe();
    }
    [Test]
    public void YouWillGetAnException()
    {
        A.CallTo(() => sut.YouCantFakeMe()).MustHaveHappened();
    }
}
```

Code Listing 18: Unit test for the unfakeable method

Note: Don't worry about .MustHaveHappened or A.CallTo yet. We'll cover both of these items in Chapter 7, "Assertions."

When we run the unit test, it fails:


WhenTryingToFakeAndUnfakeableMethod
Platform: Auto
Framework: Auto
Group by: Projects and Namespaces
1
0
1
0
0
<ClassLibrary1> (1 test)
ClassLibrary1.AClassWithAnUnfakeableMethod (1 test)
WhenTryingToFakeAndUnfakeableMethod (1 test)
YouWillGetAnException

YouWillGetAnException failed

```
FakeItEasy.Configuration.FakeConfigurationException :

  The current proxy generator can not intercept the specified method for the following reason:
    - Non virtual methods can not be intercepted.
```

Figure 16: Unit test failing when trying to fake an non-overridable member

Note here that FakeItEasy is enforcing those out-of-the-box rules, and telling us specifically why we received an exception when we ran our unit test:

"The current proxy generator can not intercept the specified method for the following reason:
- Non virtual methods can not be intercepted."

Overridable members can be faked

Let's take our class above, change the **`YouCantFakeMe`** method to **`virtual`**, and rename it to **`YouCanFakeMe`**. Here is our class now:

```
public class AClassWithAFakeableMethod
{
    public virtual void YouCanFakeMe()
    {
        //some implementation
    }
}
```

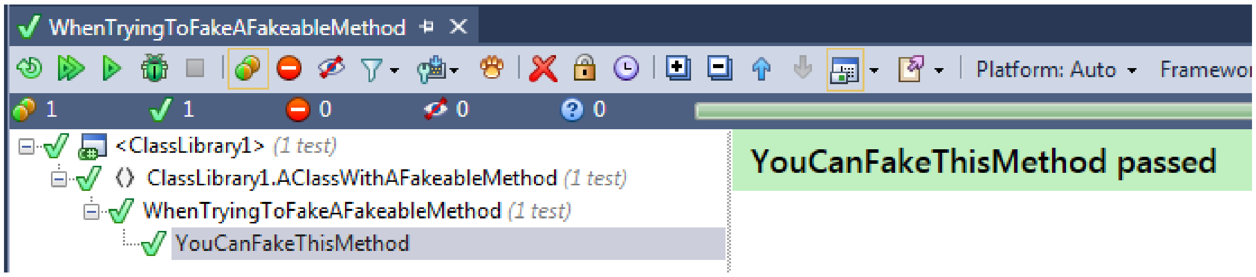
Code Listing 19: YouCanFakeMe is now a fakeable method because it's declared as virtual

And the updated unit test:

```
[TestFixture]
public class WhenTryingToFakeAFakeableMethod
{
    private AClassWithAFakeableMethod sut;
    [SetUp]
    public void Given()
    {
        sut = A.Fake<AClassWithAFakeableMethod>();
        sut.YouCanFakeMe();
    }

    [Test]
    public void YouCanFakeThisMethod()
    {
        A.CallTo(() => sut.YouCanFakeMe()).MustHaveHappened();
    }
}
```

Code Listing 20: The unit test for AClassWithAFakeableMethod

When we run the unit test, this is our output:



```
<ClassLibrary1> (1 test)
  ClassLibrary1.AClassWithAFakeableMethod (1 test)
    WhenTryingToFakeAFakeableMethod (1 test)
      YouCanFakeThisMethod
```

YouCanFakeThisMethod passed

Figure 17: Unit test passing when method is declared as virtual

Note: Although we've allowed our unit test to pass by declaring YouCanFakeThisMethod as virtual, it is not a good practice to change methods on your classes to virtual to allow them to be unit tested. Methods should be declared as virtual if you wish for them to be overridable by inheriting classes. If you find yourself declaring a good amount of your methods as virtual on classes for the sake of unit testing, that's a sign you should step back and take a look at your design. Can or should you be using an interface to represent the contract of the class you're currently using and implement the functionality that used to be in your class in the interface implementation? As usual, there is no hard and fast rule for this, but be on the lookout for this type of code creeping into your codebase.

Summary

In this chapter, we've covered the higher level "rules" that govern how FakeItEasy can work, and looked at example code that illustrates how to use it, and how not to use it. Next, we'll start learning to use FakeItEasy by learning how to create fakes.

Chapter 4 Creating a Fake

This will be the shortest chapter in the book by far, mainly because FakeItEasy makes it very easy for us to create a fake.

How to Create a Fake

Creating a fake is very straightforward (we've already seen this done in previous sections of this book). Use **`A.Fake<T>`** in order to get your fake created. You can create a fake of an interface or class.

Faking an interface:

```
public interface IAmAnInterface
{
    void DoSomething();
}

[TestFixture]
public class WhenFakingAnInterface

{
    [SetUp]
    public void Given()
    {
        var aFakeInterface = A.Fake<IAmAnInterface>();
    }
}
```

Code Listing 21: Creating a fake of an interface

Faking a class:

```
public class AClass
{
    public virtual void DoSomething()
    {
        //some implementation
    }
}

[TestFixture]
public class WhenFakingAClass
{
    [SetUp]
    public void Given()
    {
        var aFakeClass = A.Fake<AClass>();
    }
}
```

Code Listing 22: Creating a fake of a class

Following the rules outlined in Chapter 3, it’s pretty simple to create a fake. But now that we’ve created a fake, what exactly can we do with it?

Chapter 5 Configuring Calls to a Fake

A.CallTo

Now that we know how to create a fake, it's time to start specifying calls to configure on the fake. What does this mean? When we create a fake, we're doing so with an explicit purpose to define some type of behavior we want or expect on the faked item. In order to get that desired behavior, we must configure the call.

We configure a call by invoking the **`CallTo`** method on the static **`A`** class:

```
A.CallTo(() => [theFake].[aFakesMember])
```

Code Listing 23: The CallTo method

`theFake` in Code Listing 23 is a fake we've configured. From there we should be able be able to specify which member on the fake to a call (**`aFakesMember`**). If you look at the IntelliSense presented to you when you type **`A.CallTo`**, you'll see a number of overloads; let's explore those overloads.

Exploring the Overloads

If we look at the overloads of **`CallTo`** in the IDE, we'll see the following:

```
public static class A
{
    public static IVoidArgumentValidationConfiguration
        CallTo(Expression<Action> callSpecification);
    public static IReturnValueArgumentValidationConfiguration<T>
        CallTo<T>(Expression<Func<T>> callSpecification);
    public static IAnyCallConfigurationWithNoReturnTypeSpecified
        CallTo(object fake);
}
```

Code Listing 24: CallTo overloads

Don't worry about the interfaces being returned from the **`CallTo`** methods for now, but instead, look at what the methods take. **`A.CallTo`** is the "gateway" to getting FakeItEasy to do anything. Let's quickly explore each method call before continuing:

- **`CallTo (Expression<Action> callSpecification)`**: We'll use this **`CallTo`** overload to configure members that don't return a value.
- **`CallTo<T> (Expression<Func<T>> callSpecification)`**: We'll use this **`CallTo`** overload to configure members that return a value.
- **`CallTo(object fake)`**: This is the simplest signature of the three overloads. We'll use this **`CallTo`** overload for faking a class's members.

Note: Faking a class is covered in Chapter 9, "Faking the Sut." For now, we'll concentrate on the two overloads that take each of the .NET delegate types.

Before we continue, let's introduce an abstraction we'll work with on and off throughout the book. Sending email is a ubiquitous requirement of most systems built today, so let's introduce an abstraction over the functionality of sending an email, **`ISendEmail`**:

```
public interface ISendEmail
{
    string GetEmailServerAddress();
    bool BodyIsHtml { get; set; }
    void SendMail();
}
```

Code Listing 25: ISendEmail and its members

The **`ISendEmail`** interface has three members: two methods (one that returns a string, and one that returns void), and a property getter/setter.

Calling Methods

Here's how to use **`A.CallTo`** to configure the **`GetEmailServerAddress`** method on the faked **`ISendEmail`** interface:

```
var emailSender = A.Fake<ISendEmail>();
A.CallTo(() => emailSender.GetEmailServerAddress());
```

Code Listing 26: Configuring a call to a method that returns a value

Note: We're purposely not including return behavior in these code listings. We'll be exploring return values in Chapter 6, "Configuring Calls to a Fake."

Here's how to use **`A.CallTo`** to configure the **`SendMail`** method on the faked **`ISendEmail`** interface:

```
var emailSender = A.Fake<ISendEmail>();
A.CallTo(() => emailSender.SendMail());
```

Code Listing 27: Configuring a call to a method that does not return a value

Note here that the configuration is the same for both methods, but if you press F12 into the **`CallTo`** definition for each configured method, you'll see a difference.

When we're setting up a call to **`SendMail`**, you're providing an **`Expression<Action>`** to **`A.CallTo`**, and when you're setting up a call to **`GetEmailServerAddress`**, you're providing an **`Expression<Func<T>>`** to **`A.CallTo`**.

Note: This is because SendMail does not return a value, and GetEmailServerAddress does, which is the difference between the Action<T> and Func<T> delegate types in the .NET framework. More on the difference between Action<T> and Func<T> can be found here: http://stackoverflow.com/questions/4317479/func-vs-action-vs-predicate.

Calling Properties

Here's how to use **`A.CallTo`** to configure the **`BodyIsHtml`** property on the faked **`ISendEmail`** interface:

```
var emailSender = A.Fake<ISendEmail>();
A.CallTo(() => emailSender.BodyIsHtml);
```

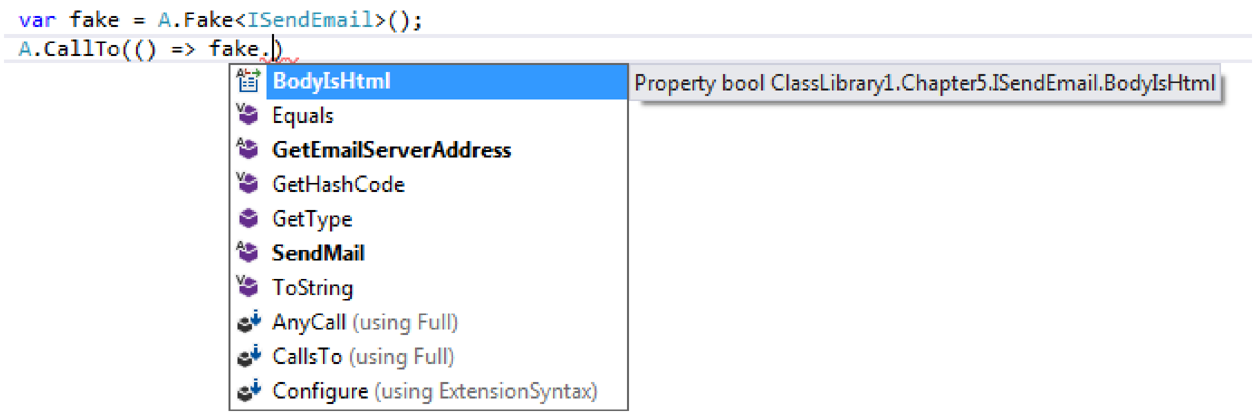
Code Listing 28: Configuring a call to a property

Note: It is possible to call into protected members as well. We'll be going over how to do just that in Chapter 9, "Faking the Sut."

A Closer Look at A.CallTo

Let's dig even deeper and take a look at the IntelliSense we'll see when configuring calls on the faked **`ISendEmail`** interface.

After creating the fake, this is what we'll see available from the fake when starting to type **`A.CallTo`**:



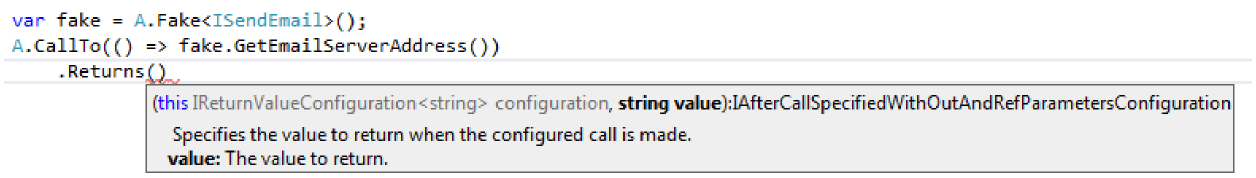
```
var fake = A.Fake<ISendEmail>();
A.CallTo(() => fake.)
```

Figure 18: Members available from the fake

We'll look at **`BodyIsHtml`**, **`GetEmailServerAddress`** and **`SendMail`** methods one at a time and see what IntelliSense is available to us.

Configuring GetEmailServerAddress

When we type **`GetEmailServerAddress`** from out created fake, we're not prompted to provide any arguments, because the method does not take any. When we close off the final parenthesis of **`A.CallTo`**, you'll see a list of other available FakeItEasy calls. For now, pick **`Returns`**. When we pick **`Returns`**, we can see in the IntelliSense that FakeItEasy already knows what type that **`GetEmailServerAddress`** returns and prompts us for a value.



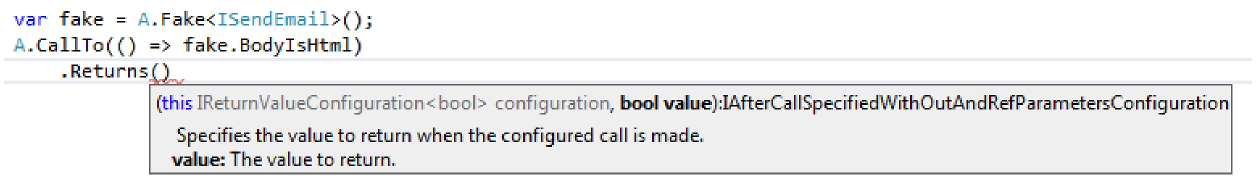
```
var fake = A.Fake<ISendEmail>();
A.CallTo(() => fake.GetEmailServerAddress())
    .Returns()
```

(this IReturnValueConfiguration<string> configuration, **string value**):IAfterCallSpecifiedWithOutAndRefParametersConfiguration
Specifies the value to return when the configured call is made.
value: The value to return.

Figure 19: Configuring GetEmailServerAddress

Configuring BodyIsHtml

For configuring BodyIsHtml, the IntelliSense looks the same.



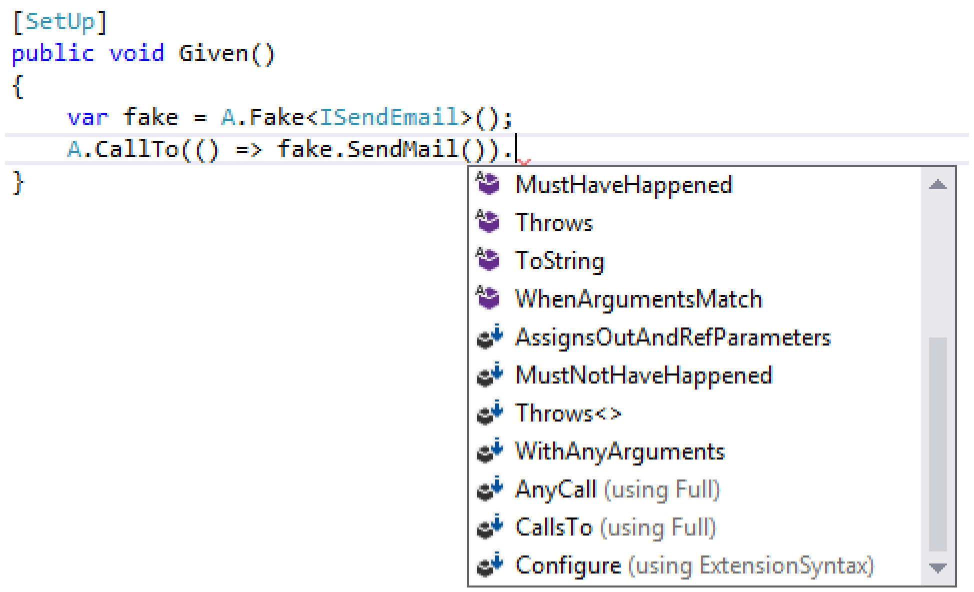
```
var fake = A.Fake<ISendEmail>();
A.CallTo(() => fake.BodyIsHtml)
    .Returns()
```

(this IReturnValueConfiguration<bool> configuration, **bool value**):IAfterCallSpecifiedWithOutAndRefParametersConfiguration
Specifies the value to return when the configured call is made.
value: The value to return.

Figure 20: Configuring BodyIsHtml

Configuring SendMail

When we configure **`SendEmail`**, things look different:



```
[SetUp]
public void Given()
{
    var fake = A.Fake<ISendEmail>();
    A.CallTo(() => fake.SendMail()).
}
```

Figure 21: Configuring SendMail

Note the lack of the **`Returns`** call here. FakeItEasy is smart enough to know not to show **`Returns`** in the available choices because **`SendMail`** does not return anything.

How does FakeItEasy do this? If you look again at **`CallTo`**, you'll see that two of the three overloads take an Expression. FakeItEasy uses this Expression to power the API choices it exposes to you in IntelliSense in your IDE based on the created fake.

Tip: A great introductory resource for learning about Expression Trees is Scott Allen's PluralSite tutorial (http://www.pluralsight.com/courses/linq-fundamentals) on LINQ Fundamentals.

Thankfully, FakeItEasy did all the heavy lifting and wrote the code that is responsible for parsing the expression to give us, the consumer, a fluent API with which to work. All we need to do is to specify the fake, and then, using FakeItEasy's guidance, configure it.

Summary

In this section, we've learned about how to configure a fake, explored **`A.CallTo`**'s overloads, and showed some configurations for different types of members on fake.

In order for the fake to be useful, we want control something about it. We now know how to configure a fake, but we really can’t do much with the configured fake yet—and that’s where the true power of FakeItEasy lies. We’ll learn how to do this in the next chapter, “Specifying a Fake’s Behavior.”

Chapter 6 Specifying a Fake's Behavior

In order for any configured fake to be useful, we're going to want to control something about it. By specifying behavior on the configured fake, we'll be able to test all possible execution paths in our SUT, because we control the dependencies injected into the SUT by using FakeItEasy.

Return Values

Returning values will be one of the most common behaviors you're going to want to control on your fake. Let's explore a couple of commonly used ways to specify return values.

Returns

The **`Returns`** call is probably the most widely used call for specifying behavior on a fake. Let's change our **`ISendEmail`** interface introduced in Chapter 5. To start with, let's just have it define one method, **`GetEmailServerAddress`**:

```
public interface ISendEmail
{
    string GetEmailServerAddress();
}
```

Code Listing 29: The ISendEmail interface

Here is how to specify the return value of **`GetEmailServerAddress`**:

```
A.CallTo(() => emailSender.GetEmailServerAddress()).Returns("SMTPServerAddress");
```

Code Listing 30: Returning a hard-coded value

Hard-coded values work fine, as well as any variables you've setup in your unit test:

```
A.CallTo(() => emailSender.GetEmailServerAddress()).Returns(smtpServerAddress);
```

Code Listing 31: Returning a variable set up in your unit test

`Returns` can also work with collections. Let's add a **`GetAllCcRecipients`** method to our **`ISendEmail`** interface that returns a **`List<string>`** that represents any email addresses that we wanted to send an email to in our **`ISendEmail`** interface:

```
public interface ISendEmail
{
    string GetEmailServerAddress();
    List<string> GetAllCcRecipients();
}
```

Code Listing 32: Adding GetAllCcRecipients to the ISendEmail interface

Here is the **Returns** call:

```
A.CallTo(() => emailSender.GetAllCcRecipients()).Returns(new List<string> {
    "CcRecipient1@somewhere.com", "CcRecipient2@somewhere.com" });
```

Code Listing 33: The Returns call for testing GetAllCcRecipients

We're using hard-coded values here, but we could easily plug in a variable that is declared and populated in the test setup. **Returns** looks at the return type of the configured call, and forces you to provide something that matches the type.

Let's take a look into a couple more useful ways to deal with return values next.

ReturnsNextFromSequence

Sometimes return values that are determined at runtime are not known at design time when using FakeItEasy. A fake of a type that implements IEnumerable would be an example. Another example would be calling into the same instance of a fake multiple times and getting a different result back each time in the same calling context for the SUT.

For situations like this, there is **ReturnsNextFromSequence**. For a more concrete look at this functionality, let's again explore this call through an example. We'll be using three separate interfaces for this sample.

We're going to introduce an interface called **IProvideNewGuids**:

```
public interface IProvideNewGuids
{
    Guid GenerateNewId();
}
```

Code Listing 34: An abstraction to generating new IDs that are GUIDs

The role of this abstraction is very straightforward—it abstracts the creation of new globally unique identifiers (GUIDs) away from code that needs the functionality. This reason could be different ways to generate GUIDs based on environment, usage, etc; but again, this is sample code, so if this seems like an odd thing to abstract, bear with me until the end of the sample.

The next interface is **ICustomerRepository**:

```
public interface ICustomerRepository
{
    List<Customer> GetAllCustomers();
}
```

Code Listing 35: The ICustomerRepository interface

ICustomerRepository simply allows us to get a list of customers. The final interface, **ISendEmail**, allows us to send an email.

```
public interface ISendEmail
{
    void SendMail(Email email);
}
```

Code Listing 36: The ISendEmail interface

Note that **`SendMail`** takes an **`Email`** object. Let's look at **`Email`**:

```
public class Email
{
    public Guid Id { get; set; }
    public string To { get; set; }
}
```

Code Listing 37: The Email class

Email takes an ID that is a GUID, and a string that represents a "To" address. For the sake of the example, let's say that a requirement of the business is to be able to uniquely identify every single email sent in the system, even for the same customer. That being the case, the **`Id`** field takes a GUID, which we'll populate whenever we send an email using **`ISendEmail`**.

Now that we have an idea of the dependencies involved in this example, as well as a business requirement, here is a **`CustomerService`** class that shows them in use:

```
public class CustomerService
{
    private readonly ISendEmail emailSender;
    private readonly ICustomerRepository customerRepository;
    private readonly IProvideNewGuids guidProvider;

    public CustomerService(ISendEmail emailSender, ICustomerRepository
        customerRepository, IProvideNewGuids guidProvider)
    {
        this.emailSender = emailSender;
        this.customerRepository = customerRepository;
        this.guidProvider = guidProvider;
    }

    public void SendEmailToAllCustomers()
    {
        var customers = customerRepository.GetAllCustomers();
        foreach (var customer in customers)
        {
            var email = new Email { Id = guidProvider.GenerateNewId(),
                To = customer.Email };
            emailSender.SendMail(email);
        }
    }
}
```

Code Listing 38: The CustomerService class using all three dependencies

When we invoke **SendEmailToAllCustomers**, the method will reach out to the **GetAllCustomers** method on **ICustomerRepository**, and for each customer returned, create a new **Email** object using our **IProvideNewGuids** provider to populate the **Id** field. The code then uses the final dependency, **ISendEmail**, to send the email by calling **SendMail** and passing the **Email** object to it.

At the beginning of this section, I mentioned we can use **ReturnsNextFromSequence** to test values that are only available at runtime. Looking back over the sample code I just provided, can you find the value that will only be available at runtime? If you answered "the GUID being generated by the **IProvideNewGuids** dependency," then you are correct.

Now that we've identified the values that we'll have to deal with at run time, here is the unit test for the **CustomerService** class:

```
[TestFixture]
public class WhenSendingEmailToAllCustomers
{
    private readonly Guid guid1 = Guid.NewGuid();
    private readonly Guid guid2 = Guid.NewGuid();

    [SetUp]
    public void Given()
    {
        var customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => customerRepository.GetAllCustomers()).Returns(new List<Customer>
        {
            new Customer { Email = "customer1email@somewhere.com" },
            new Customer { Email = "customer2email@somewhere.com" }
        });

        var guidProvider = A.Fake<IProvideNewGuids>();
        A.CallTo(() => guidProvider.GenerateNewId())
            .ReturnsNextFromSequence(guid1, guid2);

        var emailSender = A.Fake<ISendEmail>();

        var sut = new CustomerService(emailSender, customerRepository, guidProvider);
        sut.SendEmailToAllCustomers();
    }
}
```

Code Listing 39: Using ReturnsNextFromSequence

We provide the two GUIDs upfront in our test setup, then use those values in the **ReturnsNextFromSequence** call to the faked **IProvideNewGuids** dependency. When the **CustomerService** class loops through each customer returned by **ICustomerRepository**, a new **Email** will be created with each GUID in sequence. That **Email** will then be sent via **ISendEmail** to the correct customer email address.

Note: So far, we've only covered using FakeItEasy in test setup. You might be wondering where the assertions are for this unit test. Right now, since this chapter is focusing on specifying a fake's behavior, we're focusing on that in our test setup; we're not ready to move into assertions yet. We'll learn about Assertions using FakeItEasy in Chapter 7.

ReturnsLazily

`ReturnsLazily` is a very powerful method you can use to configure a fake's behavior when **`ReturnsNextFromSequence`** won't provide you with what you need. **`ReturnsLazily`** comes in handy when you're trying to test code that returns different objects in the same call to the SUT. Let's explore an example.

Let's say we want to get a list of customer names as a CSV string given this interface definition:

```
public interface ICustomerRepository
{
    Customer GetCustomerById(int id);
}
```

Code Listing 40: The ICustomerRepository interface

And this customer class:

```
public class Customer
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Code Listing 41: The Customer class

Here is the implementation of a **`CustomerService`** class that shows how to do it:

```
public class CustomerService
{
    private readonly ICustomerRepository customerRepository;

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

    public string GetCustomerNamesAsCsv(int[] customerIds)
    {
        var customers = new StringBuilder();
        foreach (var customerId in customerIds)
        {
            var customer = customerRepository.GetCustomerById(customerId);
            customers.Append(string.Format("{0} {1},",
                customer.FirstName, customer.LastName));
        }
        RemoveTrailingComma(customers);
        return customers.ToString();
    }

    private static void RemoveTrailingComma(StringBuilder stringBuilder)
    {
        stringBuilder.Remove(stringBuilder.Length - 1, 1);
    }
}
```

Code Listing 42: The CustomerService class

In the **`CustomerService`** class, you can see that we're looping through each **`customerId`** supplied to the **`GetCustomerNamesAsCsv`** method and calling to our **`customerRepository`** to get each customer by **`Id`** in the **`foreach`** loop. We're formatting the name to *FirstName LastName*, then using **`stringBuilder`** to append the formatted name into a comma-delimited string.

Note: There is a better way to implement the CustomerService class, which is to call ONCE to the repository for all customers you need, instead of making a separate repository call by for each customer by Id. This would result in fewer read calls to the database. Also, we could easily return the formatted customer's first name and last name using a LINQ projection instead of picking it apart in the GetCustomerNamesAsCsv method using a StringBuilder. Because this code is illustrating how to use ReturnsLazily, we'll keep it as-is for now.

And here is the unit test using **`ReturnsLazily`**:

```
[TestFixture]
public class WhenGettingCustomerNamesAsCsv
{
    private string result;

    [SetUp]
    public void Given()
    {
        var customers = new List<Customer>
        {
            new Customer { Id = 1, FirstName = "FirstName1", LastName = "LastName1" },
            new Customer { Id = 2, FirstName = "FirstName2", LastName = "LastName2" }
        };

        var employeeRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => employeeRepository.GetCustomerById(A<int>.Ignored))
            .ReturnsLazily<Customer, int>(
                id => customers.Single(customer => customer.Id == id));

        var sut = new CustomerService(employeeRepository);
        result = sut.GetCustomerNamesAsCsv(customers.Select(x => x.Id).ToArray());
    }
    [Test]
    public void ReturnsCustomerNamesAsCsv()
    {
        Assert.That(result, Is.EqualTo("FirstName1 LastName1,FirstName2 LastName2"));
    }
}
```

Code Listing 43: The unit test for the CustomerService class using ReturnsLazily

If the call to **ReturnsLazily** is leaving you scratching your head, let's first examine the signature of **ReturnsLazily** that we're using in this unit test.

```
// Summary:
//     Specifies a function used to produce a return value when the configured call
//     is made.  The function will be called each time this call is made and can
//     return different values each time.
//
// Parameters:
//   configuration:
//     The configuration to extend.
//
//   valueProducer:
//     A function that produces the return value.
//
// Type parameters:
//   TReturnType:
//     The type of the return value.
//
//   T1:
//     Type of the first argument of the faked method call.
//
// Returns:
//     A configuration object.
public static IAfterCallSpecifiedWithOutAndRefParametersConfiguration
ReturnsLazily<TReturnType, T1>(this IReturnValueConfiguration<TReturnType>
configuration,
Func<T1, TReturnType> valueProducer);
```

Code Listing 44: The ReturnsLazily overload

Note the **ReturnsLazily<TReturnType, T1>** is the opposite of how .NET **Func** delegate works; **ReturnsLazily** takes the return type (in this case, **TReturnType**) as the FIRST generic type instead of the last generic type. In the method definition, you can see the .NET **Func** delegate type specifying the **TReturnType** being passed in last. In this case, the **TReturnType** is a **Customer**.

Returning to the unit test code, we use **A<int>.Ignored** to configure the behavior on the **GetCustomerById** method, and instead, delegate how return values will be determined at runtime using **ReturnsLazily**.

Looking at our **ReturnsLazily** code:

```
.ReturnsLazily<Customer, int> (id => customers.Single(customer => customer.Id == id));
```

We're returning a customer, and passing an **`int`** to the delegate that will figure out a customer to return for each time **`GetCustomerNamesAsCsv`** is invoked. In this case our delegate is a LINQ query against the **`List<Customer>`** setup in the unit test that will change the ID each time **`GetCustomerNamesAsCsv`** is invoked.

What's happening at runtime is that every time **`GetCustomerNamesAsCsv`** is invoked, we're querying our customer list with the next available ID. Basically, the code is taking the next customer available in the list and using that as the return value from **`GetCustomerNamesAsCsv`**.

As you can see, **`ReturnsLazily`** can provide powerful functionality when testing a SUT's method in which different objects are returned from a single call. Please explore the other overloads for this very useful operator.

Doing Nothing

There are times we want to do nothing. No, not you—the configured fake! Thankfully, FakeItEasy gives this to us out of the box.

First, let's change our ISendEmail interface to:

```
public interface ISendEmail
{
    void SendMail();
}
```

Code Listing 45: The ISendEmail interface

And let's change our CustomerService definition to:

```
public class CustomerService
{
    private readonly ISendEmail emailSender;
    private readonly ICustomerRepository customerRepository;

    public CustomerService(ISendEmail emailSender,
        ICustomerRepository customerRepository)
    {
        this.emailSender = emailSender;
        this.customerRepository = customerRepository;
    }

    public void SendEmailToAllCustomersAsWellAsDoSomethingElse()
    {
        var customers = customerRepository.GetAllCustomers();
        foreach (var customer in customers)
        {
            //although this call is being made, we don't care about the setup, b/c it
doesn't directly affect our results
            emailSender.SendMail();
        }
    }
}
```

Code Listing 46: The CustomerService class

A call to a non-configured fake will result in nothing happening. There are two ways to handle this.

One is to specifically tell FakeItEasy that the fake should do nothing:

```
[TestFixture]
public class ATestWhereWeDontCareAboutISendEmailBySpecifyingDoesNothing
{
    [SetUp]
    public void Given()
    {
        var customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => customerRepository.GetAllCustomers())
            .Returns(new List<Customer> { new Customer() });

        var emailSender = A.Fake<ISendEmail>();
        A.CallTo(() => emailSender.SendMail()).DoesNothing();

        var sut = new CustomerService(emailSender, customerRepository);
    }
}
```

Code Listing 47: Explicitly telling FakeItEasy to do nothing

Here we explicitly tell FakeItEasy we want a call to **`SendEmail`** from the faked **`ISendEmail`** to do nothing. But like I said earlier, we get this behavior by default from FakeItEasy, so we could easily shorten the above unit test setup to the setup directly below.

```
[TestFixture]
public class ATestWhereWeDontCareAboutISendEmail
{
    [SetUp]
    public void Given()
    {
        var customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => customerRepository.GetAllCustomers())
            .Returns(new List<Customer> { new Customer() });
        var sut = new CustomerService(A.Fake<ISendEmail>(), customerRepository);
    }
}
```

Code Listing 48: Pass A.Fake<ISendEmail> directly into CustomerService

Here you can see we're using **`A.Fake<ISendEmail>`** directly as an argument when newing up **`CustomerService`**. We don't have to tell FakeItEasy to **`DoNothing`**, we just pass in a newly created fake and let the behavior default to nothing.

The only time you will have to call **`DoNothing`** explicitly is if you're working with a strict fake, which we'll cover in the next section.

Strict

Simply put, specifying **`Strict`** on any created fakes forces you to configure them. Any calls to unconfigured members throw an exception. Let's look at a quick example.

```
public interface IDoSomething
{
    string DoSomething();
    string DoSomethingElse();
}
```

Code Listing 49: The IDoSomething interface

Let's look at the how to configure a strict fake for **`IDoSomething`**:

```
[TestFixture]
public class ConfigureAStrictFakeOfIDoSomething
{
    [SetUp]
    public void Given()
    {
        var doSomething = A.Fake<IDoSomething>(x => x.Strict());
        A.CallTo(() => doSomething.DoSomething()).Returns("I did it!");

        var sut = new AClassThatNeedsToDoSomething(doSomething);
        var result = sut.DoSomethingElse();
    }
}
```

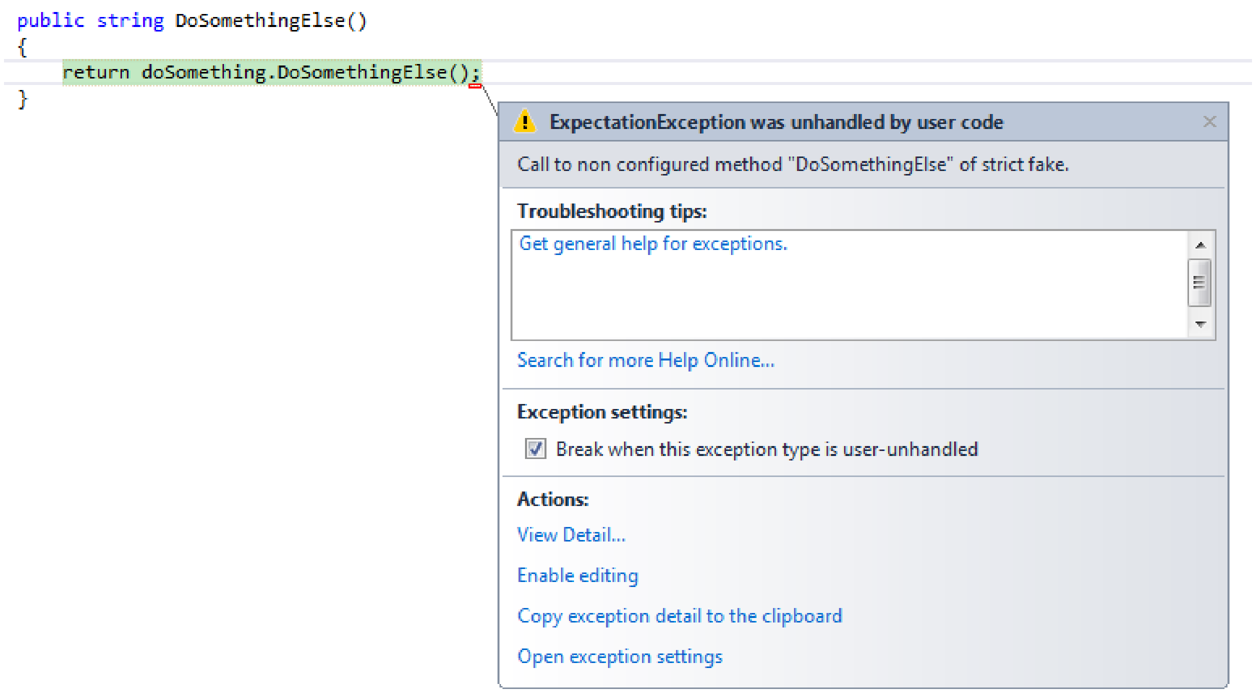
Code Listing 50: Configuring a strict fake of IDoSomething

In Code Listing 50, we create a strict fake of **`IDoSomething`** using **`x => x.Strict()`**. We configure a call to the **`DoSomething`** member on the fake. When we look at the SUT's **`DoSomethingElse`** method, you'll see that we're invoking the **`DoSomethingElse`** method on the fake, which has not been configured in the test setup:

```
public class AClassThatNeedsToDoSomething
{
    private readonly IDoSomething doSomething;

    public AClassThatNeedsToDoSomething(IDoSomething doSomething)
    {
        this.doSomething = doSomething;
    }

    public string DoSomethingElse()
    {
        return doSomething.DoSomethingElse();
    }
}
```

Code Listing 51: The AClassThatNeedsToDoSomething is invoking a non-configured member on a strict fake

When we go to run this unit test, we'll get this exception:



```
public string DoSomethingElse()
{
    return doSomething.DoSomethingElse();
}
```

ExpectationException was unhandled by user code

Call to non configured method "DoSomethingElse" of strict fake.

Troubleshooting tips:

Get general help for exceptions.

Search for more Help Online...

Exception settings:

☑ Break when this exception type is user-unhandled

Actions:

View Detail...

Enable editing

Copy exception detail to the clipboard

Open exception settings

Figure 22: You cannot invoke a non-configured member of a strict fake

Exceptions

Sometimes we want to configure a call to throw an exception. This could be because of execution flow that changes in the SUT, depending on if the call succeeds or fails. Here is how to configure a fake to throw an exception. First, we have some code updates to our examples we've used so far.

Again, we'll use the **`ISendEmail`** interface to start the example, but this time, our **`SendMail`** method will take a list of customers:

```
public interface ISendEmail
{
    void SendMail(List<Customer> customers);
}
```

Code Listing 52: The ISendEmail interface

We'll also use an **`ICustomerRepository`** interface to give us access to a list of all customers:

```
public interface ICustomerRepository
{
    List<Customer> GetAllCustomers();
}
```

Code Listing 53: The CustomerRepository interface

We've added a new class, named **`BadCustomerEmailException`**, that inherits from **`Exception`** to represent a bad customer email send attempt:

```
public class BadCustomerEmailException : Exception {}
```

Code Listing 54: BadCustomerEmailException

The **`CustomerService`** class takes and **`ISendEmail`** dependency that is invoked via the **`SendEmailToAllCustomers`** method.

```
public class CustomerService
{
    private readonly ISendEmail emailSender;
    private readonly ICustomerRepository customerRepository;

    public CustomerService(ISendEmail emailSender, ICustomerRepository
        customerRepository)
    {
        this.emailSender = emailSender;
        this.customerRepository = customerRepository;
    }

    public void SendEmailToAllCustomers()
    {
        var customers = customerRepository.GetAllCustomers();
        try
        {
            emailSender.SendMail(customers);
        }
        catch (BadCustomerEmailException ex)
        {
            //do something here like write to a log file, etc...
        }
    }
}
```

Code Listing 55: Try/catch around the SendMail method of ISendEmail in the CustomerService class

In the **`SendEmailToAllCustomers`** method, we pass all the customers returned by the repository to the email sender. If there is a problem, we catch the **`BadCustomerEmailException`**.

And finally, our unit test:

```
public class WhenSendingEmailToAllCustomersAndThereIsAnException
{
    [SetUp]
    public void Given()
    {
        var customerRepository = A.Fake<ICustomerRepository>();
        var customers = new List<Customer>()
```

```
                { new Customer { EmailAddress = "someone@somewhere.com" } };
            A.CallTo(() => customerRepository.GetAllCustomers()).Returns(customers);

            var emailSender = A.Fake<ISendEmail>();
            A.CallTo(() => emailSender.SendMail(customers))
                .Throws(new BadCustomerEmailException());

            var sut = new CustomerService(emailSender, customerRepository);
            sut.SendEmailToAllCustomers();
        }
}
```

Code Listing 56: The unit test for CustomerService

Note the use of **`Throws`** at the end of the **`A.CallTo`** line. This allows us to throw the exception from our **`ISendEmail`** fake and then write any unit tests that would need to be written for compensating actions.

Note: I've included a comment in the code in the catch block of the CustomerService.SendEmailToAllCustomers method, not an actual implementation. What you need to do here would depend on what you want to do when this exception is caught in your code; write to a log, write to a database, take compensating actions, etc.

Out and Ref Parameters

To illustrate how to handle **`Out`** and **`Ref`** parameters using FakeItEasy, let's continue with the current example we've been using so far, using the **`ISendEmail`** and **`ICustomerRepository`** interfaces.

Instead of **`GetAllCustomers`** returning a **`List<Customer>`**, it will now use an **`out`** parameter to return the list:

```
public interface ICustomerRepository
{
    void GetAllCustomers(out List<Customer> customers);
}
```

Code Listing 57: The list of customers is now an out parameter

The **`CustomerService`** class has changed as well. It no longer is using a try/catch block; it is calling **`GetAllCustomers`** from **`ICustomerRepository`**, and for each **`Customer`** returned, we send an email:

```
public class CustomerService
{
    private readonly ISendEmail emailSender;
    private readonly ICustomerRepository customerRepository;

    public CustomerService(ISendEmail emailSender,
        ICustomerRepository customerRepository)
    {
        this.emailSender = emailSender;
        this.customerRepository = customerRepository;
    }

    public void SendEmailToAllCustomers()
    {
        List<Customer> customers;
        customerRepository.GetAllCustomers(out customers);
        foreach (var customer in customers)
        {
            emailSender.SendMail();
        }
    }
}
```

Code Listing 58: The CustomerService class

Here is the updated unit test. Pay special attention to how FakeItEasy deals with the out parameter:

```
[TestFixture]
public class WhenSendingEmailToAllCustomers
{
    [SetUp]
    public void Given()
    {
        var customerRepository = A.Fake<ICustomerRepository>();
        var customers = new List<Customer> { new Customer { EmailAddress =
            "someone@somewhere.com" } };

        A.CallTo(() => customerRepository.GetAllCustomers(out customers))
            .AssignsOutAndRefParameters(customers);

        var sut = new CustomerService(A.Fake<ISendEmail>(), customerRepository);
        sut.SendEmailToAllCustomers();
    }
}
```

Code Listing 59: Using .AssignsOutAndRefParameters to test the out parameter

Here, you can see the use of **`AssignsOutAndRefParameters`** from the **`GetAllCustomers`** call.

Invokes

Sometimes a faked method's desired behavior can't be satisfactorily defined just by specifying return values, throwing exceptions, assigning **`out`** and **`ref`** parameters, or even doing nothing.

`Invokes` allows us to execute custom code when the fake’s method is called.

Let’s take a break from our current samples where we have been using the **`ISendEmail`** abstraction. To illustrate the usage of **`Invokes`**, I’d like to introduce another abstraction we’ll work with, **`IBuildCsv`**:

```
public interface IBuildCsv
{
    void SetHeader(IEnumerable<string> fields);
    void AddRow(IEnumerable<string> fields);
    string Build();
}
```

Code Listing 60: The IBuildCsv interface

This abstraction allows us to set a header for the CSV, add one to **`n`** rows to the CSV, and finally, call **`Build`** to return a CSV file based off the information we’re providing to both **`SetHeader`** and **`AddRow`**.

To put this abstraction to work, let’s first create a simple **`Customer`** class with **`FirstName`** and **`LastName`** properties:

```
public class Customer
{
    public string LastName { get; set; }
    public string FirstName { get; set; }
}
```

Code Listing 61: The Customer class

And to tie it all together, let’s create a **`CustomerService`** class that takes a list of customers, and from that list, returns a CSV file of “*Last name*, *First name*” using the **`IBuildCsv`** abstraction.

```
public class CustomerService
{
    private readonly IBuildCsv buildCsv;

    public CustomerService(IBuildCsv buildCsv)
    {
        this.buildCsv = buildCsv;
    }

    public string GetLastAndFirstNamesAsCsv(List<Customer> customers)
    {
        buildCsv.SetHeader(new[] { "Last Name", "First Name" });
        customers.ForEach(customer => buildCsv.AddRow(
            new [] { customer.LastName, customer.FirstName }));
        return buildCsv.Build();
    }
}
```

Code Listing 62: The CustomerService class

Here you can see the **`buildCsv`** dependency being passed into the **`CustomerService`** class to build a CSV file from the provided list of customers passed to **`GetLastAndFirstNameAsCsv`**.

Now that we have all our items in usage, let's write some unit tests. Let's start with our unit test setup.

```
[TestFixture]
public class WhenGettingCustomersLastAndFirstNamesAsCsv
{
    private readonly string[] headerList = { "Last Name", "First Name" };
    private string[] bodyList;
    private string[] appendedHeaders;
    private string[] appendedRows;

    [SetUp]
    public void Given()
    {
        var buildCsv = A.Fake<IBuildCsv>();
        A.CallTo(() => buildCsv.SetHeader(A<IEnumerable<string>>.Ignored))
            .Invokes(x => appendedHeaders = (string[])x.Arguments.First());
        A.CallTo(() => buildCsv.AddRow(A<IEnumerable<string>>.Ignored))
            .Invokes(x => appendedRows = (string[])x.Arguments.First());

        var customers = new List<Customer> {
            new Customer { LastName = "Doe", FirstName = "Jon"} };
        bodyList = new[] { customers[0].LastName, customers[0].FirstName };

        var sut = new CustomerService(buildCsv);
        sut.GetLastAndFirstNamesAsCsv(customers);
    }
}
```

Code Listing 63: Setup for the CustomerService class unit test using Invokes

Let's talk about this unit test setup a bit. First, we're doing regular setup items, like creating our fake, setting up some test data, creating the SUT, and finally, calling the SUT's method, **`GetLastAndFirstNamesAsCsv`**.

What is different here is the configuration of our fake. Here we see the introduction of **`Invokes`**. **`Invokes`** requires a fair bit of explanation, so bear with me while I walk you through what's happening when we setup our fake's behavior to use **`Invokes`**.

When we use **`Invokes`**, we're "hijacking" the method on our fake to execute custom code. If we hit **F12** while our cursor is over **`Invokes`** in the IDE, we'll see this definition:

```
public interface ICallbackConfiguration<out TInterface> : IHideObjectMembers
{
    TInterface Invokes(Action<FakeItEasy.Core.IFakeObjectCall> action);
}
```

Code Listing 64: The definition of Invokes

Invokes is asking for an **`Action<T>`** where **`T`** is **`FakeItEasy.Core.IFakeObjectCall.`** But what is **`FakeItEasy.Core.IFakeObjectCall`**? If we use F12 to see its definition, this is what we'll see:

```
public interface IFakeObjectCall
{
    ArgumentCollection Arguments { get; }
    object FakedObject { get; }
    MethodInfo Method { get; }
}
```

Code Listing 65: The definition of IFakeObjectCall

Look at all the information available to us about the method of the fake. For brevity, we'll only look at the **`ArgumentCollection Arguments {get;}`** property, because that's what we're using in our test setup. You should explore the other properties available here on your own.

Using F12 to get the definition of ArgumentCollection, you'll see this:

```
[Serializable]
public class ArgumentCollection : IEnumerable<object>, IEnumerable
{
    [DebuggerStepThrough]
    public ArgumentCollection(object[] arguments, IEnumerable<string> argumentNames);
    [DebuggerStepThrough]
    public ArgumentCollection(object[] arguments, MethodInfo method);
    public IEnumerable<string> ArgumentNames { get; }
    public int Count { get; }
    public static ArgumentCollection Empty { get; }
    public object this[int argumentIndex] { get; }
    public T Get<T>(int index);
    public T Get<T>(string argumentName);
    public IEnumerator<object> GetEnumerator();
}
```

Code Listing 66: The definition of ArgumentCollection

Here we see the ability to grab all sorts of information about the argument collection that is sent into the fake's method. Through these methods and properties, our example uses the **`ArgumentCollection`**, and then via LINQ, takes the first of the collection.

Returning to our example, let's break down this single line:
`buildCsv.SetHeader(A<IEnumerable<string>>.Ignored)).Invokes(x => appendedHeaders = (string[])x.Arguments.First();`

We call **`Invokes`**, then using an Action delegate, we assign the first argument of the fake's method to a local variable called **`appendedHeaders`** that we declared in our test method.

So when this code is executed in CustomerService:
`buildCsv.SetHeader(new[] { "Last Name", "First Name" });`

The code we execute will be in our test setup and **`appendedHeaders`** will be populated with the argument value that was provided to **`SetHeader`**. In this case, this value is defined in the **`GetLastAndFirstNamesAsCsv`** method on our SUT.

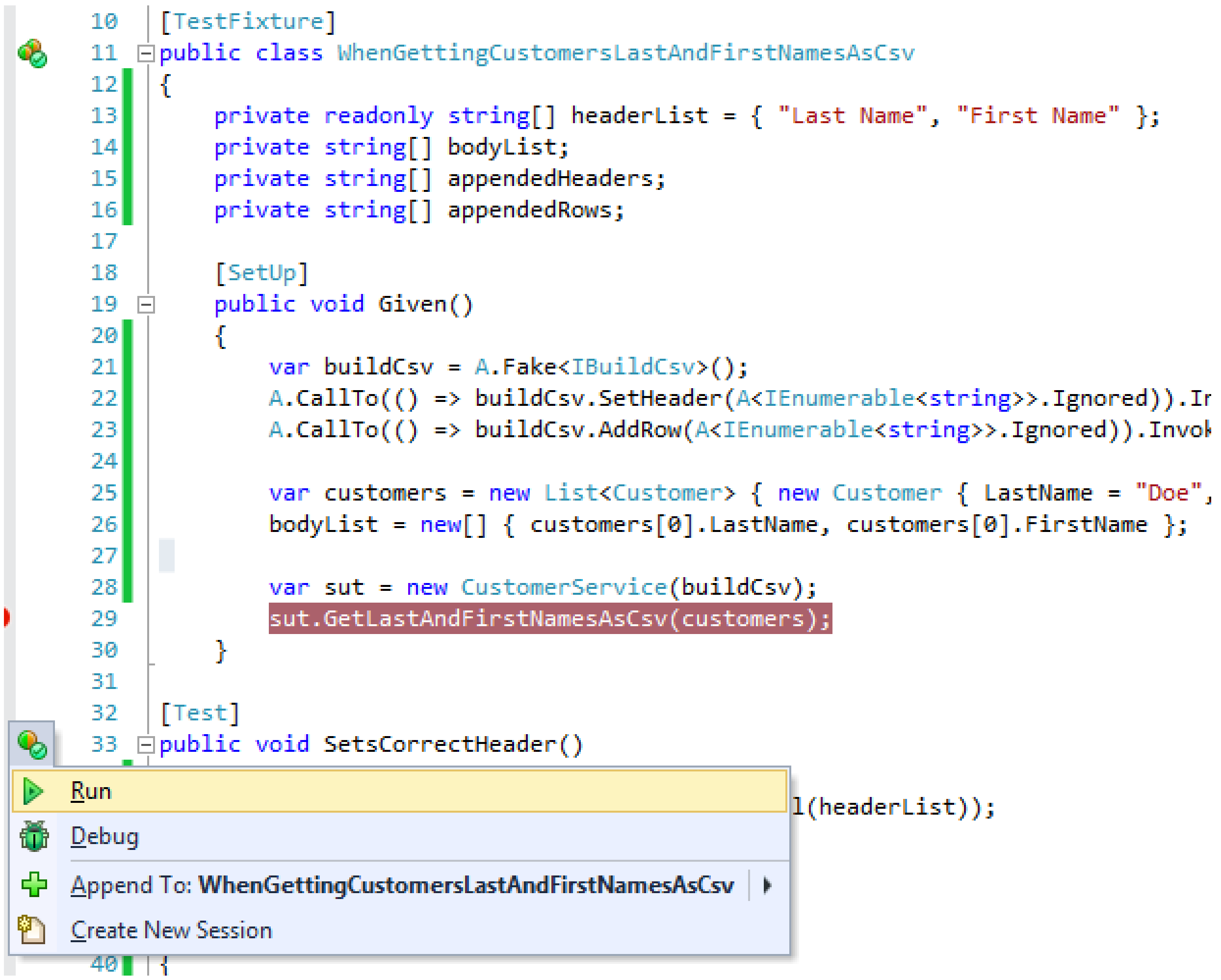
For further illustration, let's finish up our unit test and debug into one of the tests:

```
[Test]
public void SetsCorrectHeader()
{
    Assert.IsTrue(appendedHeaders.SequenceEqual(headerList));
}

[Test]
public void AddsCorrectRows()
{
    Assert.IsTrue(appendedRows.SequenceEqual(bodyList));
}
```

Code Listing 67: The test method for CustomerService

Here you can see that our assertions are very straightforward; we assert that the arrays we're populating in the **`Invokes`** call in our test setup are equal to two lists we set up in our test setup.

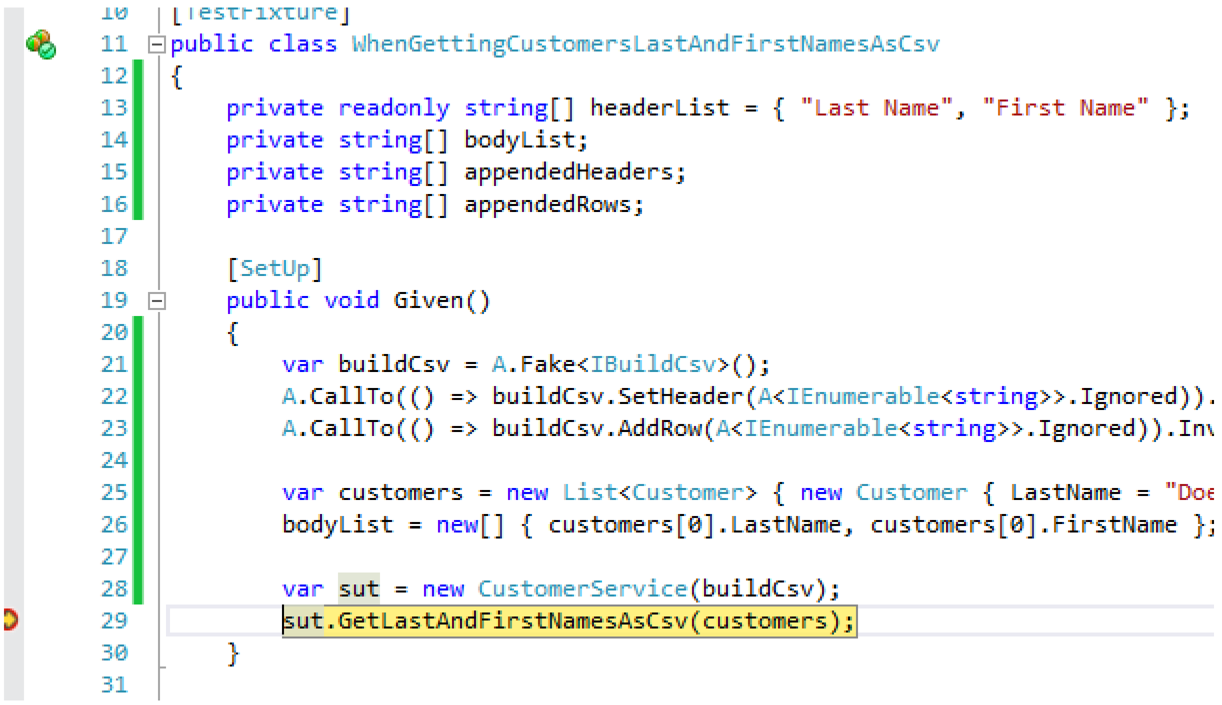
Let's set a breakpoint on the line where our SUT's method is invoked in our test setup, and select **Debug** on the first test method, **`SetsCorrectHeader`**



```
[TestFixture]
public class WhenGettingCustomersLastAndFirstNamesAsCsv
{
    private readonly string[] headerList = { "Last Name", "First Name" };
    private string[] bodyList;
    private string[] appendedHeaders;
    private string[] appendedRows;

    [SetUp]
    public void Given()
    {
        var buildCsv = A.Fake<IBuildCsv>();
        A.CallTo(() => buildCsv.SetHeader(A<IEnumerable<string>>.Ignored)).Ir
        A.CallTo(() => buildCsv.AddRow(A<IEnumerable<string>>.Ignored)).Invok

        var customers = new List<Customer> { new Customer { LastName = "Doe",
        bodyList = new[] { customers[0].LastName, customers[0].FirstName };

        var sut = new CustomerService(buildCsv);
        sut.GetLastAndFirstNamesAsCsv(customers);
    }

[Test]
public void SetsCorrectHeader()
l(headerList));
{
```

Figure 23: Put a breakpoint on sut.GetLastAndFirstNamesAsCsv and then select Debug in the ReSharper menu next to SetsCorrectHeader

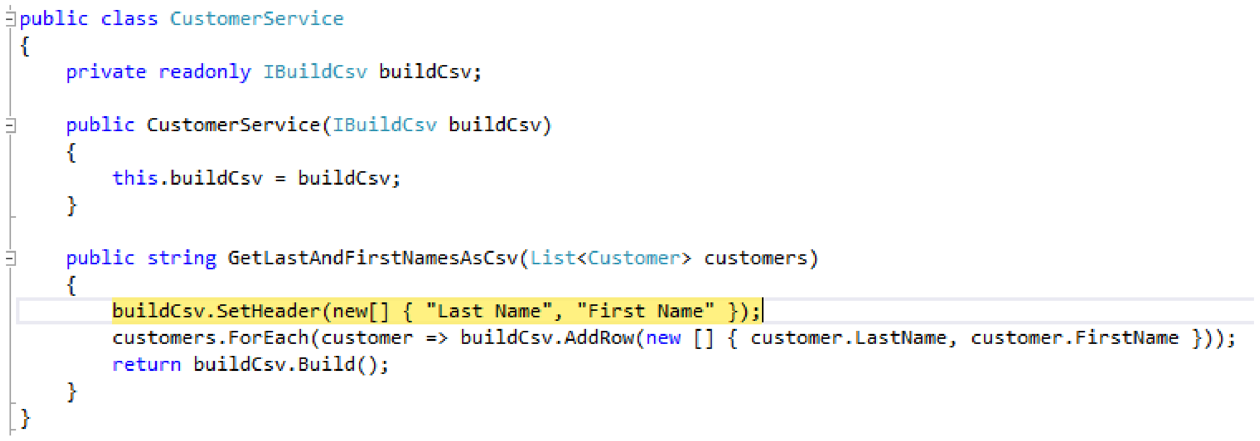
When you hit the breakpoint for **`sut.GetLastAndFirstNamesAsCsv`**, press the **F11** key (step into), and press **F10** once to stop execution on the first line of **`GetLastAndFirstNamesAsCsv`**: **`buildCsv.SetHeader(new[] { "Last Name", "First Name" });`**



```
[TestFixture]
public class WhenGettingCustomersLastAndFirstNamesAsCsv
{
    private readonly string[] headerList = { "Last Name", "First Name" };
    private string[] bodyList;
    private string[] appendedHeaders;
    private string[] appendedRows;

    [SetUp]
    public void Given()
    {
        var buildCsv = A.Fake<IBuildCsv>();
        A.CallTo(() => buildCsv.SetHeader(A<IEnumerable<string>>.Ignored)).
        A.CallTo(() => buildCsv.AddRow(A<IEnumerable<string>>.Ignored)).Inv

        var customers = new List<Customer> { new Customer { LastName = "Doe
        bodyList = new[] { customers[0].LastName, customers[0].FirstName };

        var sut = new CustomerService(buildCsv);
        sut.GetLastAndFirstNamesAsCsv(customers);
    }
```

Figure 24: When you hit this breakpoint, press F11 (step into) to step into the GetLastAndFirstNamesAsCsv method

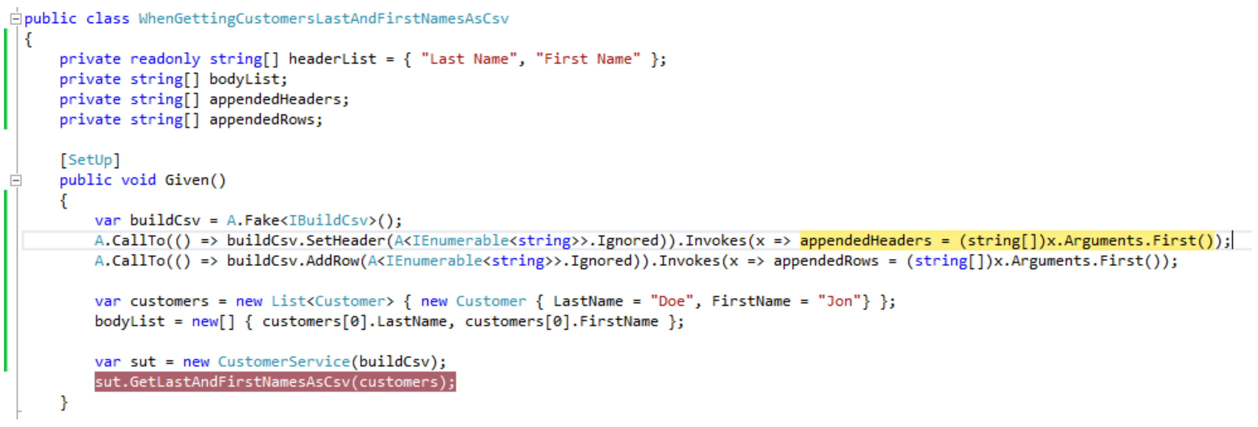
Now that we have our execution path stopped on this line, I want you to press **F11**.



```
public class CustomerService
{
    private readonly IBuildCsv buildCsv;

    public CustomerService(IBuildCsv buildCsv)
    {
        this.buildCsv = buildCsv;
    }

    public string GetLastAndFirstNamesAsCsv(List<Customer> customers)
    {
        buildCsv.SetHeader(new[] { "Last Name", "First Name" });
        customers.ForEach(customer => buildCsv.AddRow(new [] { customer.LastName, customer.FirstName }));
        return buildCsv.Build();
    }
}
```

Figure 25: Execution stopped at the first line of GetLastAndFirstNamesAsCsv

Surprise; we're back in the test setup method where we called Invokes for **`SetHeader`**.



```
public class WhenGettingCustomersLastAndFirstNamesAsCsv
{
    private readonly string[] headerList = { "Last Name", "First Name" };
    private string[] bodyList;
    private string[] appendedHeaders;
    private string[] appendedRows;

    [SetUp]
    public void Given()
    {
        var buildCsv = A.Fake<IBuildCsv>();
        A.CallTo(() => buildCsv.SetHeader(A<IEnumerable<string>>.Ignored)).Invokes(x => appendedHeaders = (string[])x.Arguments.First());
        A.CallTo(() => buildCsv.AddRow(A<IEnumerable<string>>.Ignored)).Invokes(x => appendedRows = (string[])x.Arguments.First());

        var customers = new List<Customer> { new Customer { LastName = "Doe", FirstName = "Jon"} };
        bodyList = new[] { customers[0].LastName, customers[0].FirstName };

        var sut = new CustomerService(buildCsv);
        sut.GetLastAndFirstNamesAsCsv(customers);
    }
```

Figure 26: We're back at our Invokes call in our test setup

What we're seeing here is our custom code being invoked, which adds those header values into the **`appendedHeaders`** variable, which we'll use in our test assertions.

If you'd like to keep following the execution path until this test passes, please go ahead. You'll see how we're bounced back for the call to **`AddRow`** as well. Our action we defined via **`Invokes`** will populate the local variables being used in our test setup so we can run our assertions correctly.

Summary

In this chapter, we have seen many different approaches to specifying a fake's behavior. We started with the most commonly used behavior, Returns. From there, we looked into other scenarios using different forms of Returns, and then ended with how to deal with exceptions and **`out`** and **`ref`** parameters.

Although we've covered a good bit so far, we really still have yet to see the entire toolset of FakeItEasy used. Our unit tests have only covered test setup or very basic assertions, and our fake's members have not dealt with arguments yet.

In order to get most out of FakeItEasy, we not only need to specify behavior, but we also need to deal with arguments to members on those fakes. We'll explore Assertions in the next chapter.

Chapter 7 Assertions

Up until now, we've been concentrating on using FakeItEasy in the setup of our NUnit unit tests. We haven't seen any assertions against any of our setups.

We've all written countless assertions using unit test frameworks. For NUnit, we can assert using **`Assert.That(x, Is.EqualTo(y))`** where **`x`** and **`y`** are of the same type, plus the many other types of assertions available to us via the framework.

But how do we assert that something has happened on a configured fake?

NUnit owns the creation of the SUT, and we use FakeItEasy to provide the configured fakes to the SUT, but how do we assert something has happened or not happened in our unit tests after our setup is complete? FakeItEasy provides two operators for us to do just this: **`MustHaveHappened`** and **`MustNotHaveHappened`**. We'll be exploring both of these methods in this chapter.

MustHaveHappened

`MustHaveHappened` does exactly what it says. Based on fake's setup and configuration, you can then assert that something was called on the fake using **`MustHaveHappened`**.

Basic Usage

Let's stick with our **`ISendEmail`** interface example, and make some changes to it to demonstrate how **`MustHaveHappened`** can be used:

```
public interface ISendEmail
{
    void SendMail();
}
```

Code Listing 68: The ISendEmail interface

We'll also use an **`ICustomerRepository`** interface that has a **`GetAllCustomers`** method on it that returns a list of customers:

```
public interface ICustomerRepository
{
    List<Customer> GetAllCustomers();
}
```

Code Listing 69: The ICustomerRepository interface

Now, we'll add a **CustomerService** class that allows us to send an email to all customers:

```
public class CustomerService
{
    private readonly ISendEmail emailSender;
    private readonly ICustomerRepository customerRepository;

    public CustomerService(ISendEmail emailSender, ICustomerRepository
        customerRepository)
    {
        this.emailSender = emailSender;
        this.customerRepository = customerRepository;
    }

    public void SendEmailToAllCustomers()
    {
        var customers = customerRepository.GetAllCustomers();
        foreach (var customer in customers)
        {
            emailSender.SendMail();
        }
    }
}
```

Code Listing 70: The CustomerService class

As you can see from the class, we're looping through the customers returned from **GetAllCustomers**, then calling **SendMail** for each customer. So far, nothing too complicated. We've already seen a couple examples like this when configuring calls to a fake and specifying a fake's behavior. But this time, we're going to assert that the **SendMail** method was called in our unit tests assertion. Here is the unit test code:

```
[TestFixture]
public class WhenSendingEmailToAllCustomers
{
    private ISendEmail emailSender;

    [SetUp]
    public void Given()
    {
        emailSender = A.Fake<ISendEmail>();

        var customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => customerRepository.GetAllCustomers())
            .Returns(new List<Customer> { new Customer() });

        var sut = new CustomerService(emailSender, customerRepository);
        sut.SendEmailToAllCustomers();
    }

    [Test]
    public void SendsEmail()
    {
        A.CallTo(() => emailSender.SendMail()).MustHaveHappened();
    }
}
```

Code Listing 71: Asserting that a call to the fake's SendMail method must have happened

Looking at the unit test above, you can see in the **`SendsEmail`** test method that we are asserting that a call was made to the **`SendMail`** method on the fake using **`MustHaveHappened`**.

Repeated

In looking at our unit test for **`CustomerService`** class again, it looks pretty simple. But we're only asserting that the call happened. What if we want to assert that a configured fake was called more than once, or a certain number of times? We can do this by passing a FakeItEasy abstract class called **`Repeated`** to the **`MustHaveHappened`** method.

Here is the **`Repeated`** abstract class:

```
public abstract class Repeated
{
    protected Repeated();
    public static IRepeatSpecification AtLeast { get; }
    public static IRepeatSpecification Exactly { get; }
    public static Repeated Never { get; }
    public static IRepeatSpecification NoMoreThan { get; }
    public static Repeated Like(Expression<Func<int, bool>> repeatValidation);
}
```

Code Listing 72: The FakeItEasy Repeated abstract class

There is a good amount of functionality here. Let's look at some examples to clarify.

Let's change our unit test to demonstrate how to use **`MustHaveHappened`** with **`Repeated`**. For this example, the test setup remains the same; only the test method's assertion changes:

```
[Test]
public void SendsEmail()
{
    A.CallTo(() => emailSender.SendMail()).MustHaveHappened(Repeated.Exactly.Once);
}
```

Code Listing 73: Testing that the SendMail was called once using Repeated

Note how we added **`Repeated.Exactly.Once`** as an argument to **`MustHaveHappened`**. **`Repeated`** is very useful here, as we want to make sure we're not sending the same email more than once to the same customer. Where **`Repeated`** also helps is in asserting that calls to databases are made once, or, when performing inserts or updates in a loop, asserting that the call to the database was made a certain number of times.

Speaking of asserting that a call was made more than once, let's change our unit test to reflect that. In our unit test setup, let's change the customer repository fake to return two customers instead of one:

```
A.CallTo(() => customerRepository.GetAllCustomers())
```

```
    .Returns(new List<Customer> { new Customer(), new Customer() });
```

Code Listing 74: Returning two customers instead of one

We'll also update our unit test to assert that call to **`SendMail`** was made twice:

```
A.CallTo(() => emailSender.SendMail())
   .MustHaveHappened(Repeated.Exactly.Twice);
```

Code Listing 75: Assertion using Repeated.Exactly.Twice

This is a much more specific assertion, as it's driven by the number of customers we're configuring to be returned from the **`GetAllCustomers`** method from our customer repository fake. And if it's one thing we're after in unit testing, it's to test all possible scenarios as specifically as possible.

We can improve this test by letting the data we use for configuration and behavior in the test setup drive the assertion. Here is the entire updated test class:

```
[TestFixture]
public class WhenSendingEmailToTwoCustomers
{
    private ISendEmail emailSender;
    private List<Customer> customers;

    [SetUp]
    public void Given()
    {
        emailSender = A.Fake<ISendEmail>();
        customers = new List<Customer> { new Customer(), new Customer() };

        var customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => customerRepository.GetAllCustomers()).Returns(customers);

        var sut = new CustomerService(emailSender, customerRepository);
        sut.SendEmailToAllCustomers();
    }

    [Test]
    public void SendsTwoEmails()
    {
        A.CallTo(() => emailSender.SendMail())
            .MustHaveHappened(Repeated.Exactly.Times(customers.Count));
    }
}
```

Code Listing 76: Using Repeated.Exactly.Times() with customers.Count

We've changed two things here:

- We've changed customers from a local declaration using **`var`** to a field declaration so our test method can have access to it.
- We're using **`Repeated.Exactly.Times`**. **`Times`** takes an integer for a specific amount of repeats, unlike **`Exactly.Once`** or **`Exactly.Twice`**. By passing **`Times`**, a **`customers.Count`**, we've allowed our test setup to drive our test assertion.

You should explore the other options that the **Repeated** class gives you when working with assertions with FakeItEasy.

MustNotHaveHappened

Just as we can assert that a call to a fake happened, and happened a specific number of times, we can also assert that a call to a fake did not happen. In this case, we are testing the "non-happy" path in our example. We want to cover all scenarios when we write our unit tests.

In the previous section on **MustHaveHappened**, we tested the "happy" path of an email being sent to each customer returned from the **CustomerRepository**. Let's test the path of an email **not** being sent because there are no customers returned from the **CustomerRepository**.

The implementation of the **CustomerService** class remains the same, but our test's setup and assertion changes:

```
[TestFixture]
public class WhenSendingEmailToAllCustomersAndNoCustomersExist
{
    private ISendEmail emailSender;

    [SetUp]
    public void Given()
    {
        emailSender = A.Fake<ISendEmail>();

        var customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo((() => customerRepository.GetAllCustomers()))
            .Returns(new List<Customer>());

        var sut = new CustomerService(emailSender, customerRepository);
        sut.SendEmailToAllCustomers();
    }

    [Test]
    public void DoesNotSendAnyEmail()
    {
        A.CallTo(() => emailSender.SendMail()).MustNotHaveHappened();
    }
}
```

Code Listing 77: Using MustNotHaveHappened to test a call to the fake email sender did not happen

We've changed two things in our unit test:

- In the test setup, we've configured the fake customer repository to return an empty list of customers.
- **DoesNotSendAnyEmail** now asserts that a call to the fake email sender did not happen by using **MustNotHaveHappened**.

Why did the call not happen? Let's take a look at our **CustomerService** class again:

```
public class CustomerService
{
    private readonly ISendEmail emailSender;
    private readonly ICustomerRepository customerRepository;

    public CustomerService(ISendEmail emailSender,
        ICustomerRepository customerRepository)
    {
        this.emailSender = emailSender;
        this.customerRepository = customerRepository;
    }

    public void SendEmailToAllCustomers()
    {
        var customers = customerRepository.GetAllCustomers();
        foreach (var customer in customers)
        {
            emailSender.SendMail();
        }
    }
}
```

Code Listing 78: The CustomerService class

Since we've configured our customer repository to return an empty list, there are no results to loop through. Because of this, the **`emailSender.SendMail`** code is never invoked.

Summary

In this chapter we've learned how to use assertions with FakeItEasy. We looked at examples of how to use **`MustHaveHappened`** and **`MustNotHaveHappened`**. From there, we looked at **`Repeated`**, and then refactored a unit test to allow the FakeItEasy test setup to power the test assertions. Now that we know how to use FakeItEasy assertions, let's tackle how to use FakeItEasy with testing methods that take arguments.

Chapter 8 Arguments

So far we've seen how to create a fake, how to configure it, how to specify behavior, and how to use FakeItEasy for assertions. Through all the samples so far, we've been using an **`ISendEmail`** interface that exposes members with no arguments.

```
public interface ISendEmail
{
    void SendMail();
}
```

Code Listing 79: The ISendEmail interface

In the real world, calling a **`SendMail`** method that takes no arguments is really not that useful. We know we want to send email, but to send email, you need information like a "from" address, a "to" address, and subject and body, at the minimum.

In this chapter, we'll be exploring how to pass and constrain arguments to faked members, as well as looking how these constraints can be used in our assertions on our fakes.

Passing Arguments to Methods

Let's start by adding some arguments to our **`SendMail`** member on our **`ISendEmail`** interface:

```
public interface ISendEmail
{
    void SendMail(string from, string to, string subject, string body);
}
```

Code Listing 80: The new ISendEmail interface that takes arguments

Let's also add an **`Email`** member to our **`Customer`** class:

```
public class Customer
{
    public string Email { get; set; }
}
```

Code Listing 81: The Customer class with an Email property

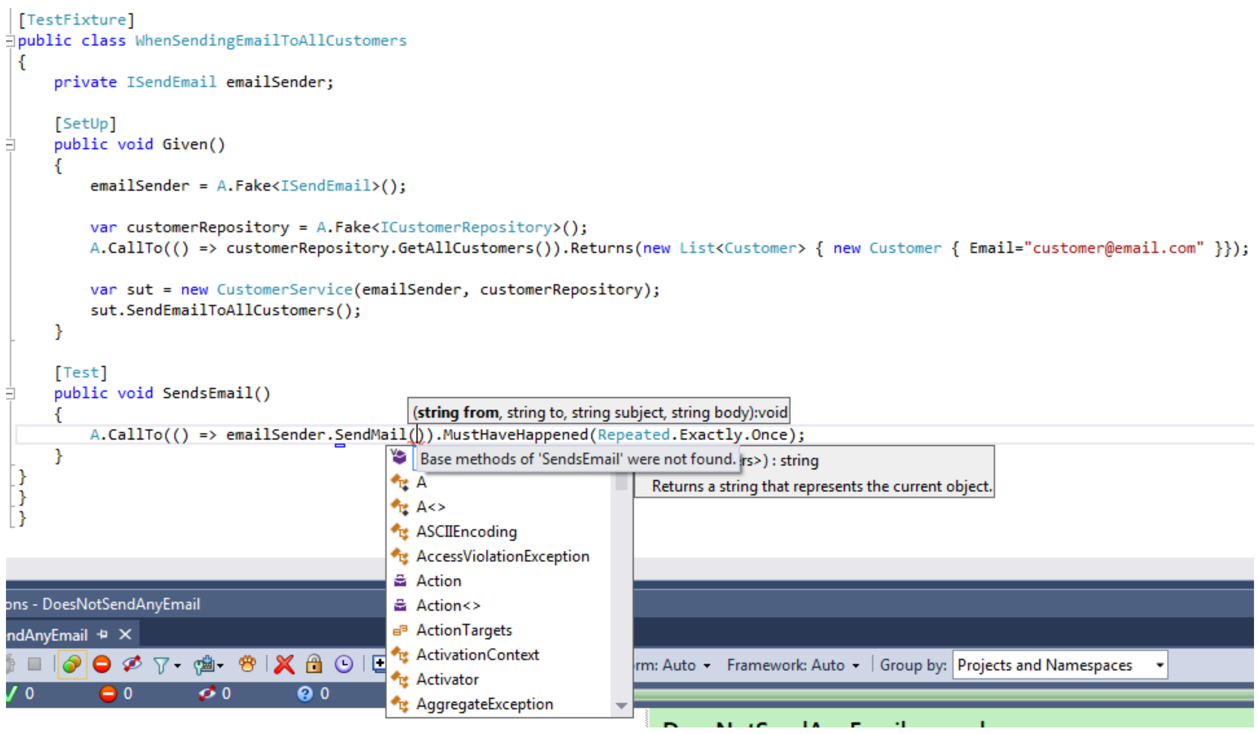
Returning to our previous examples of a **`CustomerService`** class, let's take a look at how the changed **`ISendEmail`** interface and **`Customer`** class look when being used in code:

```
public class CustomerService
{
    private readonly ISendEmail emailSender;
    private readonly ICustomerRepository customerRepository;

    public CustomerService(ISendEmail emailSender,
        ICustomerRepository customerRepository)
    {
        this.emailSender = emailSender;
        this.customerRepository = customerRepository;
    }

    public void SendEmailToAllCustomers()
    {
        var customers = customerRepository.GetAllCustomers();
        foreach (var customer in customers)
        {
            emailSender.SendMail("acompany@somewhere.com", customer.Email,
                 "subject", "body");
        }
    }
}
```

Code Listing 82: The CustomerService class

When we begin looping through our returned customers, for each customer, we're passing arguments to **`SendMail`**. Three arguments are hard-coded, and one is using the **`Email`** member on the **`Customer`** class.

Let's write a unit test that asserts that **`SendMail`** is invoked for each customer returned from **`ICustomerRepository`**.



```
[TestFixture]
public class WhenSendingEmailToAllCustomers
{
    private ISendEmail emailSender;

    [SetUp]
    public void Given()
    {
        emailSender = A.Fake<ISendEmail>();

        var customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => customerRepository.GetAllCustomers()).Returns(new List<Customer> { new Customer { Email="customer@email.com" }});

        var sut = new CustomerService(emailSender, customerRepository);
        sut.SendEmailToAllCustomers();
    }

    [Test]
    public void SendsEmail()
    {
        A.CallTo(() => emailSender.SendMail()).MustHaveHappened(Repeated.Exactly.Once);
    }
}
}
}
```

(string from, string to, string subject, string body):void

Base methods of 'SendsEmail' were not found.

rs>) : string

Returns a string that represents the current object.

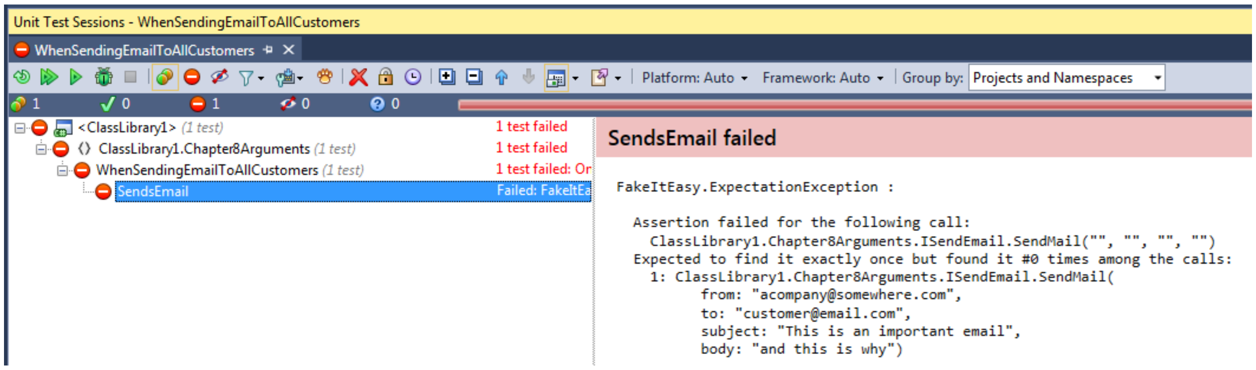
Figure 27: Asserting a call to SendMail happens when using SendMail with arguments

As you can see from Figure 27, when we go to write our assertion against **`SendMail`**, we're prompted for the arguments that **`SendMail`** requires. What do we put here? For now, since we're just trying to assert that the call happened a specified number of times, let's put in **`string.Empty`** for each argument.

```
[TestFixture]
public class WhenSendingEmailToAllCustomers
{
    private ISendEmail emailSender;

    [SetUp]
    public void Given()
    {
        emailSender = A.Fake<ISendEmail>();

        var customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => customerRepository.GetAllCustomers()).Returns(
            new List<Customer> { new Customer { Email="customer@email.com" }});

        var sut = new CustomerService(emailSender, customerRepository);
        sut.SendEmailToAllCustomers();
    }

    [Test]
    public void SendsEmail()
    {
        A.CallTo(() => emailSender
            .SendMail(string.Empty, string.Empty, string.Empty, string.Empty))
                .MustHaveHappened(Repeated.Exactly.Once);
    }
}
```

Code Listing 83: Passing string.Empty in for each argument to SendMail

Now we have a compiling test. But when we go to run this test, it fails:



Unit Test Sessions - WhenSendingEmailToAllCustomers

```
<ClassLibrary1> (1 test)                                    1 test failed
  ClassLibrary1.Chapter8Arguments (1 test)                  1 test failed
    WhenSendingEmailToAllCustomers (1 test)                 1 test failed: Or
      SendsEmail                                            Failed: FakeItEa
```

SendsEmail failed

```
FakeItEasy.ExpectationException :

  Assertion failed for the following call:
    ClassLibrary1.Chapter8Arguments.ISendEmail.SendMail("", "", "", "")
  Expected to find it exactly once but found it #0 times among the calls:
    1: ClassLibrary1.Chapter8Arguments.ISendEmail.SendMail(
          from: "acompany@somewhere.com",
          to: "customer@email.com",
          subject: "This is an important email",
          body: "and this is why")
```

WhenSendingEmailToAllCustomers
Platform: Auto
Framework: Auto
Group by: Projects and Namespaces
1
0
1
0
0

Figure 28: SendsEmail failed

Why? FakeItEasy is now expecting certain values for each call to **`SendMail`**. In our assertion, we're passing all **`string.Empty`** values, which allows the test to compile, but fails when we run it because **`SendMail`** is being invoked with arguments that are not equal to all **`string.Empty`** values.

Knowing this information, let's re-write the test method to pass in the correct arguments:

```
[TestFixture]
public class WhenSendingEmailToAllCustomers
{
    private ISendEmail emailSender;
    private Customer customer;

    [SetUp]
    public void Given()
    {
        emailSender = A.Fake<ISendEmail>();
        customer = new Customer { Email = "customer@email.com" };

        var customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => customerRepository.GetAllCustomers())
            .Returns(new List<Customer> { customer });

        var sut = new CustomerService(emailSender, customerRepository);
        sut.SendEmailToAllCustomers();
    }

    [Test]
    public void SendsEmail()
    {
        A.CallTo(() => emailSender
            .SendMail("acompany@somewhere.com", customer.Email, "subject", "body"))
                .MustHaveHappened(Repeated.Exactly.Once);
    }
}
```

Code Listing 84: Correcting the arguments for the SendMail call

When we run this test, it passes. Note how we took the three hard-coded values from the **`CustomerService`** class in combination with the customer's email address and used those values in our assertion in the **`SendsEmail`** test method. Now we're asserting against the correct argument values as well as the number of calls to the **`SendMail`** method.

A<T>.Ignored

In Code Listing 83, we initially tried to pass four **`string.Empty`** values to our **`SendMail`** method in our assertion code, and quickly learned we could not do that. We needed to pass the correct values for our test to pass.

But sometimes, you don't care about the values of arguments passed to a fake. A good example of this would be a scenario we already covered earlier in the book in Chapter 7 on Assertions, where we were asserting that a call to **`SendMail`** did NOT happen using **`MustNotHaveHappened`**. This is where **`A<T>.Ignored`** comes in handy.

Let's return to that example, asserting **`SendMail`** was not called in order to demonstrate how to use **`A<T>.Ignored`**. In this case, our **`CustomerService`** class does not change from Code Listing 78, but our unit test will.

Here is the unit test for testing that a call to **`SendEmail`** did not happen using **`A<T>.Ignored`**:

```
[TestFixture]
public class WhenSendingEmailToCustomersAndNoCustomersExist
{
    private ISendEmail emailSender;

    [SetUp]
    public void Given()
    {
        emailSender = A.Fake<ISendEmail>();
        var sut = new CustomerService(emailSender, A.Fake<ICustomerRepository>());
        sut.SendEmailToAllCustomers();
    }

    [Test]
    public void DoesNotSendEmail()
    {
        A.CallTo(() => emailSender.SendMail(A<string>.Ignored, A<string>.Ignored,
            A<string>.Ignored, A<string>.Ignored)).MustNotHaveHappened();
    }
}
```

Code Listing 85: Using A<T>.Ignored to assert that a call to SendMail did not happen

Since the compiler forces us to provide values to **`SendMail`**, but we don't really care about the values for testing since **`SendMail`** was not called in this unit test, we use **`A<T>.Ignored`** by passing it a string type for **`T`** for each argument. **`A<T>.Ignored`** works with all types, not just primitive types.

Note: Although we could still pass in string.Empty for each argument to SendMail in this test case, and our test will still pass, by using A<T>.Ignored, you're being explicit about the intent of the unit test. Our intent here is to show whoever is reading the code that we truly don't care about the values being passed to SendMail for this test. If we were to use string.Empty in place of A<T>.Ignored, are we expecting empty strings for the unit test to pass, or do we not care about the values of the arguments? This is an important differentiation to make. A<T>.Ignored clears up that question for us.

Tip: you can use A<T>._ as shorthand for A<T>.Ignored

A.Dummy<T>

`A.Dummy<T>` is very similar to **`A<T>.Ignored`**. So what's the difference? **`A<T>.Ignored`** should be used for configuration and assertions for methods on fakes**.** **`A.Dummy<T>`** can be used to pass in default values for **`T`** to a method on a class that has been instantiated using the new keyword. **`A.Dummy<T>`** cannot be used to configure calls. Let's look at a quick example.

We'll again use sending email as an example. Here is the **`ISendEmail`** interface for this example:

```
public interface ISendEmail
{
    Result SendEmail(string from, string to);
}
```

Code Listing 86: The ISendEmail interface

You can see that we're returning a **`Result`** object from the **`SendEmail`** method call. This **`Result`** object will contain a list of potential error messages. This is what the **`Result`** class looks like:

```
public class Result
{
    public Result()
    {
        this.ErrorMessages = new List<string>();
    }

    public List<string> ErrorMessages;
}
```

Code Listing 87: The Result class

In the constructor, we are newing up the list so the consuming code doesn't have to deal with null reference exceptions when there are no errors.

Next, our SUT, the **`CustomerService`** class:

```
public class CustomerService
{
    private readonly ISendEmail emailSender;

    public CustomerService(ISendEmail emailSender)
    {
        this.emailSender = emailSender;
    }
    public Result SendEmail(string from, string to)
    {
        var result = new Result();

        if (string.IsNullOrEmpty(to))
        {
            result.ErrorMessages.Add("Cannot send an email with an empty to address");
            return result;
        }

        emailSender.SendEmail(from, to);

        return result;
    }
}
```

Code Listing 88: The CustomerService class

In the **`SendEmail`** method, you can see that we're doing a **`string.IsNullOrEmpty`** check on the **`to`** argument passed into the method. If it's null or empty, we add a message to the **`ErrorMessages`** list on **`Result`**, and return the result without invoking **`emailSender.SendMail`**. If the **`to`** argument passes the **`string.IsNullOrEmpty`** check, then we invoke the emailSender's **`SendMail`** method and return the result with an empty list.

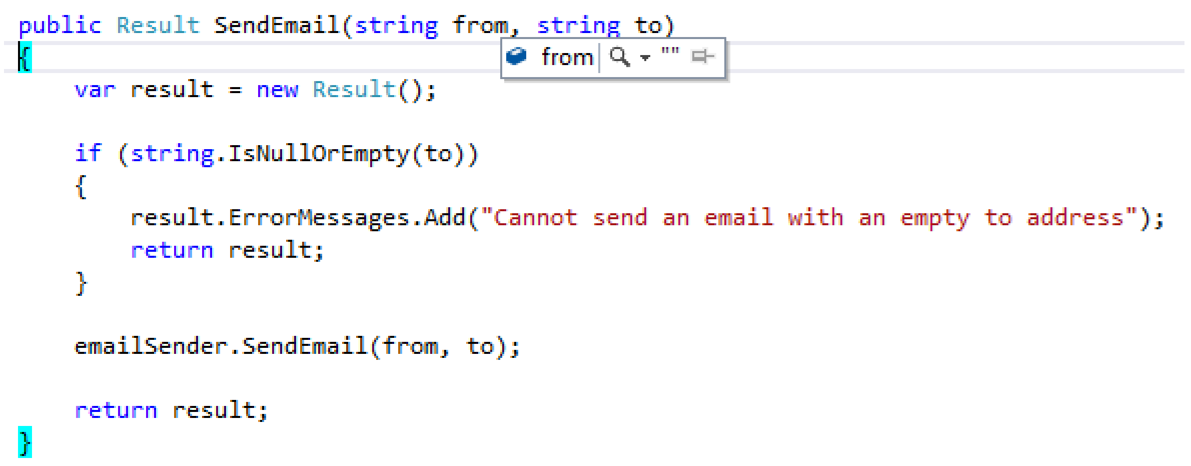
We're going to write a unit test that tests a scenario when the **`to`** argument is **`NullOrEmpty`**. Here is the **`CustomerServiceTests`** use of **`A.Dummy<T>`**.

```
[TestFixture]
public class  WhenSendingAnEmailWithAnEmptyToAddress
{
    private Result result;

    [SetUp]
    public void Given()
    {
        var sut = new CustomerService(A.Fake<ISendEmail>());
        result = sut.SendEmail(A.Dummy<string>(), "");
    }

    [Test]
    public void ReturnsErrorMessage()
    {
        Assert.That(result.ErrorMessages.Single(),
            Is.EqualTo("Cannot send an email with an empty to address"));
    }
}
```

Code Listing 89: CustomerServiceTests using A.Dummy<T>

In this test, you'll see we're passing in **`A.Dummy<string>`** for the **`from`** argument. Because our execution path for this test does not rely on this **`from`** argument value at all, we're using **`A.Dummy<string>`** to represent the **`from`** argument.

If you set a breakpoint on **`result = sut.SendEmail(A.Dummy<string>(), "");`**, debug this unit test and step into **`SendEmail`**, you'll see **`A.Dummy<T>`** passes the string default to **`SendEmail`** for the **`from`** argument:



```
public Result SendEmail(string from, string to)
{
    var result = new Result();

    if (string.IsNullOrEmpty(to))
    {
        result.ErrorMessages.Add("Cannot send an email with an empty to address");
        return result;
    }

    emailSender.SendEmail(from, to);

    return result;
}
```

Figure 29: Using A.Dummy<T> results in the default for T being passed into the method. In this case, it's an empty string

The decision to use **`A.Dummy<T>`** here instead of passing in a **`string.Empty`** value in this unit test is similar to the decision to use **`A<string>.Ignored`** in the assertion in Code Listing 85; using **`A.Dummy<T>`** communicates intent better.

When you see **`A.Dummy<T>`** being used, it's a sign that the value for a given argument is not important for the particular test you're writing. This most likely means the value will not be used in the execution path in the SUT's method as we see in our **`SendEmail`** method on **`CustomerService`**.

Again, you *could* pass **`string.Empty`** for the **`from`** argument and the test would still pass. But does that mean the test needs a **`string.Empty`** value for the **`from`** argument for the test to pass? Does that mean the **`from`** argument's value will be used in the execution path for the test? Without looking at the SUT's code, the test setup using FakeItEasy does not easily answer that question. Using **`A.Dummy<T>`** explicitly says to the person looking at your code, "this value is not used in the execution path, so we don't care about it."

Don't keep other programmers guessing about your intent. Make it clear with **`A.Dummy<T>`**

Constraining Arguments

We've already been constraining arguments in the previous section on Passing Arguments to Methods. That in itself was a way to constrain arguments and then assert that the fake's member was called with the correct values.

We're going to take the next step in constraining arguments by examining the very powerful **`That.Matches`** operator.

That.Matches

Testing a method that takes primitive types when asserting that something must have happened is fairly straightforward—but we're not always so lucky to deal with primitive types when we're working with methods on fakes.

FakeItEasy provides a way to constrain arguments that are passed to methods on fakes. To demonstrate this functionality, it's time to change **`ISendEmail`** again. Instead of taking four strings, **`SendMail`** will now take an **`Email`** object.

```
public interface ISendEmail
{
    void SendMail(Email email);
}
```

Code Listing 90: New ISendMail method signature, introduction of Email class

```
public class Email
{
    public string From { get; set; }
    public string To { get; set; }
    public string Subject { get; set; }
    public string Body { get; set; }
}
```

Code Listing 91: Email class

All we've really done here is encapsulate the four string vales that used to be passed to **`SendMail`** into an **`Email`** class.

Here is the how the change impacts our **`CustomerService`** class:

```
public class CustomerService
{
    private readonly ISendEmail emailSender;
    private readonly ICustomerRepository customerRepository;

    public CustomerService(ISendEmail emailSender,
        ICustomerRepository customerRepository)
    {
        this.emailSender = emailSender;
        this.customerRepository = customerRepository;
    }

    public void SendEmailToAllCustomers()
    {
        var customers = customerRepository.GetAllCustomers();
        foreach (var customer in customers)
        {
            emailSender.SendMail(
                new Email { From = "acompany@somewhere.com", To = customer.Email,
                    Subject = "subject", Body = "body" });
        }
    }
}
```

Code Listing 92: Creating a new Email object and passing in the four values as parameters

Note how we're "newing up" an **`Email`** object and populating the values with the same values that used to be passed to the **`SendMail`** method when it took four strings.

We want to assert that **`SendMail`** was called with the correct customer's email address. This is how we write a unit test for that scenario using **`That.Matches`**:

```
[TestFixture]
public class WhenSendingEmailToAllCustomers
{
    private ISendEmail emailSender;
    private const string customersEmail = "somecustomer@somewhere.com";

    [SetUp]
    public void Given()
    {
        emailSender = A.Fake<ISendEmail>();

        var customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => customerRepository.GetAllCustomers())
            .Returns(new List<Customer> { new Customer { Email = customersEmail }});

        var sut = new CustomerService(emailSender, customerRepository);
        sut.SendEmailToAllCustomers();
    }

    [Test]
    public void SendsEmail()
    {
        A.CallTo(() => emailSender.SendMail(
            A<Email>.That.Matches(email => email.To == customersEmail)))
                .MustHaveHappened(Repeated.Exactly.Once);
    }
}
```

Code Listing 93*: Using .That.Matches to test the CustomerService class*

In our earlier example, we used real strings to test the happy path of **`SendMail`**, and **`A<string>.Ignored`** to test the non-happy path. Now that **`SendMail`** takes an **`Email`** object, you can see how we're using **`That.Matches`** to test the correct values were used on **`Email`** when making the **`SendMail`** call.

`A<Email>.That.Matches(email => email.To == customersEmail)`

`That.Matches` takes an **`Expression<Func<T, bool>>`**, where **`T`** is the type specified in **`A<T>.`** In this example, **`T`** is **`Email`**.

Note: I did not assert that the other properties of Email were correct in the preceding unit test like I did for earlier unit tests. I left them out to make the code sample easier to read, not because they're not important.

Using **`That.Matches`** in a FakeItEasy call is called *custom matching*. This means we need to write our own matcher. Let's explore a couple other pre-written matchers that can provide some common matching scenarios, instead of having to use **`That.Matches`**.

That.IsInstanceOf(T)

`That.IsInstanceOf` can be used for testing a specific type is being passed to a SUT's method. This comes in really handy when you're working with a SUT's method that takes an object or an interface as its type. This method might be given multiple object types within the context of a single test on a SUT's method.

Let's explore the simpler of the two examples: a SUT's method that takes an object as an argument. Since I work with a messaging bus called NServiceBus every day, for this example, we'll introduce a new abstraction called **`IBus`**:

```
public interface IBus
{
    void Send(object message);
}
```

Code Listing 94: The IBus interface

Think of the **`IBus`** interface as an abstraction over a service bus that is responsible for sending "messages" within a system. This **`IBus`** interface allows us to send any type of object.

Now that we have our abstraction set up, here is the "message" we'll be sending:

```
public class CreateCustomer
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
}
```

Code Listing 95: The CreateCustomer message

Think of the message as a simple DTO (http://martinfowler.com/eaaCatalog/dataTransferObject.html) that contains data about the customer we want to create.

Here is the implementation using **`IBus`** and **`CreateCustomer`** in a class called **`CustomerService`**:

```
public class CustomerService
{
    private readonly IBus bus;

    public CustomerService(IBus bus)
    {
        this.bus = bus;
    }

    public void CreateCustomer(string firstName, string lastName, string email)
    {
        bus.Send(new CreateCustomer { FirstName = firstName, LastName = lastName,
            Email = email });
    }
}
```

Code Listing 96: The CustomerService class

The **`CreateCustomer`** method takes three strings about the customer we want to create and uses that information to populate the **`CreateCustomer`** message. That message is then passed to the **`Send`** method on **`IBus`**. In a service-bus based architecture, this “message” would have a corresponding handler somewhere that handles the **`CreateCustomer`** message and does something with it (for example, writing a new customer to the database).

Here is the unit test:

```
[TestFixture]
public class WhenCreatingACustomer
{
    private IBus bus;

    [SetUp]
    public void Given()
    {
        bus = A.Fake<IBus>();
        var sut = new CustomerService(bus);
        sut.CreateCustomer("FirstName", "LastName", "Email");
    }
    [Test]
    public void SendsCreateCustomer()
    {
        A.CallTo(() => bus.Send(
            A<object>.That.IsInstanceOf(typeof(CreateCustomer))))
                .MustHaveHappened(Repeated.Exactly.Once);
    }
}
```

Code Listing 97: Unit test for the CreateCustomer class

What we need to test here is that the type passed to **`bus.Send`** is a type of **`CreateCustomer`** and that the call happened. We do this by using **`IsInstanceOf`** in our assertion.

Although we've successfully tested for the correct message type our unit test, this test is not complete yet. We still would need to test that the values we're passing to the SUT's **`CreateCustomer`** method end up on the **`CreateCustomer`** message.

For more information on how assert each individual property of an object instance is equal to input provided in the test setup, see the "Dealing With Object" section in this chapter.

That.IsSameSequenceAs<T>

Sometimes we want to assert that a sequence is the same when passing a collection to a fake's method. This especially useful when working with common collection types in .NET like **`IEnumerable<T>`** and **`List<T>`**. This is where **`IsSameSequenceAs<T>`** comes in handy.

I'm going to reintroduce the example code from Chapter 6 that showcased **`Invokes`** using the **`IBuildCsv`** interface.

As a reminder, here are the classes and interfaces we dealt with earlier:

```
public interface IBuildCsv
{
    void SetHeader(IEnumerable<string> fields);
    void AddRow(IEnumerable<string> fields);
    string Build();
}
```

Code Listing 98: The IBuildCsv abstraction

```
public class Customer
{
    public string LastName { get; set; }
    public string FirstName { get; set; }
}
```

Code Listing 99: The Customer class

```
public class CustomerService
{
    private readonly IBuildCsv buildCsv;
    public CustomerService(IBuildCsv buildCsv)
    {
        this.buildCsv = buildCsv;
    }

    public string GetLastAndFirstNamesAsCsv(List<Customer> customers)
    {
        buildCsv.SetHeader(new[] { "Last Name", "First Name" });
        customers.ForEach(customer => buildCsv.AddRow(new [] { customer.LastName,
            customer.FirstName }));
        return buildCsv.Build();
    }
}
```

Code Listing 100: The CustomerService class

Our goal here is the same as it was when we were working with **`Invokes`**. We want to make sure the correct header is built and then the correct values are added to each "row" in the CSV based on the customer list passed into the SUT's method.

Let's start by taking a look at the test setup for CustomerService:

```
[TestFixture]
public class WhenGettingCustomersLastAndFirstNamesAsCsv
{
    private IBuildCsv buildCsv;
    private List<Customer> customers;

    [SetUp]
    public void Given()
    {
        buildCsv = A.Fake<IBuildCsv>();
        var sut = new CustomerService(buildCsv);
        customers = new List<Customer>
        {
            new Customer { LastName = "Doe", FirstName = "Jon"},
            new Customer { LastName = "McCarthy", FirstName = "Michael" }
        };

        sut.GetLastAndFirstNamesAsCsv(customers);
    }
}
```

Code Listing 101: Unit test setup for testing CustomerService class

Here we're creating our fake, creating our SUT, and then setting up a list of customers to use. We then pass the list of customers to **`GetLastAndFirstNamesAsCsv`** on the SUT. Note that we're not defining behavior of our fake in the test setup. We're just creating it and passing it to the constructor of the SUT. For this example, we'll be using our fake in assertions only.

Next, let's explore four tests in the test class based on the setup we've done so far.

```
[Test]
public void SetsCorrectHeader()
{
    A.CallTo(() => buildCsv.SetHeader(A<IEnumerable<string>>
        .That.IsSameSequenceAs(new[] { "Last Name", "First Name"})))
            .MustHaveHappened(Repeated.Exactly.Once);
}

[Test]
public void AddsCorrectRows()
{
    A.CallTo(() => buildCsv.AddRow(A<IEnumerable<string>>.That.IsSameSequenceAs
        (new[] { customers[0].LastName, customers[0].FirstName })))
            .MustHaveHappened(Repeated.Exactly.Once);
    A.CallTo(() => buildCsv.AddRow(A<IEnumerable<string>>.That.IsSameSequenceAs(
        new[] { customers[1].LastName, customers[1].FirstName})))
            .MustHaveHappened(Repeated.Exactly.Once);
}

[Test]
public void AddRowsIsCalledForEachCustomer()
{
    A.CallTo(() => buildCsv.AddRow(A<IEnumerable<string>>.Ignored))
        .MustHaveHappened(Repeated.Exactly.Times(customers.Count));
}

[Test]
public void CsvIsBuilt()
{
    A.CallTo(() => buildCsv.Build()).MustHaveHappened(Repeated.Exactly.Once);
}
```

Code Listing 102: The test methods for CustomerService

- **`SetsCorrectHeader`**: In this test, we use **`That.IsSameSequenceAs`** to base the assertion against the hard-coded **`IEnumerable<string>`** that is also present in the **`GetLastAndFirstNamesAsCsv`** method of the SUT. In this case, using the hard-coded values in our unit test is acceptable because the method we're testing on the SUT is also working with a hard-coded **`IEnumerable<string>`**.
- **`AddsCorrectRows`**: We use the same approach that we used for **`SetsCorrectHeader`**, except in this case, we have to assert that the **`AddRow`** method was called with the appropriate customer last name and first name values for each customer in the list by using **`That.IsSameSequenceAs`**. In order to do this, we use the data we set up in the **`customers`** variable and then access that data via index in the test. There is a better way to write this particular unit test that we'll look at after the rest of the test methods overview.
- **`AddRowsIsCalledForEachCustomer`**: This test method does not use **`That.IsSameSequenceAs`**, but I wanted to include it because nowhere are we asserting that the **`AddRow`** method on our fake is being called for the number of customers we've passed to **`GetLastAndFirstNamesAsCsv`**. This test takes care of covering that assertion.
- **`CsvIsBuilt`**: In order to get the CSV string back, we need to assert that this method was called. If not, all we'll be returning from **`GetLastAndFirstNamesAsCsv`** is an empty string.

As promised earlier, let's revisit the **AddsCorrectRows** unit test. I mentioned, there is a better way to write this. Based on the test setup and the assertions that need to be made, there is a way to let the test setup (input) to the SUT drive the assertion. Take a second to look at both the test setup and the unit tests and see if you can figure it out.

Since we're currently using **customers.Count** as part of our assertion in the **AddRowsIsCalledForEachCustomer** test method, we can use this same list of customers to assert that **AddRow** was called for each customer in that list. Here is the updated **AddsCorrectRows** unit test:

```
[Test]
public void AddsCorrectRows()
{
    foreach (var customer in customers)
    {
        A.CallTo(() => buildCsv.AddRow(A<IEnumerable<string>>.That.IsSameSequenceAs
            (new[] { customer.LastName, customer.FirstName })))
                .MustHaveHappened(Repeated.Exactly.Once);
    }
}
```

Code Listing 103: The improved AddsCorrectRows unit test

Here we let the customer list drive the assertion against each **AddRow** call. This eliminates the multiple line call we had earlier, in which we used a hard-coded index against the customer list. By making this change, we've allowed our test input to drive the assertions 100 percent, as well as improved the accuracy and maintainability of our tests.

Dealing with Object

So far, we've been constraining arguments that have all been defined as custom types or primitives. We've examined different methods available to us via the **That** keyword. But what happens when we're dealing with constraining arguments on a class and one of the class's members is of type **object**?

Continuing with our customer example, we're going to change things around a bit and introduce the concept of sending an email for "preferred" customers instead of all customers. We'll add a class that represents the preferred customer email, and we'll assign that to a property that is of type **object** on an **Email** class.

Note: The code I'm about to present is only for the sake of this example. There are better ways to implement what I'm about to write, but if you're like most programmers, you're working in a codebase where change might be very difficult, or in some cases, impossible, which means you could potentially have to write unit tests against an existing codebase you have very little control over. So please keep in mind the code that appears next is not a recommended way to implement this particular piece of functionality.

The **PreferredCustomerEmail** class:

```
public class PreferredCustomerEmail
{
    public string Email { get; set; }
}
```

Code Listing 104: The PreferredCustomerEmail class

The **Email** class:

```
public class Email
{
    public object EmailType { get; set; }
}
```

Code Listing 105: The Email class

Note the **EmailType** property on the **Email** class.

The intention with the **Email** class is to assign a **PreferredCustomerEmail** class instance to the **EmailType** property. For example, there might be a regular **CustomerEmail** class that also could be assigned to this property. Obviously, all this could be accomplished much more nicely by using inheritance, but for the sake of the example, let's continue to move forward with this code.

Let's tie it all together with the **CustomerService** class:

```
public class CustomerService
{
    private readonly ISendEmail emailSender;
    private readonly ICustomerRepository customerRepository;

    public CustomerService(ISendEmail emailSender,
        ICustomerRepository customerRepository)
    {
        this.emailSender = emailSender;
        this.customerRepository = customerRepository;
    }

    public void SendEmailToPreferredCustomers()
    {
        var customers = customerRepository.GetAllCustomers();
        foreach (var customer in customers)
            if (customer.IsPreferred)
                emailSender.SendMail(new Email {
                    EmailType = new PreferredCustomerEmail { Email = customer.Email
}});
    }
}
```

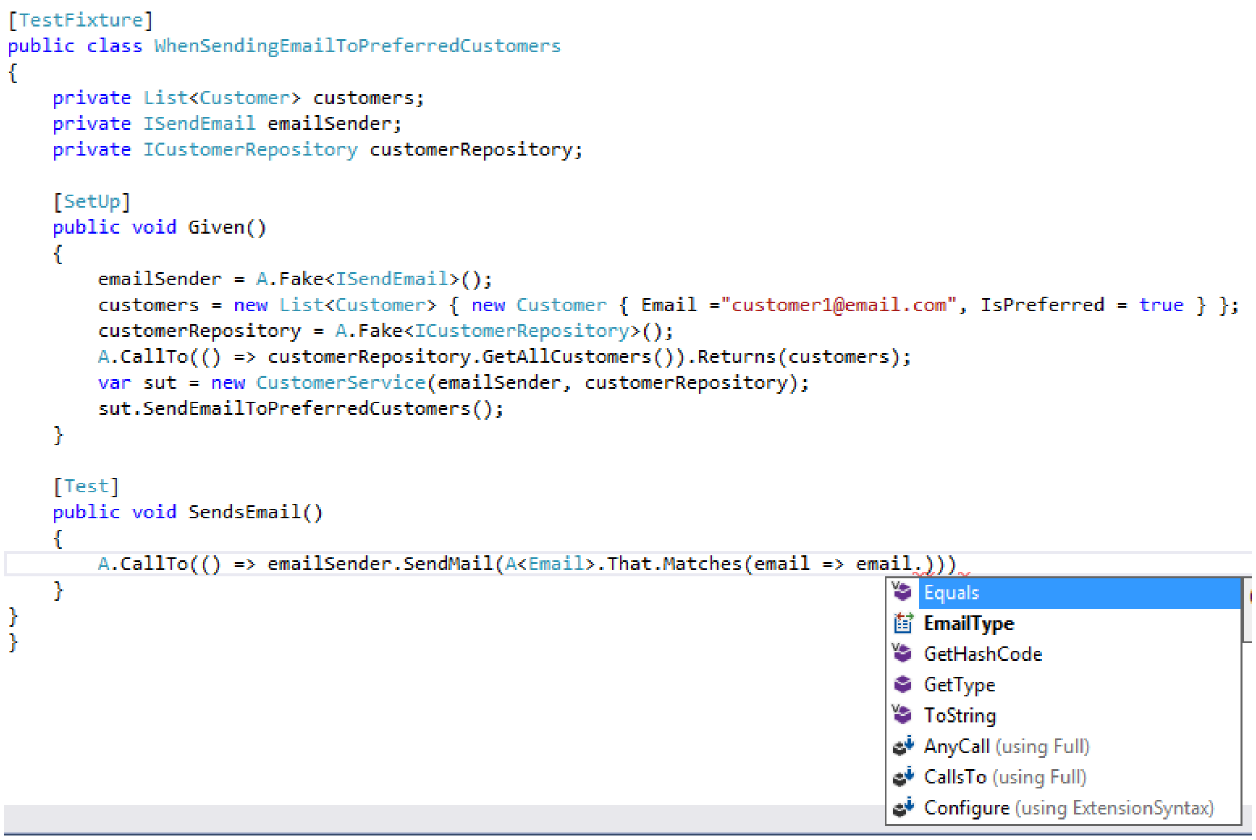
Code Listing 106: The CustomerService class

First, you can see that we're filtering the customers by this conditional statement: **`if (customer.IsPreferred)`**. We're only calling **`SendMail`** for that those customers.

Next, you can see that we're newing up a **`PreferredCustomerEmail`** in the **`SendEmailToPreferredCustomers`** method, and then assigning that instance into the **`EmailType`** property of a newed up **`Email`** instance.

So far, this might not look very different from other tests we've written in the book, but let's write a unit test for this method. What I'm trying to assert is that the email is sent to the correct email address.

Here is a screen grab of the IntelliSense in my IDE:

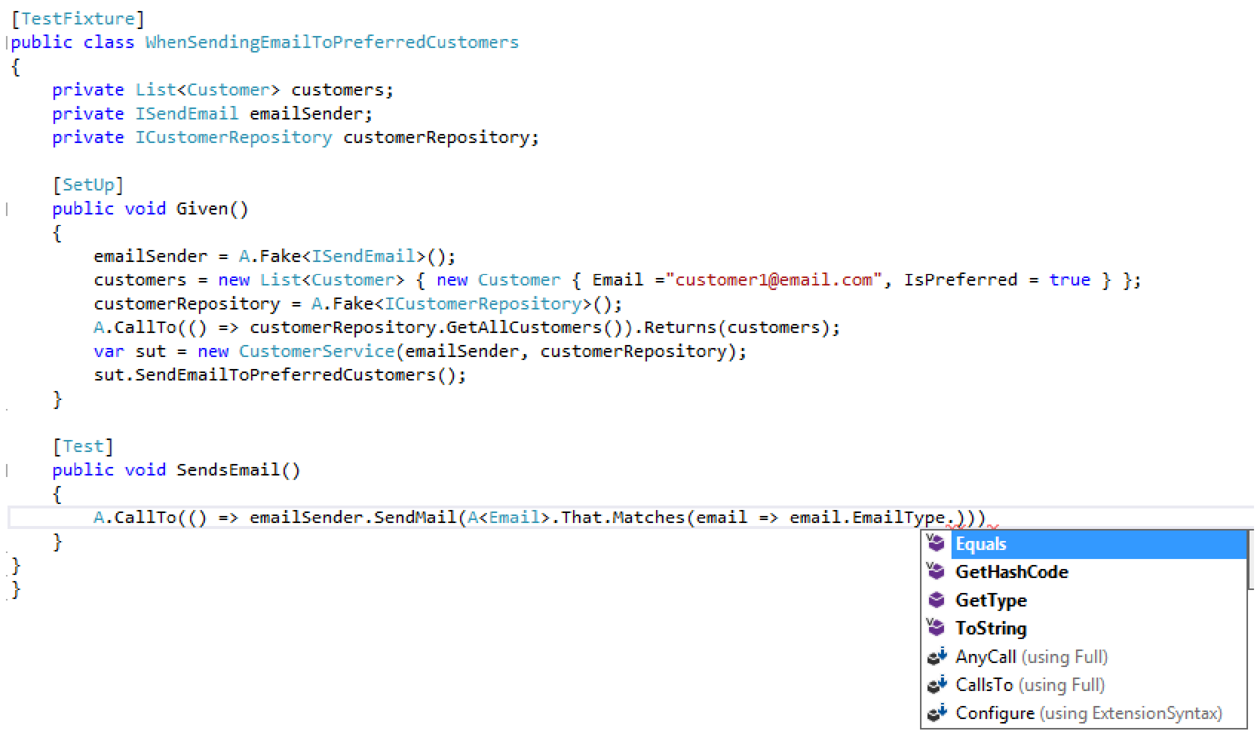


```
[TestFixture]
public class WhenSendingEmailToPreferredCustomers
{
    private List<Customer> customers;
    private ISendEmail emailSender;
    private ICustomerRepository customerRepository;

    [SetUp]
    public void Given()
    {
        emailSender = A.Fake<ISendEmail>();
        customers = new List<Customer> { new Customer { Email ="customer1@email.com", IsPreferred = true } };
        customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => customerRepository.GetAllCustomers()).Returns(customers);
        var sut = new CustomerService(emailSender, customerRepository);
        sut.SendEmailToPreferredCustomers();
    }

    [Test]
    public void SendsEmail()
    {
        A.CallTo(() => emailSender.SendMail(A<Email>.That.Matches(email => email.)))
    }
}
}
```

Figure 30: Trying to get the email address to assert against in the unit test

In earlier examples, when we were trying to assert that the email was sent to the correct email address, our **`Email`** class contained a string property called **`Email`**, and we could assert against that value easily. But now that our **`Email`** class takes an object property called **`EmailType`**, how do we get at the actual email address we want to assert against?

Let's choose **`EmailType`**, and see what IntelliSense gives us:



```
[TestFixture]
public class WhenSendingEmailToPreferredCustomers
{
    private List<Customer> customers;
    private ISendEmail emailSender;
    private ICustomerRepository customerRepository;

    [SetUp]
    public void Given()
    {
        emailSender = A.Fake<ISendEmail>();
        customers = new List<Customer> { new Customer { Email ="customer1@email.com", IsPreferred = true } };
        customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => customerRepository.GetAllCustomers()).Returns(customers);
        var sut = new CustomerService(emailSender, customerRepository);
        sut.SendEmailToPreferredCustomers();
    }

    [Test]
    public void SendsEmail()
    {
        A.CallTo(() => emailSender.SendMail(A<Email>.That.Matches(email => email.EmailType.)))
    }
}
}
```

Figure 31: We still can't get to the email value we want to assert against

Here you'll see that we still cannot get to the email value we want to assert against. The only things for us to pick are the IntelliSense items offered to us by an object type.

How are we going to test this class? We need a way to cast **`EmailType`** to the correct class in order to get access to that classes **`Email`** property.

Here is the unit test that will make it possible:

```
[TestFixture]
public class WhenSendingEmailToPreferredCustomers
{
    private List<Customer> customers;
    private ISendEmail emailSender;
    private ICustomerRepository customerRepository;

    [SetUp]
    public void Given()
    {
        emailSender = A.Fake<ISendEmail>();
        customers = new List<Customer> { new Customer { Email ="customer1@email.com",
            IsPreferred = true } };

        customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => customerRepository.GetAllCustomers()).Returns(customers);

        var sut = new CustomerService(emailSender, customerRepository);
        sut.SendEmailToPreferredCustomers();
    }
    [Test]
    public void SendsEmail()
    {
        A.CallTo(() => emailSender.SendMail(A<Email>.That.Matches(
            x => (x.EmailType as PreferredCustomerEmail).Email == customers[0].Email)))
                .MustHaveHappened(Repeated.Exactly.Once);
    }
}
```

Code Listing 107: Casting the object property type to the correct class to get the email value

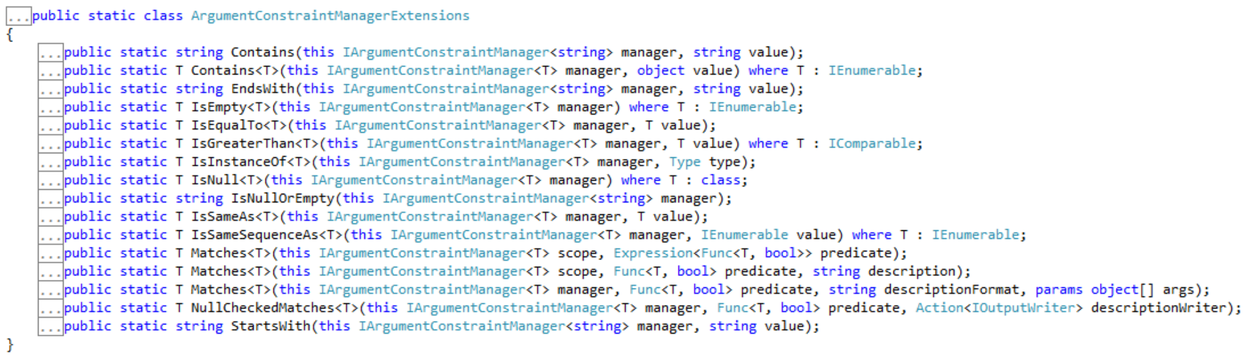
Here you can see that we're still using **`A<Email>.That.Matches`**, but in our lambda, instead of **`x => x.Email`**, we're using **`x => (x.EmailType as PreferredCustomerEmail)`**.

By casting the object to the type we need in **`Matches`**, we can then finally access the **`Email`** property that contains the email address we wish to assert against.

Extra Credit: Write a unit test that will test the non-happy path of the implementation currently in the **`CustomerService`** class. You can see the conditional check in there, and that's almost always a sign that multiple tests should be written.

Other Available Matchers

FakeItEasy provides us with other built-in matchers besides the ones we've been exploring in this chapter, via **`That`**. In our IDE, if we hit **F12**, and file over the **`Matches`** function, we'll be brought to the **`ArgumentConstraintManagerExtensions`** static class:



```
...public static class ArgumentConstraintManagerExtensions
{
    ...public static string Contains(this IArgumentConstraintManager<string> manager, string value);
    ...public static T Contains<T>(this IArgumentConstraintManager<T> manager, object value) where T : IEnumerable;
    ...public static string EndsWith(this IArgumentConstraintManager<string> manager, string value);
    ...public static T IsEmpty<T>(this IArgumentConstraintManager<T> manager) where T : IEnumerable;
    ...public static T IsEqualTo<T>(this IArgumentConstraintManager<T> manager, T value);
    ...public static T IsGreaterThan<T>(this IArgumentConstraintManager<T> manager, T value) where T : IComparable;
    ...public static T IsInstanceOf<T>(this IArgumentConstraintManager<T> manager, Type type);
    ...public static T IsNull<T>(this IArgumentConstraintManager<T> manager) where T : class;
    ...public static string IsNullOrEmpty(this IArgumentConstraintManager<string> manager);
    ...public static T IsSameAs<T>(this IArgumentConstraintManager<T> manager, T value);
    ...public static T IsSameSequenceAs<T>(this IArgumentConstraintManager<T> manager, IEnumerable value) where T : IEnumerable;
    ...public static T Matches<T>(this IArgumentConstraintManager<T> scope, Expression<Func<T, bool>> predicate);
    ...public static T Matches<T>(this IArgumentConstraintManager<T> scope, Func<T, bool> predicate, string description);
    ...public static T Matches<T>(this IArgumentConstraintManager<T> manager, Func<T, bool> predicate, string descriptionFormat, params object[] args);
    ...public static T NullCheckedMatches<T>(this IArgumentConstraintManager<T> manager, Func<T, bool> predicate, Action<IOutputWriter> descriptionWriter);
    ...public static string StartsWith(this IArgumentConstraintManager<string> manager, string value);
}
```

Figure 32: The ArgumentConstraintManagerExtensions class

You are encouraged to explore and experiment with all the matchers provided by this static class. A full list of available matchers can be found here: https://github.com/FakeItEasy/FakeItEasy/wiki/Argument-Constraints.

Summary

Argument matchers are very powerful, whether using the ones FakeItEasy provides, or writing your own custom matcher(s). I recommend you spend some time working with all of the FakeItEasy-supplied matchers as well as trying to build your own. In this chapter, we learned how to pass arguments to methods, how to use FakeItEasy-provided matchers, how to write custom matchers, and how to deal with **`object`**.

At this point, we've covered all the basics of what you need to get up and running with FakeItEasy in your unit tests. This should be able to cover 75 percent or more of your faking and unit testing needs. All of the examples we've been using and exploring up until this point have been focusing on faking interfaces that have been injected into classes (our SUT), and then specifying setup and behavior for those faked interfaces.

What we have NOT seen yet is if we need to fake the SUT itself. In the next chapter, we'll work through an entire example of why we would want to fake the SUT, how to fake the SUT, and explore how it differs from using FakeItEasy to configure injected dependencies.

Chapter 9 Faking the SUT

For most of this book, we've been exploring FakeItEasy's API from the perspective of creating fakes for dependencies that are injected into the SUT. We've learned a good amount on how to create, configure, and then assert calls to faked dependences so we can test our SUT.

But what happens when we fake the SUT?

Many times, we need to fake the SUT because the SUT is using inheritance in some fashion, and there is no other way for FakeItEasy to get involved (another great reason to use inheritance as your last resort, not your first). If we're trying to get a hold of the behavior and expectations of members that are declared as protected virtual, protected abstract void, protected overrides, and virtual readonly, we need to fake the SUT in order to write the unit test.

I need to add this disclaimer here: I do not recommend faking the SUT unless you have to.

If this code is "overriding" code in the SUT via FakeItEasy configuration, then there is a chance the real code that you're faking would never be tested. In other words, if you don't watch yourself, you could end up releasing untested code to production.

Some readers may not think this is a big deal, but at my current job, we have almost 90 percent unit test coverage (the missing 10 percent are covered by integration tests), and our programming department takes great care to keep all code written covered 100 percent (or as close as possible) by unit tests.

That being said, let's take a look at how to fake the SUT in this chapter, where the SUT is inheriting from an abstract class.

Creating the Fake

This is how we create a fake of our SUT:

```
var sut = A.Fake<ClassThatIsMySut>();
```

Code Listing 108: Creating a fake of the SUT

This syntax should look familiar to you by now. But this time, we're not creating a fake of an interface that the SUT uses and injecting it in via the SUT's constructor, but rather, creating a fake of the SUT itself.

Here is the overload of A.**CallTo** we'll be using to fake the SUT:

```
// Summary:
//     Gets a configuration object allowing for further configuration of any call
//     to the specified faked object.
//
```

```
// Parameters:
//   fake:
//     The fake to configure.
//
// Returns:
//     A configuration object.
public static IAnyCallConfigurationWithNoReturnTypeSpecified CallTo(object fake);
```

Code Listing 109: CallTo overload for faking the SUT

Let's dive into faking the SUT with a redesign of some of the supporting classes involved in sending an email.

A New EmailSender

Let's set up a new example with our **ISendEmail** interface we've using throughout the book, but this time, we'll inject **ISendEmail** into an abstract base class, which will contain the functionality to actually invoke the call to **SendMail**.

Our **ISendEmail** interface:

```
public interface ISendEmail
{
    void SendMail(string from, string to, string subject, string body);
}
```

Code Listing 110: The ISendEmail interface

Here is our **Customer** class:

```
public class Customer
{
    public string Email { get; set; }
}
```

Code Listing 111: The Customer class

Let's add a new abstract class called **EmailBase**:

```
public abstract class EmailBase
{
    private readonly ISendEmail emailProvider;

    protected EmailBase(ISendEmail emailProvider)
    {
        this.emailProvider = emailProvider;
    }

    protected void SendEmailToCustomers(string subject, string body,
        List<Customer> customers)
    {
        foreach (var customer in customers)
        {
            emailProvider.SendMail(GetFromEmailAddress(), customer.Email, subject,
                body);
        }
    }

    protected virtual string GetFromEmailAddress()
    {
        return ConfigurationManager.AppSettings["DefaultFromAddress"];
    }
}
```

Code Listing 112: Abstract class EmailBase

Let’s take a look at what **EmailBase** is doing:

- It takes an **ISendEmail** interface to delegate sending email.
- Its **SendEmailToCustomers** method takes a list of customers, and then for each customer, invokes a call to **SendMail**.
- It contains a protected virtual method called **GetFromEmailAddress**.
    - This method wraps a call to the **ConfigurationManager** that is responsible for retrieving a “From” email address from configuration

EmailBase is not really giving us more functionality than the fakes of **ISendEmail** we’ve been working with for most of the book, except for one thing: it does not required the caller to pass a “From” email address into the **SendMail** method. In this example, let’s assume that the same “From” address will be used any time we’re sending email to customers.

Finally, here is the **ICustomerRepository** interface:

```
public interface ICustomerRepository
{
    List<Customer> GetAllCustomersWithOrderTotalsOfOneHundredOrGreater();
}
```

Code Listing 113: The ICustomerRepository interface

GetAllCustomersWithOrderTotalsOfOneHundredOrGreater allows our service class to retrieve a list of customers that have placed one hundred or more orders.

A New Service Class

Next, let’s introduce a service class, **AdminEmailService**:

```
public class AdminEmailService : EmailBase
{
    private readonly ICustomerRepository customerRepository;

    public AdminEmailService(ICustomerRepository customerRepository,
        ISendEmail emailProvider)
        : base(emailProvider)
    {
        this.customerRepository = customerRepository;
    }

    public void SendPromotionalEmail(string subject, string body)
    {
        var customers = customerRepository
            .GetAllCustomersWithOrderTotalsOfOneHundredOrGreater();
        SendEmailToCustomers(subject, body, customers);
    }
}
```

Code Listing 114: The AdminEmailService class

The **`AdminEmailService`** class provides an easy way to send promotional emails to customers that have 100 or more orders placed. For example, the email we're sending could contain a coupon code for discounts because we want to reward our highest purchasing customers. The functionality in this class could be used by the sales department, marketing department, etc. in order to send out these types of emails.

Note that this class inherits from **`EmailBase`**, and takes an **`ICustomerRepository`** and **`ISendEmail`** dependency. But this class does not use the **`ISendEmail`** dependency; it passes that dependency into the constructor of its super-class, **`EmailBase`**.

The Unit Tests

Here is the **`SetUp`** method of the unit test:

```
[TestFixture]
public class WhenSendingPromotionalEmail
{
    private ISendEmail emailSender;
    private List<Customer> customers;
    private const string emailSubject = "EmailSubject";
    private const string emailBody = "EmailBody";
    private const string theConfiguredFromAddress = "someone@somecompany.com";

    [SetUp]
    public void Given()
    {
        customers = new List<Customer>
        {
            new Customer { Email = "customer1@email.com" },
            new Customer { Email = "customer2@email.com" }
        };

        var customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => customerRepository
            .GetAllCustomersWithOrderTotalsOfOneHundredOrGreater()).Returns(customers);

        emailSender = A.Fake<ISendEmail>();

        var sut = A.Fake<AdminEmailService>(x => x.WithArgumentsForConstructor(() =>
            new AdminEmailService(customerRepository, emailSender)));
        A.CallTo(sut).Where(x => x.Method.Name == "GetFromEmailAddress")
            .WithReturnType<string>().Returns(theConfiguredFromAddress);

        sut.SendPromotionalEmail(emailSubject, emailBody);
    }
}
```

Code Listing 115: Setup for unit test of AdminEmailService

Upon first glance, some things look very much the same as we've seen from our unit tests written so far in this book. We are still creating a fake of **`ICustomerRepository`** and **`ISendEmail`** using FakeItEasy, but after that creation and configuration code, things start to look very different.

Faking the SUT introduces us to new API calls we have not seen yet. Let's explore each of the differences in-depth.

Creating the Fake

First, let's take a look at the code we're using for creating our fake:

```
var sut = A.Fake<AdminEmailService>(x => x.WithArgumentsForConstructor(() =>
    new AdminEmailService(customerRepository, emailSender)));
```

Code Listing 116: Creating a fake of the SUT

Since the **`AdminEmailService`** takes two items in its constructor, we need to supply that to the **`A.Fake<AdminEmailService>`** call. We do this by passing a lambda to the call that does two things:

- Uses the new keyword to create the **`AdminEmailService`** class
- Passes the two earlier created fakes (**`ICustomerRepository`** and **`ISendEmail`**) to the constructor of **`AdminEmailService`**

These two operations are done by calling **`WithArgumentsForConstructor`**, then passing a lambda to that call, which FakeItEasy uses to examine the expression in order to figure out which arguments will be used when creating the fake.

If we look at the signature for the overloaded **`A.Fake<T>`** creation, we'll see the following:

```
// Summary:
//     Creates a fake object of the type T.
//
// Parameters:
//   options:
//     A lambda where options for the built fake object can be specified.
//
// Type parameters:
//   T:
//     The type of fake object to create.
//
// Returns:
```

```
//      A fake object.
public static T Fake<T>(Action<FakeItEasy.Creation.IFakeOptionsBuilder<T>> options);
```

Code Listing 117: Fake<T> using IFakeOptionsBuilder

Note here that the Action used by **`Fake<T>`** uses **`IFakeOptionsBuilder<T>`**. Exploring the **`IFakeOptionsBuilder<T>`** interface, we see the following:

```
public interface IFakeOptionsBuilder<T> : IHideObjectMembers
{
    IFakeOptionsBuilder<T> Implements(Type interfaceType);
    IFakeOptionsBuilder<T> OnFakeCreated(Action<T> action);
    IFakeOptionsBuilder<T>
        WithAdditionalAttributes(IEnumerable<Reflection.Emit.CustomAttributeBuilder>
            customAttributeBuilders);
    IFakeOptionsBuilder<T> WithArgumentsForConstructor(Expression<Func<T>>
        constructorCall);
    IFakeOptionsBuilder<T> WithArgumentsForConstructor(IEnumerable<object>
       argumentsForConstructor);
    IFakeOptionsBuilderForWrappers<T> Wrapping(T wrappedInstance);
}
```

Code Listing 118: IFakeOptionsBuilder interface

There are multiple overloads of **`WithArgumentsForConstructor`**. Please feel free to explore the other options you can call when creating a fake with **`IFakeOptionsBuilder`** on your own.

Configuring the Fake

Now that we have a fake of our SUT created, we configure it using the following code:

```
A.CallTo(sut).Where(x => x.Method.Name == "GetFromEmailAddress")
    .WithReturnType<string>().Returns(theConfiguredFromAddress);
```

Code Listing 119: Configuring the faked SUT

Since the only thing going on in **`AdminEmailService`** is a call **`ICustomerRepository`** to get the customers that have placed 100 or more orders, and since we've already taken care of configuring those fakes earlier in the setup code, what we have left to do is configure the protected virtual method, **`GetFromEmailAddress`**.

`Where(x => x.Method.Name == "GetFromEmailAddress")`

This is how we allow FakeItEasy to get a hold of that method to configure it. Note this is done by providing a lambda to the **`Where`** call, using that lambda to look up the **`Method.Name`** and supplying the name of the method. We also have to use **`WithReturnType<T>`** in order to tell FakeItEasy that we expect a string back from this call.

The **`Where`** method we're using here is NOT the **`Where`** method provided to us by LINQ. It's a **`Where`** method that is provided to us by a FakeItEasy extension, and it looks like this:

```
public static class WhereConfigurationExtensions
```

```
{
    public static T Where<T>(this IWhereConfiguration<T> configuration,
        Expression<Func<FakeItEasy.Core.IFakeObjectCall, bool>> predicate);
}
```

Code Listing 120: The Where extenstion method in FakeItEasy

The **`Where`** extension uses an **`Expression<Func<T, bool>>`** in order to figure out which method to look up by the provided string.

The configuration code ends with a call to **`Returns`**, which allows us to specify a value we expect to be returned from the configured method call.

An Important Note

If any of you at this point are looking at the name of the method being supplied as a string to the configuration and not something that is strongly typed by FakeItEasy, and thinking that looks wrong, in a way you're right.

When we began this chapter, I stated that if you can avoid faking the SUT, then don't do it. I also mentioned that many times, when you're faking the SUT, you're usually dealing with a class that inherits from a base class, and there most likely will be some behavior in that base class you'll need to configure.

This his is how we have to deal with this scenario using FakeItEasy. The danger here, of course, is at compile time. If someone changes the name of **`GetFromEmailAddress`** method on **`EmailBase`**, you're not going to know at compile time—you'll only know at runtime.

Which again, is not a big deal, because everyone is running ALL their unit tests before they commit their code, right? Or maybe you have a Continuous Integration (http://en.wikipedia.org/wiki/Continuous_integration) server set up that will kick back failing builds as a result of code that won't compile, or code where there are any failing unit tests.

In a perfect world, yes… but many of us work in an imperfect world where developers do commit code without running all unit tests, and we don't have Continuous Integration in place.

If you choose to use FakeItEasy in this way, be aware of these potential problems.

Assertions

Now that we have our fake created and configured, it's time to start using it in assertions for our unit tests. If you remember from our test setup, we have two customers that we expect email to be sent to. Here is our first unit test:

```
[Test]
public void SendsTheCorrectAmountOfTimes()
{
    A.CallTo(() => emailSender
        .SendMail(theConfiguredFromAddress, A<string>._, emailSubject, emailBody))
```

```
            .MustHaveHappened(Repeated.Exactly.Twice);
}
```

Code Listing 121: Assertion for the emailSender being called twice

This type of assertion code should look very familiar to you. Note that we're using **`A<string>._`** shorthand for **`A<string>.Ignored`**. Since we set up two customers to be returned from our **`customerRepository`** fake, we expect **`SendMail`** to be invoked twice. How could we change this test to be better? How about this:

```
[Test]
public void SendsTheCorrectAmountOfTimes()
{
    A.CallTo(() => emailSender
        .SendMail(theConfiguredFromAddress, A<string>._, emailSubject, emailBody))
            .MustHaveHappened(Repeated.Exactly.Times(customers.Count()));
}
```

Code Listing 122: Using customers.Count() instead of hard coding .twice

We've improved the test by getting its count from the customer's collection that we defined in our unit test setup.

Still, we can do better.

We know it's important that the email is sent twice, but if two emails were sent to the wrong customer, I think the customers who received those emails would be scratching their heads. Furthermore, we just gave a low-purchasing customer a discount coupon code that they should not be receiving.

Let's test for the actual email address value:

```
[Test]
public void SendsToCorrectCustomers()
{
    A.CallTo(() => emailSender.SendMail(theConfiguredFromAddress, customers[0].Email,
        emailSubject, emailBody)).MustHaveHappened(Repeated.Exactly.Once);
    A.CallTo(() => emailSender.SendMail(theConfiguredFromAddress, customers[1].Email,
        emailSubject, emailBody)).MustHaveHappened(Repeated.Exactly.Once);
}
```

Code Listing 123: Testing that we're sending email to the correct customers

This is testing what should be tested: that the correct email addresses were used when invoking **`SendMail`**. But again, it looks like we have those indexes hard-coded for the customers list; we've lost the flexibility of the **`customers.Count`** change from earlier. Let's make one more correction.

```
[Test]
public void SendsToCorrectCustomers()
{
    foreach (var customer in customers)
        A.CallTo(() => emailSender.SendMail(theConfiguredFromAddress, customer.Email,
            emailSubject, emailBody))
```

```
                .MustHaveHappened(Repeated.Exactly.Once);
}
```

Code Listing 124: Interating through customers for each assertion

We're looping through each customer and asserting that the correct customer's email address was used. Now if we want to change the number of customers in our setup, we don't have to change multiple lines of code and mess with indexes in our test.

Calling Protected Property Getters/Setters

Similar to how we configured FakeItEasy's behavior when calling the **`GetFromEmailAddress`** method in Code Listing 119 in the previous section, it's possible to call protected property getters/setters as well. Let's explore a code sample to call a protected abstract property getter.

The interfaces from the previous sample stay the same:

```
public interface ICustomerRepository
{
    List<Customer> GetAllCustomersWithOrderTotalsOfOneHundredOrGreater();
}
```

Code Listing 125: The ICustomerRepository interface

```
public interface ISendEmail
{
    void SendMail(string from, string to, string subject, string body);
}
```

Code Listing 126: The ISendEmail interface

We're going to change the **`EmailBase`** class from the previous example:

```
public abstract class EmailBase
{
    private readonly ISendEmail emailProvider;

    protected EmailBase(ISendEmail emailProvider)
    {
        this.emailProvider = emailProvider;
    }

    protected void SendEmailToCustomers(string subject, string body,
        IEnumerable<Customer> customers)
    {
        foreach (var customer in customers)
        {
            emailProvider.SendMail(FromEmailAddress, customer.Email, subject, body);
        }
    }
    protected abstract string FromEmailAddress { get; }
}
```

Code Listing 127: Abstract class EmailBase

In the **EmailBase** class, instead of having a protected method to get the from email address, we've changed the mechanism to retrieve the **FromEmailAddress** to a protected abstract property getter. Any classes that inherit from this base class will need to provide an implementation of **FromEmailAddress**. Delegating the implementation of the retrieving the **FromEmailAddress** to the inheriting class adds a level of flexibility to this class in that it's not looking up the email address using a hard-coded string via **ConfigurationManager** that can only be one value for one given configuration.

Note: Declaring protected abstract methods/properties on a base class like this is a GoF Design Pattern called a Template Method(http://www.dofactory.com/net/template-method-design-pattern). One of the more useful design patterns, it allows you combine functionality in an abstract class with the contract enforcement of an interface, and allows that functionality to vary by each class that inherits from the abstract class.

Let's create a class that will use **EmailBase**. Sticking with the notion that we want a way to send promotional emails to customers, we'll create a **PromotionalEmailService** class:

```
public class PromotionalEmailService : EmailBase
{
    private readonly ICustomerRepository customerRepository;

    public PromotionalEmailService(ICustomerRepository customerRepository,
        ISendEmail emailProvider) : base(emailProvider)
    {
        this.customerRepository = customerRepository;
    }

    public void SendEmail(string subject, string body)
    {
        var customers = customerRepository
            .GetAllCustomersWithOrderTotalsOfOneHundredOrGreater();
        SendEmailToCustomers(subject, body, customers);
    }

    protected override string FromEmailAddress
    {
        get { return "APromotionalEmail@somecompany.com"; }
    }
}
```

Code Listing 128: The PromotionalEmailService class

The **PromotionalEmailService** class inherits from **EmailBase**, takes **ICustomerRepository** and **ISendEmail** dependencies, and passes **ISendEmail** to the **EmailBase**'s constructor. This class is also forced to implement the **FromEmailAddress** getter property via the abstract classes' protected abstraction property declaration.

Let's write some unit tests for this class to see how to deal with that protected abstract property-getter using FakeItEasy. First, the **SetUp** method on the test class:

```
public class WhenSendingPromotionalEmail
{
    private List<Customer> customers;
    private ISendEmail emailSender;
    private const string subject = "Subject";
    private const string body = "Body";
    private const string fromAddress = "fromAddress";

    [SetUp]
    public void Given()
    {
        customers = new List<Customer>
        {
            new Customer { Email = "customer1@email.com" },
            new Customer { Email = "customer2@email.com" }
        };

        emailSender = A.Fake<ISendEmail>();
        var customerRepository = A.Fake<ICustomerRepository>();
        A.CallTo(() => customerRepository
            .GetAllCustomersWithOrderTotalsOfOneHundredOrGreater()).Returns(customers);

        var sut = A.Fake<PromotionalEmailService>(x => x.WithArgumentsForConstructor(
            () => new PromotionalEmailService(customerRepository, emailSender)));
        A.CallTo(sut).Where(x => x.Method.Name == "get_FromEmailAddress")
            .WithReturnType<string>().Returns(fromAddress);
        sut.SendEmail(subject, body);
    }
}
```

Code Listing 129: The PromotionalEmailServiceTests unit test setup method

In our **Where** clause, in order to access the **FromEmailAddress** property, we append a **get_** to the front of the name of the property we want to use in order to tell FakeItEasy what type and value to return. Appending that **get_** to the front of the **FromEmailAddress** property allows FakeItEasy to get access to the property.

The test methods of our unit test class:

```
[Test]
public void SendsTheCorrectAmountOfTimes()
{
    A.CallTo(() => emailSender
        .SendMail(fromAddress, A<string>._, subject, body))
            .MustHaveHappened(Repeated.Exactly.Times(customers.Count()));
}

 [Test]
public void SendsToCorrectCustomers()
{
    foreach (var customer in customers)
    {
        A.CallTo(() => emailSender
            .SendMail(fromAddress, customer.Email, subject, body))
                .MustHaveHappened(Repeated.Exactly.Once);
    }
}
```

Code Listing 130: The test methods for the PromotionalEmailServiceTests class

We have two assertions in our unit test based on our setup. **SendsTheCorrectAmountOfTimes** makes sure that **SendMail** is called for the number of customers in our setup. **SendsToCorrectCustomers** makes sure each email is going to the correct customer.

Now we've seen how to call a protected property getter. We call a protected property *setter* in the same exact way, except instead of prepending **get_** to the beginning of the property name, we preppend **set_** to the beginning of the property name.

If, in Code Listing 129, the **FromEmailAddress** was a property setter instead of a property getter, the configuration code would look like this:

```
A.CallTo(sut).Where(x => x.Method.Name == "set_FromEmailAddress")
    .Invokes((string a) => fromAddress = a);
```

Code Listing 131: What the configuration code would look like if FromEmailAddress was a property setter instead of property getter

Summary

In this chapter, we've covered why we would need to create a fake of the SUT. We then wrote unit tests for the faked SUT, and finished with a quick look at how to get at protected properties with FakeItEasy. In the next chapter, we'll take a look at how to use FakeItEasy to test MVC action methods.

Chapter 10 MVC and FakeItEasy

Introduction

HttpContext.

Just the sound of that class makes the most seasoned .NET developers quake in their boots. HttpContext is the BIGGEST sealed object created in the history of mankind, a true Microsoft treasure. A treasure, unfortunately, all of us are stuck with.

The real problem with HttpContext isn't the fact that it's gigantic and sealed (although that doesn't help), but the fact that it's so easy to reach out and use HttpContext in an MVC controller, and not even know you've done it.

That being said, I have an assignment for you.

It's to read this blog post on how to mock HttpContext. This is required reading to get the most out of this section, so please stop now, read the article in its entirety, and return to the book when you're done.

Before reading this blog post, I didn't realize trying to mock HttpContext defeated even the mighty Chuck Norris…but let's talk a bit about the solution the author outlined.

An Interface for Everything! (not so fast…)

This article shows how the author went about abstracting the pieces of HttpContext via an interface that allowed his previously untestable controller classes to be testable.

For example, the author created the **`ICurrentUser`** interface, which wrapped HttpContext calls and then injected that interface into the **`OrderController`**.

Maybe you were thinking of the FakeItEasy code you would write to fake that interface to allow the **`OrderController`** to be testable. This is a viable solution and will work, but let's look at some of the drawbacks.

HttpContext is huge. If you need to fake more than just a couple calls to HttpContext.**`Current.User.IsInRole`** (like HttpRequest, HttpResponse, HttpSessionState, etc.), you're going to end up either writing a huge interface with a bunch of members, which violates the interface segregation principle (http://www.objectmentor.com/resources/articles/isp.pdf), or writing a bunch of little interfaces that start to overwhelm your controller's constructor with dependencies. You still need room in your controller's constructor for other dependencies, like repositories.

There has to be a better way.

There is, using FakeItEasy MVC's controller extensibility points, and new "Base" classes that .NET has added to its framework to allow testing of those sealed classes. Here's how.

MVC and ControllerContext

If we look at the **`ControllerBase`** class that all MVC controllers inherit from, we'll see a property called **`ControllerContext`**.

```
public abstract class ControllerBase : IController
{
    protected ControllerBase();
    public ControllerContext ControllerContext { get; set; }
    public TempDataDictionary TempData { get; set; }
    public bool ValidateRequest { get; set; }
    public IValueProvider ValueProvider { get; set; }
    [Dynamic]
    public dynamic ViewBag { get; }
    public ViewDataDictionary ViewData { get; set; }
    protected virtual void Execute(System.Web.Routing.RequestContext requestContext);
    protected abstract void ExecuteCore();
    protected virtual void Initialize(System.Web.Routing.RequestContext
       requestContext);
}
```

Code Listing 132: The ControllerContext property on the ControllerBase class

It is this extensibility point added by Microsoft to the controller inheritance structure that is going to allow us to use FakeItEasy to control things like HttpResponse, HttpRequest, HttpSessionState, and other sealed classes that FakeItEasy would otherwise not be able to control.

Let's take a look the **`ControllerContext`** class.

```
public class ControllerContext
{
    public ControllerContext();
    protected ControllerContext(ControllerContext controllerContext);
    public ControllerContext(RequestContext requestContext, ControllerBase controller);
    public ControllerContext(HttpContextBase httpContext, RouteData routeData,
        ControllerBase controller);
    public virtual ControllerBase Controller { get; set; }
    public IDisplayMode DisplayMode { get; set; }
    public virtual HttpContextBase HttpContext { get; set; }
    public virtual bool IsChildAction { get; }
    public ViewContext ParentActionViewContext { get; }
    public RequestContext RequestContext { get; set; }
    public virtual RouteData RouteData { get; set; }
}
```

Code Listing 133: The ControllerContext class

We have four overloads for the constructor, including one that takes no parameters. We're going to explore the overload that takes three parameters: **`public ControllerContext(HttpContextBase httpContext`**, **`RouteData routeData`**, and **`ControllerBase controller)`**.

But first, what is **`HttpContextBase`**? What happened to **`HttpContext`**?

Putting the "Base" in System.Web's Sealed Classes

Returning to the "Chuck Norris" example at the beginning of this chapter, Microsoft knew it needed to do something to allow MVC to be a unit-testing-friendly framework. Its current lineup of classes in the System.Web library consisted of all sealed classes, which are not testable by most mocking and faking frameworks.

As a result, they created "Base" classes that are exposed via the MVC framework. At runtime, these "Base" classes are delegating to the real static classes.

For example, here is the class declaration for **`HttpContext`** (un-testable):

```
public sealed class HttpContext : IServiceProvider, IPrincipalContainer
```

Code Listing 134: Class declaration for HttpContext; not testable because it's sealed

And here is the class declaration for **`HttpContextBase`** (testable):

```
public abstract class HttpContextBase : IServiceProvider
```

Code Listing 135: Class declaration for HttpContextBase

Going all the way back to Chapter 3, "Introducing FakeItEasy," we remember that you cannot fake a sealed class with FakeItEasy.

So what makes the members of **`HttpContextBase`** fake-able? Every one of them is declared as virtual. By declaring members as virtual, you allow FakeItEasy to be able to use them in configuration, behavior, and assertions. The same applies for the **`HttpRequestBase`** and **`HttpResponseBase`** classes and their respective untestable classes (**`HttpRequest`** and **`HttpResponse`**) they delegate to at runtime.

Setting Up ControllerContext with FakeItEasy

Now that we have a solid understanding of the "Base" classes in System.Web, let's take a look at how to use FakeItEasy to set up the fakes we'll need when creating a **`ControllerContext`**.

To start off, let's build a VERY simple MVC controller and call it **`HomeController`**:

```
public class HomeController : Controller
{
    [HttpPost]
    public void WriteToResponse()
    {
        Response.Write("writing to response");
    }
}
```

Code Listing 136: The HomeController class

The **`WriteToResponse`** method on **`HomeController`** writes a string to the response stream. Usually, given the richness of MVC's models and model-binding support, we rarely perform operations like this, but for the sake of the example, let's stick use this action method as a starting point.

Here is the FakeItEasy setup method for testing the MVC action method in Code Listing 136:

```
[TestFixture]
public class WhenWritingToResponse
{
    private HttpResponseBase httpResponse;

    [SetUp]
    public void Given()
    {
        var sut = new HomeController();
        var httpContext = A.Fake<HttpContextBase>();
        httpResponse = A.Fake<HttpResponseBase>();
        A.CallTo(() => httpContext.Response).Returns(httpResponse);
        var context = new ControllerContext(new RequestContext(httpContext,
            new RouteData()), sut);

        sut.ControllerContext = context;
        sut.WriteToResponse();
    }
}
```

Code Listing 137: The setup method for the class testing HomeController

Newing up our SUT is done as it's always been done—by newing it up directly, and then passing any dependencies to its constructor. Since **`HomeController`** takes no dependencies, creating the SUT is very straightforward in this example.

On the next line, you can see we're creating a fake of the **`HttpContextBase`** class:
`var httpContext = A.Fake<HttpContextBase>();`

After we create a fake of **`HttpContextBase`**, we create a fake of **`HttpResponseBase`**:
`httpResponse = A.Fake<HttpResponseBase>();`

Once both of these fakes are created, we can now configure that calling the faked **`HttpContextBase`**'s **`Response`** property will return the faked **`HttpResponseBase`**.

Now we can finally get around to creating our **`ControllerContext`** class. We do so by passing in a new **`RequestContext`**, and passing the faked **`HttpContextBase`**, a new **`RouteData`**, and the controller that context will be used for into the constructor for **`RequestContext`**.

To finish up the test setup, we then assign the SUT’s **`ControllerContext`** property to the newly created **`ControllerContext`** object that we passed our configured fakes to. After we’ve done this, we invoke the **`WriteToResponse`** action method on our SUT.

The Unit Test

What we need to test in this case is very, very simple. We want to assert that a message of “writing to response” was passed to the **`Write`** method of the **`Response`** object exactly once. Here is what the code looks like:

```
[Test]
public void WritesToResponse()
{
    A.CallTo(() => httpResponse.Write("writing to response"))
        .MustHaveHappened(Repeated.Exactly.Once);
}
```

Code Listing 138: Testing the that the correct string was written by Response.Write

Compared to our setup method, this code looks very familiar. Upon first glance, you can’t even tell we’re testing an MVC controller; the code reads like most of the other test methods we’ve seen in the book so far.

But what about testing something more complex than writing to the response stream? Let’s expand our example to perform some more of the more common MVC tasks.

Faking Session

Many times, we’re working with **`Session`** in our controllers. Whether that means reading or writing from it, **`Session`** is another very easy item in **`HttpContext`** to reach out to and use. Fortunately, Microsoft has provided the **`HttpSessionStateBase`** class in which all of its members are declared as virtual.

Here is a new action method where we’re adding a new item to **`Session`**:

```
[HttpPost]
public void AddCustomerEmailToSession(string customersEmail)
{
    Session.Add("CustomerEmail", customersEmail);
}
```

Code Listing 139: The new AddCustomerEmailToSession action method

And here is our new test setup to include the session state in our faked **`HttpContextBase`**:

```
[TestFixture]
public class WhenAddingToSession
{
    private const string customerEmail = "customer@email.com";
    private HttpSessionStateBase httpSession;

    [SetUp]
    public void Given()
    {
        var sut = new HomeController();

        var httpContext = A.Fake<HttpContextBase>();
        var httpResponse = A.Fake<HttpResponseBase>();
        httpSession = A.Fake<HttpSessionStateBase>();

        A.CallTo(() => httpContext.Response).Returns(httpResponse);
        A.CallTo(() => httpContext.Session).Returns(httpSession);

        var context = new ControllerContext(new RequestContext(httpContext,
            new RouteData()), sut);
        sut.ControllerContext = context;

        sut.AddCustomerEmailToSession(customerEmail);
    }
}
```

Code Listing 140: The test setup including a faked HttpSessionStateBase

We're creating a fake of **`HttpSessionStateBase`** and then setting that fake to be returned when the **`Session`** property on the faked **`HttpContextBase`** is invoked. By building up this "chain" of fakes, we now have full control of the most common calls made from **`HttpContext`** in the controller.

Here is the test that asserts we added the customer's email address to **`Session`**:

```
[Test]
public void AddCustomerEmailToSession()
{
    A.CallTo(() => httpSession.Add("CustomerEmail", customerEmail))
        .MustHaveHappened(Repeated.Exactly.Once);
}
```

Code Listing 141: The unit test for adding customer email to Session

Faking IPrincipal

So far, we've look at how to fake **`HttpContextBase`**, **`HttpRequestBase`**, **`HttpResponseBase`**, and **`HttpSessionStateBase`**. This covers a good amount of the type of code you'll be using in an MVC Controller, but there is one large part of an MVC application we have not talked about yet, and that is Authentication and Authorization. Let's take a look at how we can unit test code that uses **`IPrincipal`** in a controller's action method.

Note: You can find more information on IPrincipal here: https://msdn.microsoft.com/en-us/library/system.security.principal.iprincipal(v=vs.110).aspx.

Let's say that before we send an email, we want to reach out to the **`User`** property on the controller class and grab the current name of the authenticated user. We do this by accessing **`User.Identity.Name`** in our controller's action method. Here is an example using our current **`HomeController`**.

I've added the **`ISendEmail`** interface to this example and injected it into **`HomeController`**'s constructor.

```
public class HomeController : Controller
{
    private readonly ISendEmail sendEmail;

    public HomeController(ISendEmail sendEmail)
    {
        this.sendEmail = sendEmail;
    }

    [HttpPost]
    public void SendCustomerEmail(string to)
    {
        var user = User.Identity.Name;
        sendEmail.SendEmailTo("somecompany@somewhere.com", to,
            string.Format("This email is intended for {0}", user), "this is an email");
    }
}
```

Code Listing 142: Using User.Identity.Name to get the name of the authenticated user

You can see where we reach out to **`User.Identity.Name`** to get the name of the currently authenticated user. The **`User`** property is exposed as a read-only property off of the **`public abstract class Controller`** class that all controllers inherit from by default.

The property on the base controller class is of type **`IPrinciple`**:

```
public IPrincipal User { get; }
```

Code Listing 143: The User property of the abstract controller class

Looking at **`IPrincipal`**, we see the following:

```
public interface IPrincipal
{
    IIdentity Identity { get; }
    bool IsInRole(string role);
}
```

Code Listing 144: The IPrincipal interface

From this interface definition, you can see the **`Identity`** property, which is a type of **`IIdentity`**.

```
public interface IIdentity
{
    string AuthenticationType { get; }
    bool IsAuthenticated { get; }
    string Name { get; }
}
```

Code Listing 145: IIdentity interface

At runtime, .NET provides an implementation of **`IPrincipal`**, but when unit testing, it does not. Let's take a look at how to use FakeItEasy to fake **`IPrincipal`** to make the **`SendCustomerEmail`** action method testable.

Here is our test setup:

```
[TestFixture]
public class WhenSendingCustomerEmail
{
    private ISendEmail emailSender;
    private const string emailAddress = "customer1@somewhere.com";
    private const string userName = "UserName";

    [SetUp]
    public void Given()
    {
        emailSender = A.Fake<ISendEmail>();
        var sut = new HomeController(emailSender);
        sut.ControllerContext = new ControllerContext(new RequestContext(
            A.Fake<HttpContextBase>(), new RouteData()), sut);

        var principal = A.Fake<IPrincipal>();
        var identity = new GenericIdentity(userName);
        A.CallTo(() => principal.Identity).Returns(identity);
        A.CallTo(() => sut.ControllerContext.HttpContext.User).Returns(principal);

        sut.SendCustomerEmail(emailAddress);
    }
}
```

Code Listing 146: Unit test setup for controller action method using User.Identity.Name

Since we've added the **`ISendEmail`** interface to be used by the controller's action method, we first create a fake of **`ISendEmail`**, and pass that fake to the HomeController's constructor. We'll be using that configured fake for assertion in our test method.

Next, we've simplified the setup of the **`ControllerContext`** here mainly because the only thing we really need is a faked **`HttpContextBase`**. We don't need to set up a fake **`HttpRequestBase`**, **`HttpResponseBase`**, or **`HttpSessionStateBase`** because we're not using any of that code in our current action method under test.

After the **ControllerContext** is set on our SUT, we create a fake of **IPrincipal**, create a **GenericIdentity**, and then return that identity when the **Identity** property on our faked **IPrincipal** is called. The final line of code configures the **User** property on our **ControllerContext**.**HttpContext** property to return our faked **IPrincipal**.

Finally, now that we have our fakes created and configured, we call our SUT's **SendCustomerEmail** method, passing it an email address.

Here is the unit test method:

```
[Test]
public void SendsEmailToCustomerWithUserNameInSubject()
{
    A.CallTo(() => emailSender.SendEmailTo("somecompany@somewhere.com", emailAddress,
        string.Format("This email is intended for {0}", userName), "this is an email"))
            .MustHaveHappened(Repeated.Exactly.Once);
}
```

Code Listing 147: The unit test for SendCustomerEmail

We use our faked **EmailSender** to assert that **SendEmailTo** was invoked with the correct arguments, and one of those arguments includes the username of the authenticated user, which we configured via our faked **IPrincipal**.

Faking UrlHelper

To begin with, what is **UrlHelper**? If you've had to construct a URL while in a controller action method, and you've written code like the following, then you've used **UrlHelper** before.

```
var returnUrl = Url.Action("Index", "Home", null, Request.Url.Scheme);
```

Code Listing 148: Using UrlHelper in a controller

If you were to click F12 with your cursor over the "Url" part of **Url.Action**, you'd find yourself in the abstract Controller base class's **Url** property:

```
public UrlHelper Url { get; set; }
```

Code Listing 149: Rhe Url property is type of UrlHelper

Now that we know what **UrlHelper** is, let's take a look at how to test it with FakeItEasy.

Let's return to our **HomeController** class and add another action method named **BuildUrl**.

```
public ActionResult BuildUrl()
{
    var model = new BuildUrl { Url = Url.Action("Index", "Home", null,
        Request.Url.Scheme) };
    return View(model);
}
```

Code Listing 150: The BuildUrl action method

The **`BuildUrl`** method creates and populates a **`BuildUrl`** model class, and then returns the populated model in the returned view. Here is the **`BuildUrl`** model class.

```
public class BuildUrl
{
    public string Url { get; set; }
}
```

Code Listing 151: The BuildUrl model class

Note the use of **`Url.Action`** as well as **`Request.Url.Scheme`** in the **`BuildUrl`** method. We'll need fakes of both of these items in order to make this method testable. Let's start with the setup method for a unit test for this action method.

```
[TestFixture]
public class WhenBuildingUrl
{
    private HomeController sut;
    private string fullyQualifiedUrl;
    const string uri = "http://www.somewhere.com";

    [SetUp]
    public void Given()
    {
        sut = new HomeController();

        var httpContext = A.Fake<HttpContextBase>();
        var httpRequest = A.Fake<HttpRequestBase>();
        var httpResponse = A.Fake<HttpResponseBase>();

        A.CallTo(() => httpContext.Request).Returns(httpRequest);
        A.CallTo(() => httpContext.Response).Returns(httpResponse);

        var context = new ControllerContext(new RequestContext(httpContext,
            new RouteData()), sut);
        sut.ControllerContext = context;

        var fakeUri = new Uri(uri);
        A.CallTo(() => sut.ControllerContext.RequestContext.HttpContext.Request.Url)
            .Returns(fakeUri);

        fullyQualifiedUrl = string.Format("{0}/home/index", uri);
        sut.Url = A.Fake<UrlHelper>();
        A.CallTo(() => sut.Url.Action(A<string>.Ignored, A<string>.Ignored, null,
            A<string>.Ignored)).Returns(fullyQualifiedUrl);
    }
}
```

Code Listing 152: The test setup for BuildUrl

Let's explore some of the differences in setup from what we've seen so far in this chapter for faking **`UrlHelper`**:

- We are creating a fake of **`HttpRequestBase`**. **`HttpRequestBase`** has the **`Url`** property that is used for this code in the controller, **`Request.Url.Scheme`**. In order to make sure **`Request`** is not null when we run our unit test, we create a fake of it.
- Once we have a fake of HttpRequestBase created, we configure the call to **`sut.ControllerContext.RequestContext.HttpContext.Request.Url`** to return the URL we'll be building. We do this by populating the **`fakeUri`** variable with this line of code: **`var fakeUri = new Uri(uri);`**
- Next, we populate the **`fullyQualifiedUrl`** variable using the **`fakeUri`** with a string.Format call. We then create a fake **`UrlHelper`**. Again, a call to **`Url.Action`** in the controller uses **`UrlHelper`**. Once we have our fake created, we configure it to return the correct action using our **`uri`** variable.

The rest of the test setup is very similar to previous examples in this chapter.

Let's take a look at our test method.

```
[Test]
public void ReturnsTheCorrectUrl()
{
    var result = (ViewResult)sut.BuildUrl();
    var model = (BuildUrl)result.Model;
    Assert.That(model.Url, Is.EqualTo(fullyQualifiedUrl));
}
```

Code Listing 153: The assertion for testing BuildUrl

Here, we're asserting that the **`Url`** property of the model is equal to the **`fullyQualifiedUrl`** that we set up in our unit test setup method.

Summary

In this chapter, we've seen how to use FakeItEasy in conjunction with MVC's newest extensibility points and .NET's base classes to created fully testable controller action methods. We covered how to set up and work around most of Microsoft's big "sealed" classes to allow testability across all your MVC action methods. As you progress through writing tests for your controllers in your system, you'll start to write a LOT of code that looks the same. Feel free to experiment with putting the fake setup calls into a shared library, or, like we did at my current job, write some extension methods for the Controller class that live in your unit testing project.

In Closing

We've learned a good bit about FakeItEasy in this book. By this point, you should be comfortable using FakeItEasy in your unit tests for some of the most common scenarios you'll encounter. For any questions you still might have, or for other scenarios that were not covered in this book, check out the following links:

- FakeItEasy on GitHub: https://github.com/FakeItEasy/FakeItEasy
- FakeItEasy on PluralSite: http://www.pluralsight.com/courses/fakeiteasy
- FakeItEasy on StackOverflow: http://stackoverflow.com/questions/tagged/fakeiteasy