Object Oriented Development

Mocking with Moq Walkthrough

What does this walkthrough cover?

This walkthrough will introduce you to mocking, which is a Test Driven Development technique that we will use in conjunction with MS Test. We will be using the Moq framework.

How long will the walkthrough take to complete?

3-4 hours

What should you have already completed?

TDD and Collections

What do you need?

In order to complete this tutorial exercise you will need:

* Visual Studio 2013 Community Edition
* Notepad++
* Source Control
* Moq

What does this walkthrough cover?

* An overview of the concept of mocking frameworks and why we use them
* Mock objects
* Verifying behaviour
* Stubbing expected behaviour
* Matchers

Mocking Frameworks

Many modern Object Oriented languages have frameworks (libraries, if you prefer) that allow developers to utilise *Mocking* as part of the Test Driven Development Process. C# is no exception and whilst there are a number of mocking frameworks in C#, one of the most popular (and easiest to learn) is **Moq**.

The API can be found here: <https://github.com/Moq/moq4>

What is mocking and why would I want to use it?

Mocking frameworks typically provide a number of benefits for when we are applying the TDD process.

1. Testing Behaviour

**Question: What is it that XUnit really tests? Are there any limitations to the types of methods we can test with XUnit?**

**Answer: XUnit tests *State*, that is, the values of variables at a given point in time. XUnit *cannot* test void methods. This is a big limitation.**

Mocking frameworks allow developers to write tests that test the *behaviour* of their application. Behaviour here is defined to mean a *method call*. As such we can invoke (call) a method within our application and then observe *which methods it calls* in other objects, *how many* *times* it calls those methods, and *the parameters that are passed to them*.

1. Mock Objects

We can also use Moq to produce **Mock Objects**. Mock objects are *fake* versions of real objects; they are hollow with no defined functionality. Methods inside mock objects perform no logic, if they are defined to return a value, they will simply return the default value for that type; zero for primitives or null for objects.

**Question: Why is this useful?**

**Answer: Isolation. They allow us to isolate parts of our system from one another.**

**Scenario**

Imagine you have just finished developing a class via unit testing that must make a series of calls to a database.

All of your tests pass, your code works as expected.

**Question: If the database crashes, should all your tests still pass? Does the fact that the database is down now mean that your code is incorrect?**

**Answer: Your code has not changed. Your tests *should* still pass. So we need a way that we can isolate our test code from the database. We will achieve this via mocking.**

1. Dependency Injection

Finally, mocking encourages us to write code that makes use of Dependency Injection. Dependency Injection is the concept of *giving* an object all of the associated dependencies it needs to do its job, rather than have it make them itself.

Dependency injection allows us to write better Object Oriented code and ties in directly to Single Responsibility, Open/Close and Dependency Inversion, as we will see.

Setting up Moq

We will be building on the Project in the previous XUnit walkthrough- **TDDBookShopWalkthrough**.

If you have not done so already, add that code to SVN so that we may revert to the earlier version if we wish. However, there is no need to have a working version, as this walkthrough tests different functionality.

Moq is a library external to .Net – we need to install it. We can install it very easily from inside Visual Studio.

Right click your unit test project in the solution explorer and click ‘Manage NuGet Packages’. NuGet is a package manager that Visual Studio uses to install and maintain all external libraries that your solution relies on.

In the Nuget window use the search bar to find Moq and select it when it appears. Press the install button to install it to your unit test project.

Alternatively you can use the NuGet Package Manager Console to install libraries. The console can be found under the tools option in the menu bar. The command to install NuGet is:

Install-package moq

Ensure that the correct project (ie. The unit test project) is selected in the default project dropdown at the top of the console.

Moq Example

**Requirements**

We will be returning to our Book Shop application here, but adding some additional functionality.

**The book shop application will consists of a stock checker service that, before allowing a user to add a book to a basket, should make a call to the database and check whether that book is currently in stock.**

**The stock checker will take the unique ISBN of a book and make a call to a database to check how many are in stock.**

Note that the requirements talk about a database. You will not have covered how to make calls to a database from C# yet, so we will have to fake it using mocks (exactly how they are intended to be used).

Creating a Test Class

As mentioned above, Moq is designed to be used in conjunction with XUnit, not instead of it. As such, our tests will use much of the same syntax and ideas as our XUnit ones did.

Create a new class in UnitTests called **StockCheckerTest**

## Test 1

*The NumberInStock method of the StockChecker object should make a call to the ReadQuantity method of a DatabaseReader object once when called.*

Initially, our test will look something like this:

[Fact]

**public** **void** Test\_NumberInStock\_CallsReadQuantityMethodOfOurDatabaseReaderExactlyOnce\_WhenCalled()

{

//Arrange

string isbn = "ABC1234567";

StockChecker stockChecker = **new** StockChecker();

//Act

stockChecker.NumberInStock(isbn);

//Assert

mockDatabaseReader.Verify(r => r.ReadQuantity(It.IsAny<string>),Times.Once);

}

As we can see, the Arrange/Act/Assert structure of the test is maintained, the difference here is that instead of using one of XUnit **Assert** methods, we are instead using Moq’s **Verify** method as our actual test condition.

Within this test we are:

* Creating our StockChecker object, which we will be testing
* Making a call to the NumberInStock(isbn) method of the stockChecker object, passing it an isbn to check
* Verifying that, after we call NumberInStock(isbn), the method ReadQuantity() is called 1 time in the object mockDatabaseReader and being passed a string of some kind

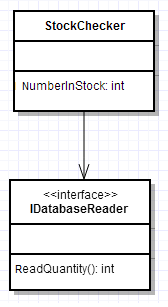
You can think of *Verify* like a line of dominoes. We are flicking the first domino (calling the *NumberInStock(isbn)* method) and watching what happens as a result. In this case, we are expecting a call to be made to the *ReadQuantity()* method in another object.

**Matchers**

***It.IsAny()* is part of a family of methods within Moq called matchers. They allow us to be very general in defining our test conditions and are a good way to, at least initially just test that a method is being called and allows us to ignore *what arguments* it’s being called with.**

**Some other matchers are *Is() and Match().* Custom matchers can be defined as so (here, using Book as an example).**

So, we are expecting a call to a *mockDatabaseObject*, however the object *mockDatabaseReader* does not yet exist, so there is no way that its method can be called. Let’s think about what our UML may look like:



Ultimately, we will need to write a class that is capable of running queries on a database, we are not however currently interested in the *DatabaseReader* class, so we will simply define it as an interface and worry about it later (when we are, in turn testing *it*).

As such, *DatabaseReader* will not have an implementation; it won’t do anything. We cannot make any actual calls to a database. This is not a concern however, as we are currently interested in the *StockChecker* object. As long as it does its job correctly, we should be happy. What’s its job? To make calls to the *DatabaseReader*. What the *DatabaseReader* does when we call it is immaterial.

mockDatabaseReader

Within our test, we are referring to an object called *mockDatabaseReader*, but where does it come from? We can’t instantiate an IDatabaseReader because it is an interface (even if it weren’t, we still wouldn’t *want* to).

We instead will make a *mock* object of type IDatabaseReader.

A **mock object** looks like a real object, as far as our program is concerned, but is actually just a hollow representation of an object. This has two benefits:

1. We can mock objects and interfaces that haven’t even been written yet and treat them as if they had been
2. We can ignore the actual implementation of an object that *has* already been written
   1. The main benefit here is that if our database reader *had* been written and we were to use it directly, it would make dozens and dozens of calls to our database every time we ran our unit tests. Something we don’t want. Using a mock allows us to ignore that implementation and pretend it does nothing.

So, we’ll alter our test to add a mock database reader object. Moq provides a build in method to create mocks that is straightforward to use.

[Fact]

**public** **void** test\_NumberInStock\_CallsReadQuantityMethodOfOurDatabaseReaderExactlyOnce\_WhenCalled()

{

//Arrange

string isbn = "ABC1234567";

Mock<IDatabaseReader> mockDatabaseReader = new M*ock*<IDatabaseReader>();

StockChecker stockChecker = **new** StockChecker();

//Act

stockChecker.numberInStock(isbn);

//Assert

mockDatabaseReader.Verify(r => r.ReadQuantity(It.IsAny<string>()),Times.Once);

}

**Rule of Thumb**

**The only *real* object that you should have inside your test class should be the one you are testing. Everything else should ideally be mock objects. This means that we are fully isolating the class under test of any influencing outside factors, meaning we can focus on the class we are actually developing right now.**

Compiling Our Test

Let’s move onto step 2 of the TDD process and make our code compile:

* Create an interface called IDatabaseReader
  + Give it a method called ReadQuantity(String id)

**Public interface** IDatabaseReader

{

**int** ReadQuantity(string id);

}

* Create a class called StockChecker
  + Give it a method called NumberInStock(String isbn)

**public** **class** StockChecker

{

**public** **int** NumberInStock(string isbn){

**return** 0;

}

}

Your test should now compile. Run it and watch it fail; make sure you are happy with *why* it fails at this point.

Making Our Test Pass

Step 4 of TDD asks that we get our test to pass in the simplest way possible. How can we do this?

One thing that may spring to mind is to create a new object of type IDatabaseReader inside our StockChecker object and then call its ReadQuantity method. As it stands, this will not work as IDatabaseReader is an interface (which we cannot instantiate). Even if IDatabaseReader were a class though and we were to instantiate it and call its method, our test would still not pass as it would be a *different* object to the one in our test (which we are watching with verify).

In order to make our test pass, we need to pass a reference to our mock object from our test, into the class we are testing. We will use **dependency injection** to achieve this.

We need to add a custom constructor to our StockChecker object and pass it an IDatabaseReader.

**public** **class** StockChecker {

**private** IDatabaseReader reader;

**public** StockChecker(IDatabaseReader reader){

**this**.reader = reader;

}

**public** **int** NumberInStock(String isbn){

**return** 0;

}

}

It is now impossible to create a new StockChecker object without also giving it a DatabaseReader object of some kind (we’re using dependency inversion to allow us to pass any class that implements DatabaseReader). Once we have created our StockChecker, it will then have a *permanent member variable* of type DatabaseReader, so whenever it needs one, it will use the one we gave it at creation.

These two objects are now *associated* with one another and will be until StockChecker no longer exists in memory.

Update our test to inject our StockChecker object a DatabaseReader:

[Fact]

**public** **void** Test\_NumberInStock\_CallsReadQuantityMethodOfOurDatabaseReaderExacltyOnce\_WhenCalled()

{

//Arrange

string isbn = "ABC1234567";

Mock<IDatabaseReader> mockDatabaseReader =

new M*ock*<IDatabaseReader>();

StockChecker stockChecker = n**ew** StockChecker(mockDatabaseReader.Object);

//Act

stockChecker.NumberInStock(isbn);

//Assert

mockDatabaseReader.Verify(r => r.ReadQuantity(It.IsAny<string>()),Times.Once);

}

Here we are injecting our *mock* DatabaseReader object into our StockChecker. When we run our program for real in a live environment, we will instead inject a real, functioning DatabaseReader and it will instead call a real object capable of making real calls to a database.

A Passing Test

Now, at last, we can make our test pass by amending NumberInStock(String isbn) in StockChecker to make the required call:

**public** **int** NumberInStock(string isbn){

reader.ReadQuantity("");

**return** 0;

}

Just pass an empty string into *ReadQuantity* for the time being. Run your test, it should now pass.

Recap

In writing our first Test, we have utilised the Moq framework to produce a mock object, used dependency injection to associate it with the object that needs it and finally we have verified that one of the mock objects methods is called (as a result of calling the method we are actually testing). We also used a matcher to make our test simpler.

Note that it is impossible to use the *verify* method with real objects, so we are in fact *required* to use mock objects when testing behaviour within our system.

Test 2

*When passed the ISBN number of a book, the stock checker method of book service should make a call to the database reader object exactly once, passing the ISBN number it was given to the database reader object.*

Test 3

*If 3 copies of the book exist within the database, then when the database object returns the value 3, the stock checker method should return true. - STUBBING*

Method Stubbing

Method stubbing is like mocking but for methods. When a mock object is created it is created without any information on what to do if one of its methods is called. In order for a mocked object to know it needs to have its methods stubbed. A method stub just tells a mocked object what to do or what to return when it’s method is called.

At this point we have successfully mocked a class and used it in a test. However, if we try to actually call the database reader in a new test it will fail.

public int NumberInStock(string isbn)

{

return \_reader.ReadQuantity(isbn);

}

[Fact]

public void Test\_NumberInStock\_ReturnsThree\_WhenThreeCopiesExistInTheDatabase()

{

//Arrange

string isbn = "ABC1234567";

int expected = 3;

Mock<IDatabaseReader> mockDatabaseReader = new Mock<IDatabaseReader>();

StockChecker stockChecker = new StockChecker(mockDatabaseReader.Object);

//Act

int actual = stockChecker.NumberInStock(isbn);

//Assert

Assert.Equal(expected, actual);

}

This test will fail because we are now trying to call the ReadQuantity method on the mocked object. Because the mocked object is a fake it doesn’t actually know what to do when that method is called.

The first thing we need to do though is create a concrete implementation of IDatabaseReader.

public class DatabaseReader : IDatabaseReader

{

public int ReadQuantity(string id)

{

return 0;

}

}

Now we can stub the ReadQuantity method and pass the mock object into the stockchecker constructor.

[Fact]

public void Test\_NumberInStock\_ReturnsThree\_WhenThreeCopiesExistInTheDatabase()

{

//Arrange

string isbn = "ABC1234567";

int expected = 3;

Mock<IDatabaseReader> mockDatabaseReader = new Mock<IDatabaseReader>();

mockDatabaseReader.Setup(r => r.ReadQuantity(It.IsAny<string>())).Returns(3);

StockChecker stockChecker = new StockChecker(mockDatabaseReader.Object);

//Act

int actual = stockChecker.NumberInStock(isbn);

//Assert

Assert.Equal(expected, actual);

}

This test now passes. If you follow the test through the debugger you will see that the real ReadQuantity method never actually gets called. The mock object sees that we want to call the ReadQuantity method and just returns us 3 because that’s what we set up the object to do.

It is ok that we never call the real ReadQuantity method because we are not testing the DatabaseReader, we are only interested in what the StockChecker is doing.