# Super High-Speed Intro to Mocking and Rhino Mocks

## Define it!

For the purpose of this essay a mock is a use of object oriented polymorphism to create an alternate implementation of a class or interface. Basically a mock is just a fake version of an object.

A mock is used for two things

1. **Stub:** Stand in for a service and provide alternate, simpler method implementations. An example would be to substitute for an object that would normally query the database – instead of querying the database the stub object will simply return a canned response.
2. **Spy:** Detect whether an object was used in a certain way. An example would be when testing an object that decides whether to call ui.NotifyUser() depending on some criteria. A spy can stand in for the ui object and call tell you if the NotifyUser() method was called.

## Why Would You Need A Mock?

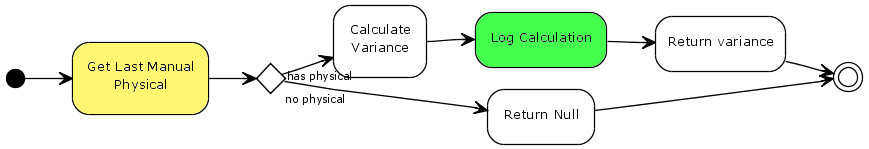
Consider a domain where a physical measurement is compared to the value on the books to calculate the difference (variance):

[[Code Link]](http://code.google.com/p/gim-projects/source/browse/Tutorial.RhinoMocks/app/Tutorial.RhinoMocks.Tests/Domain.cs)

The only logic here is in the VarianceCalculator.GetVariance method so let’s test that.

For any action there are two things that can be checked: the output, and any side-effects. So let’s write down all the conditions and side-effects that we expect.

1. If the contents object does have a manual physical then the method should return the correctly calculated quantity
2. If the contents object does have a manual physical then the method should log the calculation.
3. If the contents object does not have a manual physical then the method should return null
4. If the contents object does not have a manual physical then the method should not log anything.



Notice that an external service (IPhysicals) is queried to check if we need a physical. We do not want to actually have to set up and query the database so we need something that performs the duties of a **stub**. Also notice that we need to check whether the calculation results were passed off to a logger object. Going on our definitions from above, this requires a **spy**.

## Our Tests

Given that we have four points above we should write four tests. Here they are written in an abstract class that abstracts away the need to actually create mocks. We will create these in the next two sections.

[[Code Link]](http://code.google.com/p/gim-projects/source/browse/Tutorial.RhinoMocks/app/Tutorial.RhinoMocks.Tests/VarianceCalculatorTests.cs)

## Doing It Manually

In the interest of going slow, let’s first implement the above test fixture manually.

**MOCKS CANNOT DO ANYTHING THAT YOU CANNOT DO YOURSELF MANUALLY!**

If you do not need any further convincing then you can skip to the next section.

When you manually create an alternate implementation the resulting objects are traditionally called “Fakes”. Here is a fake implementation of IPhysicals that can stub out calls to physicals.GetLastFor()

        public class FakePhysicalsStub : IPhysicals {  
                private Quantity \_lastPhysical;  
                public FakePhysicalsStub(Quantity lastPhysical) {  
                        \_lastPhysical = lastPhysical;  
                }  
                public Quantity GetLastFor(Contents contents, PhysicalType physicalType) {  
                        return \_lastPhysical;  
                }  
        }

And here is a fake implementation of an IEventLogger spy:

        public class FakeEventLoggerSpy : IEventLogger {  
                private IList<string> \_messagesLogged = new List<string>();  
                public void Log(string message) {  
                        \_messagesLogged.Add(message);  
                }  
                public bool WasLogged(Regex expectedMessageMatcher) {   
                        return \_messagesLogged.Any(m => expectedMessageMatcher.IsMatch(m));  
                }  
        }

And here is how you would use these:

[[Code Link]](http://code.google.com/p/gim-projects/source/browse/Tutorial.RhinoMocks/app/Tutorial.RhinoMocks.Tests/ManualFakes_VarianceCalculatorTests.cs)

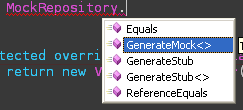
## Getting Rhino.Mocks to Do It for You

Creating fakes manually is not very difficult but it does require some thought. In addition you end up having to maintain the fakes as well as your actual implementations when making any changes to the underlying types. On the other hand, creating manual fakes will always be more powerful and flexible than an automatic alternative so do not be afraid to use them when needed.

That being said, when properly used Rhino Mocks when properly used should be able to cover 99% of scenarios. I have never really encountered a situation where its abilities proved insufficient. It is a tool that has been around for some time and offers several different API Interfaces. This tutorial targets the version of the API as of Rhino.Mocks 3.6 – while improvements are possible, there have not been any truly major changes in the last few versions and this API style can probably be considered final.

### Mock Creation

First things first; how to use Rhino.Mocks to create our stub and spy instances? The first thing to understand is that all classes created by Rhino.Mocks can naturally be both stubs and spys. The easiest way to create them is to use the MockRepository static class:



This class has two methods of consequence, *GenerateMock<T>()* and *GenerateStub<T>()*. There is very little [difference between these two](http://ayende.com/Wiki/Rhino+Mocks+3.5.ashx#Thedifferencebetweenstubsandmocks) and you can feel perfectly safe using only *GenerateStub<T>().* Calling *MockRepository.GenerateStub<IEventsLogger>() w*ill return an object of type *IEventsLogger*.

### Stubbing

This mock object can be told to act as a stub for certain method calls:

\_physicals = MockRepository.GenerateStub<IPhysicals>();

\_physicals.Stub(x => x.GetLastFor(Arg<Contents>.Is.Anything, Arg<PhysicalType>.Is.Anything) ).Return(lastManualPhysical);

This API uses extension methods and LINQ Expressions to allow you to specify which method you want to stub and what to return. As the above code implies, it also allows you to constrain the stub based on certain types of inputs.

\_physicals.Stub(x => x.GetLastFor(Arg<Contents>.Is.Null, Arg<PhysicalType>.Is.Equal(PhysicalType.Automatic))).Return(someOtherPhysical);

Is legal and works pretty much like you would expect. Other constraints are also available. There are also several shortcuts. For example the above can be expressed as

\_physicals.Stub(x => x.GetLastFor(null, PhysicalType.Automatic).Return(someOtherPhysical);

Since all arguments have to be matched exactly. Also the first stub (where any arguments are acceptable) can be more quickly expressed as:

\_physicals.Stub(x => x.GetLastFor(null, null)).IgnoreArguments().Return(lastManualPhysical);

Several other syntaxes are also available, but as of Rhino.Mocks 3.6 these are the only ones that are preferred.

### Spying

Now that we have discussed stubbing, let’s discuss the syntax for creating spy objects. As stated before, all objects created with Rhino.Mocks can be both stubs and spys so their creation is exactly the same. We only need to learn the syntax for how to verify whether a method was called or not. This is achieved with the AssertWasCalled() and AssertWasNotCalled() extension methods.

For some strange reason, the same *Arg<T>.Is* syntax does not exist for these – instead to do anything but simple pattern matching we need to provide a second parameter specifying constraints explicitly. This is a little annoying, but not that bad and doubtless will be fixed in upcoming versions. So, to specify that our IEventsLogger.Log() method was called we need to do this:

\_logger.AssertWasCalled(x => x.Log(null), c => c.IgnoreArguments());

Or to specify that it was not called:

\_logger.AssertWasNotCalled(x => x.Log(null), c => c.IgnoreArguments())

If you want a little more finely grained control over the argument constraints you can do that too. For example the following asserts that Log() was called and that the input was not null or empty:

\_logger.AssertWasCalled(x => x.Log(null), c => c.Constraints(Is.Matching<string>(m => !String.IsNullOrEmpty(m))));

Of course, as before, if you know the exact argument that you want to check for you can do it more easily:

\_logger.AssertWasCalled(x => x.Log(“this exact message was called”));

Here is the full implementation of the base VarianceCalculatorTests:

[[Code Link](http://code.google.com/p/gim-projects/source/browse/Tutorial.RhinoMocks/app/Tutorial.RhinoMocks.Tests/RhinoMocks_VarianceCalculatorTests.cs)]

### Other Stuff

You have now learned virtually everything you need to know to effectively use Rhino.Mocks. A couple other occasionally used syntaxes are:

\_physicals.Stub(x => x.GetLastFor(null, Arg<PhysicalType>.Is.Anything)).Throw(new ArgumentNullException ());

Which causes (you guessed it) an ArgumentNullException to be thrown when GetLastFor() is passed a null contents object.

Also, you can pass in a delegate to be executed when a stubbed method is called. This is primarily useful when doing integration and benchmark tests or when working with a timer.

\_physicals.Stub(x => x.GetLastFor(null, PhysicalType.Automatic)).IgnoreArguments().Do(

new Func<Contents, PhysicalType, Quantity>((c, t) => {

Console.WriteLine("Method Called With {0}, {1}", c, t);

return null;

})

);

A situation where you are finding yourself having to do something like this might be an indicator that you should just create a fake manually.

It is also possible to create mocks of concrete classes rather than just interfaces. However, this can be a little tricky since there is no way for Rhino.Mocks to stop the constructor from executing. You must therefore either provide a parameter-less protected constructor or pass in objects to be used.

public class MyClass {

string \_dependency;

public MyClass(string dependency) {

\_dependency = dependency;

}

}

//

MyClass x = MockRepository.GenerateStub<MyClass>(“”);

A word of caution however, if you find yourself really needing to mock a concrete object perhaps you might want to consider extracting an interface, or including the object in your test.

And that’s it. There is a lot of other stuff in Rhino.Mocks but it is almost guaranteed that you don’t need to know it as the above will cover nearly every single testing scenario.

## Sample Code and on Testing Style

The sample code from this tutorial is provided [here](http://code.google.com/p/gim-projects/source/browse/#svn/Tutorial.RhinoMocks). Please note that there are three sets of tests. [VarianceCalculatorTests.cs contains the base class](http://code.google.com/p/gim-projects/source/browse/Tutorial.RhinoMocks/app/Tutorial.RhinoMocks.Tests/ManualFakes_VarianceCalculatorTests.cs) which defines the tests used above. [ManualFakes\_VarianceCalculatorTests implement the tests using manually built fakes](http://code.google.com/p/gim-projects/source/browse/Tutorial.RhinoMocks/app/Tutorial.RhinoMocks.Tests/ManualFakes_VarianceCalculatorTests.cs) and [RhinoMocks\_VarianceCalculatorTests implements them using Rhino.Mocks](http://code.google.com/p/gim-projects/source/browse/Tutorial.RhinoMocks/app/Tutorial.RhinoMocks.Tests/RhinoMocks_VarianceCalculatorTests.cs).

Although it is ultimately a matter of preference, it should be noted that neither of these is done in the currently predominant testing style – [Behavior Driven Development (BDD)](http://en.wikipedia.org/wiki/Behavior_Driven_Development). Another series of tests is provided inside the BDD directory to demonstrate one of the ways of doing this using BDD.

Happy Coding!