Rhino Mocks

Rhino.Mocks is a dynamic mock object framework for the .Net platform. Its purpose is to ease testing by allowing the developer to create mock implementations of custom objects and verify the interactions using unit testing.

Background:

Rhino.Mocks is based on EasyMock.Net and released under the same BSD license. It's free for use and modification for free and commercial software. Rhino.Mocks is an attempt to create easier way to build and use mock objects and allow better refactoring support from the current tools. It's a hybrid approach between the pure Record/Replay of EasyMock.Net's model and NMock's expectation based model.

I created Rhino.Mocks because I wanted better support from refactoring tools such as ReSharper and because I don't like the way NMock handle parameterized methods (you need to pass fake constraints to get it to recognize the correct method) and the lack of extensibility [I required hacks to get to the result that I wanted].

A mock is an object that:

- Takes the interface of another object, thus allowing is to be substituted for the original object during testing.
- Allows **Expectations**, **Constraints** and **Callbacks** to be setup, specifying how the tested object *should* interact with the mocked object.
- Fails the test if any of the above conditions is violated.
- Can act as a stub, allowing the test to specify what would be returned or if an exception should be thrown.

Rhino.Mocks features:

- Supports several models of placing conditions on the mocked objects, all of which are verified on the fly during test execution. This allows testing the *interactions* between objects and fails as soon as the test deviates from the specified conditions. This has the advantage of giving the exact point of failure.
- Mocked implementations of the objects are generated on the fly which avoids a code generation during builds.
- Clear error messages for failing conditions.
- Flexible way to specify conditions using **Expectations**, **Constraints** and **Callbacks** which is type safe and easily refactorable by the current tools.

Rhino.Mocks limitations:

- Currently works only on interfaces and classes which inherit MarshelByRef.

Usage examples:

[This is a direct translation of NMock's examples, in order to show the difference between the libraries]

Here is the code that we want to test:

```
// The interface that's to be mocked.
// Rhino.Mocks can currently mock only interfaces and classes which inherit from
MarshelByRef
interface ISomething
{
	public virtual void Eat(string food, int numChews, bool
eatWithMouthClosed);

	public virtual void Nap();
}

// the class under test.
class ClassUnderTest
{
	public ISomething something; // an external object.

	public ClassUnderTest(ISomething something)
	{
		this.something = something;
	}

	// method under test.
	public void DoYourStuff()
	{
		if (someComplicatedLogic())
		{
			// interaction with external object.
			something.Eat("cheese", 22, true);
```

```
            }
            else
            {
                // anotherinteraction
                something.Nap();
            }
        }
}
```

Here is a simple test when I specify exactly what I want to get for the method, I set an **expectation** that the Eat() method will be called with exactly those three arguments. Notice that I'm using strongly typed objects throughout the test, no need for strings. If the method is called with different argument the test will fail immediately, if the method is not called, the test will fail to Verify(). Calling a method that was not explicitly expected will cause the test to fail immediately.

```
[Test]
public void SomethingTest()
{
        // Create mocked control
        MockControl mockSomething =
MockControl.CreateControl(typeof(ISomething));

        // Notice that we place the mocked instance in a variable, instead of using
        // MockInstance member, this so we can record the actions of the mocked
object
        ISomething something = (ISomething)mockSomething.MockInstance;

// pass mock into ClassUnderTest.
        ClassUnderTest classUnderTest = new ClassUnderTest(something);
```

```
        // EXPECTATIONS : how we expect the ClassUnderTest to deal with mock.
        // any deviation from the parameters would fail the test
        something.Eat("cheese",22,true);

        // REPLAYING: Finished recording the interactions, now I'm replaying it.
        mockSomething.Replay();

        // EXECUTE : any unexpected calls on the mock will fail here.
        classUnderTest.DoYourStuff();

        // VERIFY: checks that all condition have been met and that the replaying
matching the recording.
        mockSomething.Verify(); // note: no Assertions.
}
```

Suppose we are interested only in the first parameter, and want to ignore the other two. I can do that by setting a **constraint** on the method and specifying the constraints for each argument. Rhino.Mocks will verify that the Eat method was called and that the constraints validation passed or fail immediately.

```
[Test]
public void SomethingTest()
{
        // Create mocked control
        MockControl mockSomething =
MockControl.CreateControl(typeof(ISomething));

        // Notice that we place the mocked instance in a variable, instead of using
```

```
        // MockInstance member, this so we can record the actions of the mocked
object
        ISomething something = (ISomething)mockSomething.MockInstance;
        // pass mock into ClassUnderTest.
        ClassUnderTest classUnderTest = new ClassUnderTest(something);

        // EXPECTATIONS : how we expect the ClassUnderTest to deal with mock.
        // I'm passing empty values here because they are ignored when I'm
setting
        // constraints.
        something.Eat(null,0,false);

        // CONSTRAINTS: The first argument equal to cheese, and ignore the
other two
        mockSomething.Constraints(new IsEqual("cheese"),new IsAnything() ,
new IsAnything());

        // REPLAYING: Finished recording the interactions, now I'm replaying it.
        mockSomething.Replay();

        // EXECUTE : any unexpected calls on the mock will fail here.
        classUnderTest.DoYourStuff();

        // VERIFY: checks that all conditions have been met and that the
replaying matching the recording.
        mockSomething.Verify(); // note: no Assertions.
}
```

Suppose I want even finer control, perhaps I need to validate several arguments together, or I need to perform some action that is important for the test[1]. I can setup a callback for the method which will be called when the method is called, with the arguments the mocked method was called with. If you need to fail the test from your callback, throw an exception, etc.

```
// DELEGATE: Need a delegate to pass as the callback
delegate void Eat(string food, int numChews, bool eatWithMouthClosed);

// CALLBACK METHOD: this method will be called with the arguments that were
// passed by the class under test.
private void EatValidation(string food, int numChews, bool eatWithMouthClosed)
{
        Assert.FoodCanBeChew(food,numChews);
}

[Test]
public void SomethingTest()
{
        // Create mocked control
        MockControl mockSomething =
MockControl.CreateControl(typeof(ISomething));

        // Notice that we place the mocked instance in a variable, instead of using
        // MockInstance member, this so we can record the actions of the mocked
object
```

[1] This is a very touchy subject, and is wide open for abuse. Try to avoid as much as possible having anything but validation in your callback methods.

```
        ISomething something = (ISomething)mockSomething.MockInstance;
        // pass mock into ClassUnderTest.
        ClassUnderTest classUnderTest = new ClassUnderTest(something);

        // EXPECTATIONS : how we expect the ClassUnderTest to deal with mock.
        // I'm passing empty values here because they are ignored when I'm
setting
        // a callback.
        something.Eat(null,0,false);

        // CALLBACK: Setting a callback will cause it to be called when the method
is called
        // and the method's arguments will be passed to the callback
        mockSomething.Callback = new Eat(EatValidation);

        // REPLAYING: Finished recording the interactions, now I'm replaying it.
        mockSomething.Replay();

        // EXECUTE : any unexpected calls on the mock will fail here.
        classUnderTest.DoYourStuff();
        // VERIFY: checks that all callback finished successfully and that the
replaying matching the recording.
        mockSomething.Verify(); // note: no Assertions.
}
```

So, that was the usage sample, naturally this is quite contrived, so try to think how you would apply it in your code, there are quite a bit of benefit to doing so. And now for the detailed view:

Working with Rhino.Mocks:

Rhino.Mocks uses the Record / Replay model to setup the conditions for the test. Any action on the mocked object will be recorded until the MockControl's Replay() method is called. After each call, you can setup more information about the call: Return value, if and what to throw, constraints and callbacks. You set these conditions by calling the methods on the MockControl (Returns, Throws, Constraints and Callback). Most methods supplies overload that allows you to setup returning or throwing for multiply calls of the same method. The Replaying phase begins when the Replay() methods is called and any method call that is unexpected or with wrong arguments will cause the test to fail immediately. Calling Verify() at the end of the test will cause the test to fail if an expected method was not called.

The Last Method:

Most actions in Rhino.Mocks are done on the last method call, for instance, here is how to tell Rhino.Mocks to throw an exception on a method call:

```
complexObject.DoAction();
mockComplexObject.Throws = new IOException("Hard Disk Error");

comlexObject.Repair();
//Throw for three times, then return Ok message
mockComplexObject.ThrowsFor(new BadData(),3);
mockComplexObject.Returns = Message.Ok;
```

Rhino.Mocks ensure that all the requirements of a method are satisfied before the next method can be recorded, this means that for methods that return values, a return value or an exception to throw need to be set. You can setup

several behaviors for a single method. The same method but with different arguments mean a different method call, as far as Rhino.Mocks is interested.

Return Value:

A method that returns a value *must* define either Returns or Throws or their equivalents (ReturnsFor(), ThrowsFor(), AlwaysReturns, AlwaysThrow, SetupResult() or ExpectAndReturn()).

Tips & Tricks:

- If you want just a stub, you can use MockControl.CreateNiceControl(), which will not throw on unexpected calls, and would always return the default value. (Null or zero, usually).
- If you want to check the *ordering* of the calls, using MockContrl.CreateStrictControl().
- To mock a get property, you need code similar to the one below, it's easier and more readable to use the convenience methods (see separate section);

```
object dummy = someObject.MyProperty;
```

Constraints:

Rhino.Mocks currently contains a large subset of the constraints that appears in NMock. Their usage is identical.

Callbacks:

A callback need to match the last method signature, but is free to have a different return type (or no return type, of course). The return value of the callback is *ignored*, so don't expect to return some value from the callback and

get it from the mocked method. This is by design and is meant to reduce the chance that the callbacks will be abused.

Naming Conventions:

I'm currently using the following naming convention, for the MockControls, mockVariableName, for the mocked object, variableName. I'm not sure if I'm very happy with this. So if you've a better suggestion, I'm certainly open for it.

Convenience Methods:

There are several convenience methods in Rhino.Mocks, (ExpectAndReturn(), SetupResult()) which makes life a bit easier, instead of writing code that look like:

```
object dummy = someObject.MyProperty; //Record the get method
mockSomeObject.Returns = someVal; // setup the return value
```

You can write this code:

```
mockSomeObject.ExpectAndReturn(someObject.MyProperty,someVal);//Record t
he get method and return someVal when it's called.
```

You can also use SetupResult() to always return the same value for the specified method call.

Mocking Classes:

Rhino.Mocks *can* mock classes if they inherit from MarhsalByRef, it's not a recommended approach. Use of interfaces is highly recommended.

Gotcha:

- A method call is consider equal to another if they are the same method with *the exact same* arguments. This means that something("foo") and something("bar") are *not* considered to be the same method.
- Validating a method call and *expecting* a method call are two distinct subjects. A call is validated (using expectations, constraints or callbacks) only if the call is expected. You can setup the number of expected calls using the ReturnsFor(), CalledFor() and ThrowsFor() methods.
- Callback will be called for each invocation of the method, and *all* validation relies on whatever or not the callback throws. If the callback does not throw, the method is considered validated.
- The return value of callbacks is ignored, the only way to set up a return value is to use the Returns derivatives, or the convenience methods for them.
- Don't forget to call the Replay() methods

Enjoy using Rhino Mocks,

Ayende Rahein