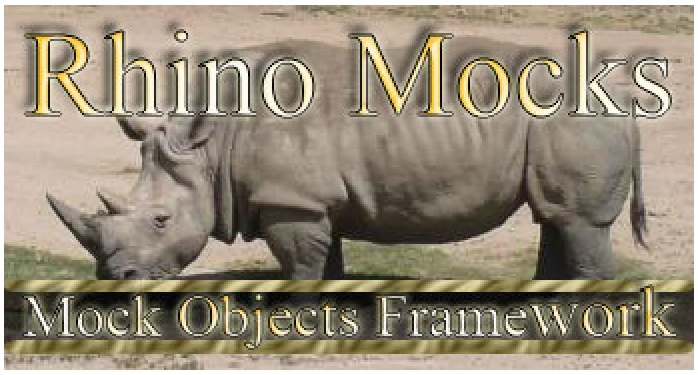

Rhino Mocks:

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.

"We ape, we mimic, and we mock."
~Laurence Olivier, Sir quotes

What is a mocked object? Mock Object.

Mock objects take stubs to an extreme and are used to set and verify expectations on interactions with model code. We found that this approach to unit testing has the side effect of making your code conform more to the Law Of Demeter.

The need for mock objects:

A Mock Object:

1. is easily created
2. is easily set up
3. is quick
4. is deterministic
5. has easily caused behavior
6. has no direct user interface
7. is directly queriable

You may want to use a Mock Object if the:

1. real object has non-deterministic behavior
2. real object is difficult to set up
3. real object has behavior that is hard to cause (e.g., network error) (similar to 1)
4. real object is slow
5. real object has (or is) a UI
6. test needs to query the object, but the queries are not available in the real object (e.g., "was this callback called?") (This needs a Mock Object that has *more* stuff than the real object; the other cases usually require a stub that is far smaller than the real object).

7. real object acts "normal" most of the time, but once a month (or even less often) it does something "exceptional". We want Unit Tests to make sure the rest of the system does the Right Thing whether or not the object is acting "normal" or "exceptional". (Is this the same as #2 ?)
8. real object does not yet exist

Mock objects are used in Unit Testing to isolate tests. Mock Objects allows you to test just the specific class / method without testing (and setting up, and tearing down, etc) the entire environment. Here is a simple example of using mock objects:

Let's say we have the following interface:

```
public interface ICarRental
{
   ICar[] FindCarsForMaxPrice(Money maxPrice);
   void RentCar(ICar carToRent, IPayable payment);
   void ReturnCar(ICar returnedCar);
}
```

I want to write a test that makes sure that a damaged car cannot be returned without some action taken. For the purpose of discussion, I want a DamagedCarException() to be thrown in this case. Here is the test I want to write, but I've some problems with it.

```
[Test]
[ExpectedException(typeof(DamagedCarException),"Car damaged! Need more money!")]
public void DamagedCarReturnedWouldThrowException()
{
   ICar rentedCar = // How do I create this class?
   ICarRental avis= new AvidCarRental();
   // How do I make the IsDamaged property return true ?
   // rentedCar.DoDamage() -- There is no such method, and I don't want to add one.
   avid.ReturnCar(rentedCar);
}
```

How am I going to test that? Creating a real instance of car require database access, so while I *could* do it, I want something better. I don't want to hit the database every time that I run my unit tests. I've been vigilant, and the classes talks via interfaces, which means that I can create a new class MockedCar, so I can create an instance of that and it will return true when the IsDamaged property is called. There is just one problem with this, the ICar interface contains 40 methods and 60 properties, dealing with all the car's details that I need (HasRadio, GasTankCapacity, Play Radio(), PreventAccident(), etc), this is quite a lot of work, in order to create one little test. But I'm diligent and I want to have tests for all of my code. So here is my test after I written by hand the MockedCar class:

```
[Test]
[ExpectedException(typeof(DamagedCarException),"Car damaged! Need more money!")]
public void DamagedCarReturnedWouldThrowException()
{
 ICar rentedCar = new MockedDamagedCar();
 ICarRental avis= new AvisCarRental();
 avis.ReturnCar(rentedCar);
}
```

Well, the test itself is easy to read, but it wasn't easy to write. I worked too hard on this test. Experience has thought the industry that what isn't simple simply wouldn't be[1]. So I was diligent on this case, but *next* time, would I be? If writing a test for this *tiny* bit of functionality is going to take me half an hour of doing grunt work, maybe it's not *that* important. I could skip this and go write *important* code.

This is the point where Mock Objects Frameworks enter the scene.

"Once," said the Mock Turtle at last, with a deep sigh, "I was a real turtle."
~Lewis Carroll, Alice in Wonderland

Mock Objects Frameworks:

A mock objects framework is a tool to ease testing by doing all the grunt work for you. There are several mock objects frameworks for .Net & Java. Probably the most common framework for creating mock objects in .Net is NMock, a port to .Net of Java's jMock. The above test using NMock would be written like this:

```
[Test]
[ExpectedException(typeof(DamagedCarException),"Car damaged! Need more money!")]
public void DamagedCarReturnedWouldThrowException()
{
 DynamicMock mockedCar = new DynamicMock(typeof(ICar));
 mockedCar.ExpectAndReturn("IsDamaged",true);
 ICarRental avis= new AvisCarRental();
 avis.ReturnCar((ICar)mockedCar.MockInstance);
}
```

Simple, powerful, and to the point, isn't it?
The problem starts when you move beyond this simple tests, let's take the example one step further, we're now trying to test that when car was returned in a good condition, then the car rental get its money. Here is the test using NMock:

```
[Test]
public void RenterPaysForCar()
{
 DynamicMock mockedCar = new DynamicMock(typeof(ICar));
 DynamicMock mockPerson = new DynamicMock(typeof(IPerson));
 mockCar.SetupResult("CurrentPrice",115);
 mockCar.SetupResult("RentedFrom",DateTime.Now.AddDays(-7));
 mockCar.SetupResult("RentedBy",mockPerson.MockInstance);
 mockCar.SetupResult("ReturnToParkingLot",new IsTypeOf(int),new IsAnything());
 ICarRental avis= new AvisCarRental();
 mockPerson.Expect("Pay",avis,805);
 avis.ReturnCar((ICar)mockedCar.MockInstance);
 mockPerson.Verify();
 mockCar.Verify();
}
```

The test above creates a mocked car, tells it the return its current price and when it was rented. The renter is mocked as well, and is expecting to pay for renting the car. (SetupResult() is a way to return a value without creating an expectation for it, more on that later). So, now we can see some of the problems with NMock, look at the

[1] A quote by Izhaq Rabin, former Prime Minister of Israel

ReturnToarkingLot method, the IsTypeOf() and IsAnything() are constraints (a very powerful part of NMock). I'm not using them to *do* anything, I have to put them there so NMock will recognize that the method exists (it otherwise try to find a method with no parameters and fail at runtime).
Another problem that NMock has is that it's not malleable to refactoring. This means that if I decide to rename the RentedBy property to Renter, I'm very likely to miss the tests, and I'll get no warning until runtime.
A recent experience with exactly this case caused me to write Rhino Mocks, but I'm jumping too far ahead. I just became aware of another mocking framework for .Net (there are several, most of which works in the same method, and share the same weaknesses).
NMock2 is still not officially released, but it's available from CVS at http://nmock.sf.net

Here is a usage sample of NMock2 in the above test:

```
 [Test]
public void RenterPaysForCar ()
{
  Mockery mocker = new Mockery();
  ICar car = mocker.NewMock(typeof(ICar)) as ICar;
  IPerson person = mocker.NewMock(typeof(IPerson)) as IPerson;
  Stub.On(car).GetProperty("CurrentPrice").Will(Return.Value(115));
  Stub.On(car).GetProperty("RentedFrom").Will(Return.Value(DateTime.Now.AddDays(-7));
  Stub.On(car).GetProperty("RentedBy").Will(Return.Value(person));
  Stub.On(car).Method("ReturnToParkingLot");
  ICarRental avis= new AvidCarRental();
  Expect.On(person).Method("Pay").With(avis,805);
  avid.ReturnCar((ICar)mockedCar.MockInstance);
  mocker.VerifyAllExpectationsHaveBeenMet();
 }
```

This is a different approach, which read much like written language.
Notice that I didn't have to specify the empty constraints to get it to work; it's now smart enough to recognize that I just don't care about the arguments and just verify that I got the call. (There are some subtleties regarding overloaded calls, but they are minor.)

One weakness that remains is that it's still breaks on refactoring. It's also remains difficult to inject your own logic to the mocked object[2].

"They mock the air with idle state."
~Thomas Gray quotes (English Poet, 1716-1771)

[2] I've not tested this, though.

Benefits of Mocking:

- Recognized testing pattern
- Encourage passing objects as method parameters rather than making references to them part of our object state.
- Mock objects tend to drive your classes into a highly compositional design with lots of little pieces and lots of interface surface
- Encourage lightweight classes and more flexible methods.

EasyMock and EasyMock.Net:

EasyMock is one of the first mock objects frameworks to appear on Java, and EasyMock.Net is the direct port to .Net. These frameworks use the explicit Record & Replay model[3] to setup and verify tests.
They make use of strongly typed object during both record & replay phases, thus supporting refactoring tools easily and safely. However, they are quite rigid in the way they accept expectations.

Here is how the test above using EasyMock.Net:

```
[Test]
public void RenterPaysForCar ()
{
  MockControl controlCar = MockControl.CreateControl(typeof(ICar));
  MockControl controlPerson = MockControl.CreateControl(typeof(IPerson));
  ICar car = controlCar.GetMock() as ICar;
  IPerson person = controlPerson.GetMock() as IPerson;
  controlCar.ExpectAndReturn(car.CurrentPrice, 115);
  controlCar.ExpectAndReturn(car.RentedFrom,DateTime.Now.AddDays(-7));
  controlCar.ExpectAndReturn(car.RentedBy,person);
  car.ReturnToParkingLot(null);
  controlCar.SetMatcher(MockControl.ALWAYS_MATCH);
  ICarRental avis= new AvisCarRental();
  person.Pay(avis,850);
 controlPerson.Replay();
  controlCar.Replay();
  avis.ReturnCar((car);
  controlCar.Verify();
  controlPerson.Verify();
 }
```

It's hard to see from an example this small, but the port to .Net didn't include the .Net style guide, which result in a very alien feeling when working with this library. In addition, there is no way to express flexible constraints on the methods or to inject your own logic.

"Be mock'd and wonder'd at."
~Henry VI Part 3, 5. 4, William Shakespeare

[3] The other frameworks uses the same model, only implacably

The case for Rhino Mocks:

So there are all these frameworks, they have their drawbacks, but then so are most things in life, why do we need yet another framework?
The reasoning is simple, I don't subscribe to NIH; at least not usually.
I had several problems with my mock objects in my tests:

- They broke several times as I refactored the application. I would've have minded that if they didn't do that at runtime. [And with a slew of errors that made me think that I would need to revert all that week's work.]
- I'd to resort to hacks to get callbacks from the mocked methods. This is not a good way to do mocks, but I'd no choice, I had to test some threading calls, and that meant callback from the mocked methods.[4]
- I was annoyed by the needless constraints passing that I'd to do in order to get NMock to recognize the correct method signature.
- I wanted a mocking framework that would adapt to my refactoring, would give me easy way to inject my own validation and would be smart enough to blip when I'm wrong.

There was no such framework, but EasyMock.Net was close. It had the essential capability of using strongly typed object to record the expectations. What it lacked was flexible validation conditions and the ability to inject custom code into the validation. Another big problem was that it was still a Java library, but on .Net, the whole API uses concepts like SetVoidMethod(), SetReturnValue().

I took EasyMock.Net and started modifying it. The first thing was to give it a .Net interface, which went pretty smoothly, the other two objectives were completed within three days (there were some hurdles, but nothing I couldn't handle with).

Rhino Mocks objects allows **Expectations**, **Constraints** and **Callbacks** to be setup, specifying how the tested object *should* interact with the mocked object. These conditions are evaluated throughout the test run, and fails early in order to give you a warning as localizable as possible.

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.

[4] Yes, I could've built my own test class for that, but for a single line of code it's a bit drastic, in my opinion.

- 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:

Here is how Rhino Mocks would handle the previous tests:

```
[Test]
public void RenterPaysForCar ()
{
  MockControl mockCar= MockControl.CreateControl(typeof(ICar));
  MockControl mockPerson = MockControl.CreateControl(typeof(IPerson));
  ICar car = controlCar.GetMock() as ICar;
  IPerson person = controlPerson.GetMock() as IPerson;
  mockCar.ExpectAndReturn(car.CurrentPrice, 115);
  mockCar.ExpectAndReturn(car.RentedFrom,DateTime.Now.AddDays(-7));
  mockCar.ExpectAndReturn(car.RentedBy,person);
  ICarRental avis= new AvisCarRental();
  mockCar.SetupResult(car.ReturnToParkingLot(avis))
  mockCar.Callback = new
          CarRentalDelegate(InformReachedParkingLot);
  person.Pay(avis,850);
  avis.ReturnCar(car);
  controlCar.Verify();
  controlPerson.Verify();
 }

delegate void CarRentalDelegatE(ICarRental carRental);
void InformReachedParkingLot(ICarRental carRental)
{//assumes that car is an instance variable, not a local variable in the te
st
  carRental.CarInParkingLot(car);
}
```

In this case, it's not much of a difference, that is because we are using expectations), the same way EasyMock.Net works. Let's try something more difficult, let's assume that when the car rental sends the car to its parking lot, they require the car to inform that it arrived safely before letting the renter go. In essence, classic threading problem, but one that is *very* hard to solve using NMock. I created a couple of hacks in order to solve. Usually the solution for those kinds of problem is to write your own test class. This isn't very practical[5] for large number of classes, here is how I would solve this using Rhino Mocks, I'll leave you implementing the same with the other framework.

You can see that we set up a callback for ReturnToParkingLot, and using that callback we get the ICarRental reference that we got. So we can do call the CarInParkingLot() andreturn from ReturnCar() method.

Another way to specify conditions is to use constraints, they are used in much the same way as they are in NMock:

```
car.AddGas(0);
mockCar.Constraints(new IsEqualOrLessThan(150));
```

The Last Method concept:

Most actions in Rhino.Mocks are done on the last method call, 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 has overloads that allows you to setup returning or throwing for multiply calls of the same method. 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()).

[5] I guess that this is a contested opinion

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;
```

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.

Mocking Classes:

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

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 the 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.

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