Mocking with Moq

🚀 Intro

When writing a test, quite often you want to only test one particular class and method. But that method might call a dependency. A mocking library allows you to create "*Mocks*" (fake objects). Those objects allow you to set its property values, specify its parameters, and return its values on the method calls.

Example:

```
// Create the mock
var mock = new Mock<IMockTarget>();

// Instruct the mock to do something
mock.SetupGet(x => x.PropertyToMock).Returns("FixedValue");

// Check that the configuration works
Assert.AreEqual("FixedValue", mock.Object.PropertyToMock);

// Verify that the mock was invoked
mock.VerifyGet(x => x.PropertyToMock);
```

Whilst this example shows the steps involved in using the mock, it is important to remember that it doesn't actually test anything, other than that the mock has been setup and used correctly. An actual test that makes use of a mock will supply the mock to the system that is to be tested.

📝 Moq Basics

1. Creating a Mock

Let's create our first Mock with the following code:

```
// Create the mock
var author = new Mock<Author>();
```

As you can see from the code above, mocking an object is simple. Simply use `Mock<>`, passing the type you want to mock. When you want to use the mocked object, you use the `.Object()` method.

📝 2. Add data to a mock

To configure a mock to return data, you will need to use the `.Setup()` and `.Return()` methods.

🛠 2.1 Setup the properties

```
// Set properties
author.Setup(x => x.Name).Returns("jon");

// Get the properties
var name = author.Object().Name;  // 'jon'
```

🛠 2.2 SetUp the Methods

Once it is created you need to set up the ideal expectations for your mock. To do so, we use the method `Setup()` like this:

- without parameters

```
mock.Setup(x => x.DoSomething()).Returns(true);
var result = mock.Object.DoSomething(); //true
```

- With explicit parameters:

```
var card = new Card("owner", "number", "CVV number");
paymentServiceMock.Setup(p => p.Charge(114,card)).Returns(true)
```

- With generic parameters:

  We can also use the helper `It`, which will allow us to pass to the method `Charge()` any values of a certain data type:

```
var card = new Card("owner", "number", "CVV number");
paymentServiceMock.Setup(p => p.Charge(It.IsAny<double>(),
card)).Returns(true)
```

- Verify whether a particular method or a property was called:

```
mockObj.Verify(t => t.GetPublicationDate()); // Defaults to
Times.AtLeastOnce
mockObj.Verify(t => t.GetPublicationDate(), Times.Never);
mockObj.Verify(t => t.GetPublicationDate(), Times.Once);
mockObj.Verify(t => t.GetPublicationDate(), Times.Exactly(2));
mockObj.Verify(t => t.GetPublicationDate(), Times.Exactly(3));
```

- Throwing exceptions

```

```

```
mockStockChecker.Setup(x => x.IsProductInStockAsync("")).Throws(new
NullReferenceException());
```

📝 3. Mock base class methods

you have a method in the base class that has been overridden in the mocked type, and you need to mock the base version of the overridden method only, you can draw on CallBase.

```
public abstract class RepositoryBase
{
    public virtual bool IsServiceConnectionValid()
    {
        //Some code
    }
}
public class AuthorRepository : RepositoryBase
{
    public void Save()
    {
        if (IsServiceConnectionValid())
        {
            //Some code
        }
    }
}
```

Now suppose we want to check if the database connection is valid. However, we may not want to test all the code inside the IsServiceConnectionValid method. For instance, the IsServiceConnectionValid method might contain code that pertains to a third-party library. We would not want to test that, right? Here is where the CallBase method in Moq comes to the rescue.

In situations like this, where you have a method in the base class that has been overridden in the mocked type, and you need to mock the base version of the overridden method only, you can draw on CallBase. The following code snippet shows how you can create a partial mock object of the AuthorRepository class by setting the CallBase property to true.

```
var mockObj = new Mock<AuthorRepository>(){CallBase = true};
mockObj.Setup(x => x.IsServiceConnectionValid()).Returns(true);
```

🚧 Basic Example

Imagine a Class called `OrderHandler` that needs two interfaces (`IStockChecker` and `IOrderRepository`) that are injected in the constructor:

```
[Fact]
public void GivenInsufficientStock_DoNotCreateOrder()
{
    // Arrange
    var mockStockChecker = new Mock<IStockChecker>();
    var mockOrderRepository = new Mock<IOrderRepository>();

    mockStockChecker.Setup(x => x.IsProductInStock()).Returns(false);

    var sut = new OrderHandler(mockStockChecker.Object,
mockOrderRepository.Object);

    // Act
    sut.ProcessOrder();

    // Assert
    mockOrderRepository.Verify(x => x.CreateOrder(It.IsAny<int>()),
Times.Never);
}
```

As you can see, the dependencies are **mocked out**. The real implementation would have called out to a database, but the class we're testing doesn't know or care about this - it just cares that whatever "`IsProductInStock`" is, we get a boolean back. It doesn't care where that comes from, as that's not it's responsibility. What is it's responsibility is that if `IsProductInStock` returns true, the class we're testing will execute `ProcessOrder`. And likewise, if it returns false, it does not execute `ProcessOrder`.

🚧 Real Example

Imagine you have a class that deals with newsletter sign-ups. You want to test the method, however, you do not want the system to send any emails. To top it off, if you tried to call the default SendSmtp() method, an exception would be thrown and your test would fail. In our example, the code for this scenario looks like this;

```
public class MyClass
{
     public bool MyMethod()
     {
        // Logic we want to test)
        var smtpSender = new SmtpSender();
        sender.SendSmtp("message");

        // The code will thrown an exception, meaning we can't unit test
it!!!
        return true;
      }
}
```

When you find yourself in these types of situations where your code can't be unit tested, mocking comes into play. In most instances to make things testable, you will need to slightly re-architect your code. Let us see how

to do that! Let us rewrite the code above so that it can be unit tested. The main thing we need to do is extract the call to the SMTP method into an interface. We can then mock that interface and inject it into the class. This will allow us to mock the part that can not otherwise be tested. To solve this problem, first, we need to wrap the SMTP's functionality into an interface, like so:

```
public interface ISmtpSender
{
   bool SendSmtp(string message);
}

public class SmtpSender : ISmtpSender
{
   public bool SendSmtp(string message)
   {
      // code
   }
}
```

We can now rewrite myClass so that it is testable. We do this by passing in the SMTP interface into the class via the constructor, like so:

```
public class MyClass
{
     ISmtpSender _sender;

     public MyClass(ISmtpSender sender)
     {
          _sender = sender;
     }

     public bool MyMethod()
     {
        // Logic we want to test)
        _sender.SendSmtp("message");
        return true;
     }
}
```

The class is now written in such a way that the part of the system that was previously untestable can now be mocked. This cane achieve this by mocking the call to ISmtpSender.SendSmtp() and adding a mocked return value of true to it:

```
[Test]
public void Test()
{
    var mockSmtpSender = new Mock<ISmtpSender>();
```

```
    mockSmtpSender.SetUp(x => x.SendSmtp(It.IsAny<string>
()).Returns(true);

    var myClass = new MyClass(mockSmtpSender.Object());
    myClass.MyMethod();
}
```

We can execute a unit test against this updated code and an exception will NOT be thrown. Calling myMethod() will not throw an exception anymore! We pass the mocked object into the class, which has been configured to return true if anything calls the SendSmtp() method. The test will now run to completion without failing. Allowing you to test and add alterations as needed.

We have successfully refactored our code and made it testable by building our class against an interface. Mocking the interface to allow us to test code that was previously untestable!

Links

- Tutorial
- Moq Quickstart
- Mocking Theory