Rhino Mocks Documentation

What is Rhino Mocks?

Rhino Mocks allows you to easily create mock objects and setup a wide range of expectations on them using strongly typed notation instead of compiler-opaque strings. It's as simple as:

```
IProjectView projectView = mocks.CreateMock<IProjectView>();
Expect.Call(projectView.Title).Return("Project's Title");
```

Definitions:

- **Mock Object - an object that pretends to be another object and allows you to set expectations on its interactions with another object.**
- **Interaction Based Testing - you specify a certain sequence of interactions between objects, initiate an action, and then verify that the sequence of interactions happened as you specified it.**
- **State Based Testing - you initiate an action and then check for the expected results (return value, property, created object, etc).**
- **Expectation - general name for validation that a particular method call is the expected one.**
- **Record & Replay model - a model that allows for recording actions on a mock object and then replaying and verifying them. All mocking frameworks uses this model. Some (NMock, TypeMock.Net, NMock2) use it implicitly and some (EasyMock.Net, Rhino Mocks) use it explicitly.**
- **Ordering - The ability to specify that replaying a sequence of method calls will occur in a specific order (or disorder).**

Change Log

Check here for Rhino Mocks Change Log

Capabilities

- **mock interfaces, delegates and classes, including those with parameterized constructors.**
- **set expectations on the called methods by using strongly typed mocks instead of strings.**
- **lends itself easily to refactoring & leaning on the compiler.**
- **allows a wide range of expectations to be set on a mock object or several mock objects.**
- **Attention: Rhino Mocks can only mock interfaces, delegates and virtual methods of classes! (Although it is possible to get work around existing code and easily apply Rhino Mocks, see Example of testing an abstract class)**

Documentation

- **Rhino Mocks Introduction**
- **Rhino Mocks Generics**
- **Rhino Mocks Partial Mocks**
- **Rhino Mocks - Stubs**
- **Rhino Mocks Ordered and Unordered**
- **Rhino Mocks Mocking Delegates**
- **Rhino Mocks Events**
- **Rhino Mocks IEventRaiser**
- **Rhino Mocks Properties**
- **Rhino Mocks Callbacks**
- **Rhino Mocks The Do() Handler**
- **Rhino Mocks Constraints**
- **Rhino Mocks Method Options Interface**
- **Rhino Mocks Setup Result**
- **Rhino Mocks Mocking classes**
- **Rhino Mocks Internal Members**
- **Rhino Mocks Record-playback Syntax**
- **Rhino Mocks With-Syntax**
- **Rhino Mocks Limitations**

- **Rhino Mocks 3.1 Quick Reference PDF**
- **Rhino Mocks 3.3 Quick Reference PDF - comprehensive examples**

Rhino Mocks Introduction

Planning the tests

[I tried to create an example for this article, but the example was both contrived and didn't portray the possibilities correctly. So most of the code samples here are taken from NHibernate Query Analyzer tests and are "real code".]

The purpose of mock objects is to allow you to test the interactions between components, this is very useful when you test code that doesn't lend itself easily to state based testing. Most of the examples here are tests that check the save routine for a view to see if it is working as expected. The requirements for this method are:

- **If the project is a new project, then ask for a name for the project. (Allow canceling save at this point)**
  - **If the new project name already exists, ask if user wants to overwrite it. (If not, cancel the save operation)**
  - **If the user approves the overwrite, delete the old project.**
- **Save the project.**

These requirements give us five scenarios to test:

1. **The project is a new project and the user cancels on new name.**
2. **The project is a new project, the user gives a new name and the project is saved.**
3. **The project is a new project, the user gives a name that already exists and does not approve an overwrite.**
4. **The project is a new project, the user gives a name that already exists and approves an overwrite.**
5. **The project already exists, so just save it.**

Trying to test the above using state based testing will be awkward and painful, using interaction based testing, it would be a breeze (other types of scenarios might be just as painful to test using interaction based testing but easier to test with state based testing).

The first test

```
[Test]
public void SaveProjectAs_CanBeCanceled()
{
  MockRepository mocks = new MockRepository();
  IProjectView projectView = mocks.CreateMock<IProjectView>();
  Project prj = new Project("Example Project");
  IProjectPresenter presenter = new ProjectPresenter(prj,projectView);
  Expect.Call(projectView.Title).Return(prj.Name);
  Expect.Call(projectView.Ask(question,answer)).Return(null);
  mocks.ReplayAll();
  Assert.IsFalse(presenter.SaveProjectAs());
  mocks.VerifyAll();
}
```

We create a MockRepository and create a mock project view, project and a project presenter, then we set the expectations. After we finished with setting up the expectations, we move to the replay state, and call the SaveProjectAs() method, which should return false if the user canceled the save process.

This example clarifies several key concepts related to Rhino Mocks.

- **We set expectation on object using the object's methods, and not strings, this reduce the chances of making a mistake that will only (hopefully) be caught at runtime, when you run the tests.**
- **We are using explicit call to move from Record state to Replay state.**
- **We verify that all the expectations has been met.**

This is about as simple example as can be had, the real test moves creating the MockRepository, the project, the view and presenter to the setup method, since they are require for each test, and the expectations verification to the teardown method, since it's easy to forget that. You can see that we expected two methods to be called, with specific arguments, and that we set a result for each. This method uses parameter matching expectations, Rhino Mocks supports several more. More info: Rhino.Mocks::Method Calls.

Mocks, Dynamic Mocks and Partial Mocks oh my!:

Rhino Mocks currently support the creation of the following types of mock objects.

- **Mock Objects - Strict replay semantics. - Created by calling CreateMock()**
- **Dynamic Mock - Loose replay semantics. - Created by calling DynamicMock()**
- **Partial Mock - Mock only requested methods. - Created by calling PartialMock()**

What is the meaning of these?

Strict replay semantics: only the methods that were explicitly recorded are accepted as valid. This mean that any call that is not expected would cause an exception and fail the test. All the expected methods must be called if the object is to pass verification.

Loose replay semantics: any method call during the replay state is accepted and if there is no special handling setup for this method a null or zero is returned. All the expected methods must be called if the object is to pass verification.

Mocking only requested methods: this is available for classes only. It basically means that any non abstract method call will use the original method (no mocking) instead of relying on Rhino Mocks' expectations. You can selectively decide which methods should be mocked.

Let's see some code that would explain it better than words. The difference between the tests are marked with bold. First, the mock object code:

```
[Test]
[ExpectedException(typeof (ExpectationViolationException),
    "IDemo.VoidNoArgs(); Expected #0, Actual #1.")]
public void MockObjectThrowsForUnexpectedCall()
{
  MockRepository mocks = new MockRepository();
  IDemo demo = mocks.CreateMock<IDemo>();
  mocks.ReplayAll();
  demo.VoidNoArgs();
  mocks.VerifyAll();//will never get here
}
```

As you can see, calling a method that wasn't explicitly setup will cause an exception to be thrown, now let's try it with a dynamic mock:

```
[Test]
public void DyamicMockAcceptUnexpectedCall()
{
  MockRepository mocks = new MockRepository();
  IDemo demo = mocks.DynamicMock<IDemo>();
  mocks.ReplayAll();
  demo.VoidNoArgs();
  mocks.VerifyAll();//works like a charm
}
```

An unexpected call is ignored when you are using a dynamic mock. However, this is the only difference between the two types, in all other ways they are identical (ordering, recording a method expectation, callbacks, etc). Any expectations that were created during the record phase must be satisfied if the dynamic mock is to pass verification.

Dynamic mocks are useful when you want to check a small piece of functionality and you don't care about whatever else may happen to the object. Mock objects are useful when you want complete control over what happens to the mocked object.

For an explanation of partial mocks, see Rhino Mocks Partial Mocks

---

Up: Rhino Mocks Documentation
Next: Rhino Mocks Generics

Rhino Mocks Generics

While Rhino Mocks is compatible with the .Net framework 1.1, it offer several goodies for those who use the 2.0 version of the .Net framework. Among them are mocking generic interfaces and classes, and using generic version of the methods in order to reduce casting.

Here is Rhino Mocks mocking a generic interface:

```
[Test]
public void MockAGenericInterface()
{
   MockRepository mocks = new MockRepository();
   IList<int> list = mocks.CreateMock<IList<int>>();
   Assert.IsNotNull(list);
   Expect.Call(list.Count).Return(5);
   mocks.ReplayAll();
   Assert.AreEqual(5, list.Count);
   mocks.VerifyAll();
}
```

And here is how to use the generic methods to create mocks:

- **For .Net 1.1:**

```
IDemo demo = (IDemo) mocks.CreateMock(typeof(IDemo));
IDemo demo = (IDemo) mocks.DynamicMock(typeof(IDemo));
```

- **And for .Net 2.0:**

```
IDemo demo = mocks.CreateMock<IDemo>();
IDemo demo = mocks.DynamicMock<IDemo>()
```

Up: Rhino Mocks Documentation
Next: Rhino Mocks Partial Mocks

Rhino Mocks Partial Mocks

Partial mocks are useful when you want to mock a part of a class. This allows you to test abstract methods in isolation, for instance.

Let's imagine this (useless) class. We want to test that the Inc() method is doing its job, but we don't want to implement a derived class yet.

```
public abstract class ProcessorBase
{
  public int Register;
  public virtual int Inc()
  {
    Register = Add(1);
    return Register;
  }
  public abstract int Add(int i);
}
```

Here is what we need to do:

```
[Test]
public void UsingPartialMocks()
{
  MockRepository mocks = new MockRepository();
  ProcessorBase proc = (ProcessorBase) mocks.PartialMock(typeof (ProcessorBase));
  Expect.Call(proc.Add(1)).Return(1);
  Expect.Call(proc.Add(1)).Return(2);
  mocks.ReplayAll();
  proc.Inc();
  Assert.AreEqual(1, proc.Register);
  proc.Inc();
  Assert.AreEqual(2, proc.Register);
  mocks.VerifyAll();
}
```

And with generics:

```
[Test]
public void UsingPartialMocks()
{
  MockRepository mocks = new MockRepository();
  ProcessorBase proc =  mocks.PartialMock<ProcessorBase>();
  Expect.Call(proc.Add(1)).Return(1);
  Expect.Call(proc.Add(1)).Return(2);
  mocks.ReplayAll();
  proc.Inc();
  Assert.AreEqual(1, proc.Register);
  proc.Inc();
  Assert.AreEqual(2, proc.Register);
  mocks.VerifyAll();
}
```

This is a contrived example, but you can see how we can use this feature to test Template Methods and abstract classes easily. A partial mock will call the method defined on the class unless you define an expectation for that method. If you have defined an expectation, it will use the normal rules for this. (Be aware, though, that once all the expectations for a method are settled, the method will be routed back to the implementation.)

Up: Rhino Mocks Documentation
Next: Rhino Mocks - Stubs

Rhino Mocks - Stubs

According to Fowler:

Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.

Relevant Blog post.

You can use mocks as stubs, you can create a dynamic mock and call PropertyBehavior on its methods. The problem is that this is really annoying. Here is the type of code that you have to write for even mildy complex stub:

```
public interface IAnimal
{
       int Legs { get; set; }
       int Eyes { get; set; }
       string Name { get; set; }
       string Species { get; set; }

       event EventHandler Hungry;
       string GetMood();
}

[Test]
public void CreateAnimalStub()
{
   MockRepository mocks = new MockRepository();
   IAnimal animal = mocks.DynamicMock<IAnimal>();
   Expect.Call(animal.Legs).PropertyBehavior();
   Expect.Call(animal.Eyes).PropertyBehavior();
   Expect.Call(animal.Name).PropertyBehavior();
   Expect.Call(animal.Species).PropertyBehavior();
}
```

That is far too long and tiring. Fortunately, Rhino Mocks has a builtin support for stubs:

```
public interface IAnimal
{
       int Legs { get; set; }
       int Eyes { get; set; }
       string Name { get; set; }
       string Species { get; set; }

       event EventHandler Hungry;
       string GetMood();
}
```

The animal instance that we got is a stub, is will respond correctly to properties and events, and it can be setup so it will return expected results from methods. Note that stubs in Rhino Mocks defaults to ignoring everything, including things that would be invalid in other mock types (such as calling a method that return a value without supplying such value). Furthermore, for the simple scenario, where all you would like to have is a stub, Rhino Mocks provide a static accessor method to make your life easier:

Generate the animal stub using the GenerateStub method.

```
[Test]
// Tests stub creation with the GenerateStub method
public void CreateAnimalStub_GenerateStub()
{

   IAnimal animal = MockRepository.GenerateStub<IAnimal>();

   animal.Name = "Snoopy";
```

```
   Assert.AreEqual( "Snoopy", animal.Name );
}
```

Generate the animal stub using the MockRepository.Stub method. Use this if you are using .Net 2.0 and do not want to use the GenerateStub method

```
[Test]
public void CreateAnimalStub_MockRepositoryStub()
{

   MockRepository mocks = new MockRepository();
   IAnimal animal = mocks.Stub<IAnimal>();

   animal.Name = "Snoopy";

   Assert.AreEqual( "Snoopy", animal.Name );
}
```

Use this alternative syntax if you are using .Net 1.1

```
[Test]
public void CreateAnimalStub_NonGenerics()
{

   MockRepository mocks = new MockRepository();
   IAnimal animal = (IAnimal) mocks.Stub( typeof ( IAnimal ), null );

   animal.Name = "Snoopy";

   Assert.AreEqual( "Snoopy", animal.Name );
}
```

For properties, you do not need to move a stub to replay mode, but if you would like to setup return values for methods, you need to let Rhino Mocks know when you are finished setting up the stub, and then you can move it to replay mode, and let the rest of the test run.

Up: Rhino Mocks Documentation
Next: Rhino Mocks Ordered and Unordered

Rhino Mocks Ordered and Unordered

Method calls in Rhino Mocks can be ordered or unordered. The default state for a recorder is unordered recording, this means that during replay, methods can come at any order. If the recorder is changed to ordered, then the methods must be called in the exact same order as they were recorded. Here is a code sample:

```
Test
public void SaveProjectAs_NewNameWithoutConflicts()
{
  using(mocks.Ordered())
  {
    Expect.Call(projectView.Title).
      Return(prj.Name);
    Expect.Call(projectView.Ask(question,answer)).
      Return( newProjectName);
    Expect.Call(repository.GetProjectByName(newProjectName)).
      Return(null);
    projectView.Title = newProjectName;
    projectView.HasChanges = false;
    repository.SaveProject(prj);
  }
  mocks.ReplayAll();
  Assert.IsTrue(presenter.SaveProjectAs());
  Assert.AreEqual(newProjectName,prj.Name);
}
```

In the above code example we ask Rhino Mocks to verify that the calls come in the exact same order. Notice that we specify expectations on several mock objects, and Rhino Mocks will handle the ordering between them. This mean that if I set the project view title before I get a project from the repository, the test will fail. In Rhino Mocks, the default is to use unordered matching, but it support unlimited depth of nesting between ordered and unordered, this means that you can create really powerful expectations. Here is a somewhat contrived example:

```
Test
public void MovingFundsUsingTransactions()
{
  MockRepository mocks = new MockRepository();
  IDatabaseManager databaseManager = mocks.CreateMock<IDatabaseManager>();
  IBankAccount accountOne = mocks.CreateMock<IBackAccount>(),
         accountTwo = mocks.CreateMock<IBackAccount>();
  using(mocks.Ordered())
  {
        Expect.Call(databaseManager.BeginTransaction()).
           Return(databaseManager);
         using(mocks.Unordered())
         {
            accountOne.Withdraw(1000);
            accountTwo.Deposit(1000);
         }
         databaseManager.Dispose();
  }
  mocks.ReplayAll();
  Bank bank = new Bank(databaseManager);
  bank.TransferFunds(accountOne,accountTwo,1000);
  mocks.VerifyAll();
}
```

This code verify that the transfer of funds from one account to another is wrapped in a transaction, but the implementation is free to withdraw from the first account first, or to deposit into the second account first, both are legal, as long as both actions happens. The reverse is true as well, you may specified unordered sequence of ordered events (I want A then B then C to happen, and I want D then E then F to happen, but I don't care which sequence comes first).

Ordering has two caveats:

- **To exit an ordering in replay state, you must call all the recorded methods. In the above example, we can move from the inner Unordered ordering (the one containing the withdraw and deposit) only after both methods were called. This falls in line with the way the recording code looks, so it shouldn't cause any surprises.**
- **You must exit all ordering before you can start replaying. This is enforced by the framework (you get an exception if you try).**

Up: Rhino Mocks Documentation
Next: Rhino Mocks Mocking Delegates

Rhino Mocks Mocking Delegates

Sometimes you want to be able to verify/affect the invocation of a delegate (this is especially useful if you're used to functional programming) and pass delegates around to do your bidding. Personally, I find it a very useful technique. Rhino Mocks allows you to handle that cleanly by mocking the delegate call, and put it under all the usual capabilities that you have in Rhino Mocks. Here is an example:

```
[Test]
public void GenericDelegate()
{
  Action< int> action = mocks.CreateMock< Action< int>>();
  for (int i = 0; i < 10; i++)
  {
    action(i);
  }
  mocks.ReplayAll();
  ForEachFromZeroToNine(action);
  mocks.VerifyAll();
}
private void ForEachFromZeroToNine(Action< int> act)
{
  for (int i = 0; i < 10; i++)
  {
    act(i);
  }
}
```

The example here is using a generic delegate, but of course all delegates are supported.

Up: Rhino Mocks Documentation
Next: Rhino Mocks Events

Rhino Mocks Events

Test Subscribe To Event

Here is how you check that the object under test subscribes to an event on a mock object:

```
public interface IWithEvents
{
     event EventHandler Blah;
     void RaiseEvent();
}

[Test]
public void VerifyingThatEventWasAttached()
{
     MockRepository mocks = new MockRepository();
     IWithEvents events = mocks.CreateMock<IWithEvents>();
     With.Mocks(mocks).Expecting(delegate
     {
          events.Blah += new EventHandler(events_Blah);
     })
    .Verify(delegate
    {
          MethodThatSubscribeToEventBlah(events);
    });
}

public void MethodThatSubscribeToEventBlah(IWithEvents events)
{
     events.Blah += new EventHandler(events_Blah);
}
```

You just record that you want to subscribe to the event and then run the code that is supposed to subscribe to it.

Test Event Was Raised

Now, let's try do it from the other side, how do check that an event was raised? Well, that is a little more complicated, but not very much so. What we need to do create an interface with a matching method name, create a mock from this interface, and then just subscribe to the event and record any expectations regarding the method. As usual, the code is clearer then the explanation:

```
public interface IEventSubscriber
{
     void Handler(object sender, EventArgs e);
}
public class WithEvents : IWithEvents
{
     #region IWithEvents Members
     public event System.EventHandler Blah;

     public void RaiseEvent()
     {
           if (Blah!=null)
                Blah(this, EventArgs.Empty);
     }
     #endregion
}

[Test]
public void VerifyingThatAnEventWasFired()
{
     MockRepository mocks = new MockRepository();
     IEventSubscriber subscriber = mocks.CreateMock<IEventSubscriber>();
     IWithEvents events = new WithEvents();
     // This doesn't create an expectation because no method is called on subscriber!!
     events.Blah+=new EventHandler(subscriber.Handler);
     subscriber.Handler(events, EventArgs.Empty);
     mocks.ReplayAll();
     events.RaiseEvent();
     mocks.VerifyAll();
}
```

One thing to notice here is that subscribing to the event does not create an expectation, since no action is taken on the subscriber instance. Then, to create an expectation that the method will be raised with the specified arguments, we just call the method we just subscribed. Since the raised event will eventually

resolve itself to the same method call, everything works as expected.

Rhino Mocks also support raising events, using the IEventRaiser interface.

Up: Rhino Mocks Documentation
Next: Rhino Mocks IEventRaiser

Rhino Mocks IEventRaiser

The event raiser is a solution to a common problem, how do you raise an event from an interface? Let us consider this code:

```
public interface IView
{
   event EventHandler Load;
}

public class Presenter
{
   public bool OnLoadCalled = false;

   public Presenter(IView view)
   {
      view.Load+= OnLoad;
   }

   private void OnLoad(object sender, EventArgs e)
   {
     OnLoadCalled  = true;
     //the code that we would like to test
   }
}
```

How are we going to test this code? We can utilize the usual event handling mechanisms to verify that the presenter was registered, but notice that the OnLoad method is private, so we can't access it normally. We would like to raise the event, so the method would be called.

IEventRaiser is the answer to that. As usual, we would like to deal with strongly typed objects, so we need to have some way to mark which event we would like to use.

Getting IEventRaiser Implementation:

```
Test
public void RaisingEventOnView()
{
   IView view = mocks.CreateMock<IView>();
   view.Load+=null;//create an expectation that someone will subscribe to this event
   LastCall.IgnoreArguments();// we don't care who is subscribing
   IEventRaiser raiseViewEvent = LastCall.GetEventRaiser();//get event raiser for the last event, in this case, View
}
```

This coincides with testing that someone is subscribing to the event, so the somewhat awkward syntax doesn't matter that much. Now, how do I raise the event?

```
Test
public void RaisingEventOnView()
{
   IView view = mocks.CreateMock<IView>();
   view.Load+=null;//create an expectation that someone will subscribe to this event
   LastCall.IgnoreArguments();// we don't care who is subscribing
   IEventRaiser raiseViewEvent = LastCall.GetEventRaiser();//get event raiser for the last event, in this case, View

   mocks.ReplayAll();

   Presenter p = new Presenter(view);
   raiseViewEvent.Raise();

   Assert.IsTrue(p.OnLoadCalled );
}
```

So, that was simple :-)

Note: Event Subscribers, like all other information on a mock object, will be cleared if BackToRecord(mock) or BackToRecordAll() are called.

Up: Rhino Mocks Documentation
Next: Rhino Mocks Properties

Rhino Mocks Properties

If you want to mock properties, you need to keep in mind that properties are merely special syntax for normal methods, a get property will translate directly to propertyType get_PropertyName() and a set property will translate directory to void set_PropertyName(propertyType value). So how do you create expectations for a property? Exactly as you would for those methods.

Here is how you set the return value for a get property:

```
IList list = mocks.CreateType<IList>();
SetupResult.For(list.Count).Return(42);
```

And here is how you would set an expectation for a set property:

```
IList list = mocks.CreateType<IList>();
list.Capacity = 500;//Will create an expectation for this call
LastCall.IgnoreArguments();//Ignore the amounts that is passed.
```

One interesting feature you can take advantage of is automatic handling of properties. You use it like this:

```
[Test]
public void PropertyBehaviorForSingleProperty()
{
  Expect.Call(demo.Prop).PropertyBehavior();
  mocks.ReplayAll();
  for (int i = 0; i < 49; i++)
  {
    demo.Prop = "ayende" + i;
    Assert.AreEqual("ayende" + i, demo.Prop);
  }
  mocks.VerifyAll();
}
```

As you can see, you specify PropertyBehavior() on the property, and that is all. Rhino Mocks will emulate a simple property behavior for the property. You can see that this is not a transient behavior, but one that remains for the lifetime of the object.

Up: Rhino Mocks Documentation
Next: Rhino Mocks Callbacks

Rhino Mocks Callbacks

A callback is a user supplied delegate that is called whenever Rhino Mocks needs to evaluate whether a method call is expected or not. This is useful in some scenarios when you want to do a complex validation on a method arguments, or have a complex interactions between objects that you need to mock in the tests. I added this feature because I want to test some threading code which had the semantics of: Start the Job, and notify me when it's done. The only way to re-create that without bringing the real thread (and killing test isolations) was to plug my own code in during the replay state and call the tested object myself.

Some things to consider before you decide to use callbacks:

- **Your callback delegate must have a bool return type and it must return true if the method passed your validation. If it didn't pass you validation, just return false and Rhino Mocks will take it from there. The return value signals to the framework that this is the method call that this callback should match.**
- **It can be abused very easily. It can be very hard to understand tests that use callbacks, because you get calls from supposedly innocent code. Do it only if you need it.**
- **You callback may (and likely will be) called several times; keep that in mind when you write it. Either wrap it all in a if (firstTimeCalled) { /*do work*/ } or make sure that repeated calls for the delegate won't have some nasty side effects.**

If it is so open to abuse, why add it?

- **Because when you need it, you really need it, and I would prefer that I'd some nice way to do it, and not some really ugly hacks.**
- **Because I respect that those who will use this framework not to take the power and abuse it.**

The technical details - In order to be a valid callback, the callback must return a Boolean, and have the same arguments as the mocked methods. You register a delegate using the following code:

```
IProjectRepository repository = mocks.CreateMock<IProjectRepository>();
IProjectView view = mocks.CreateMock<IProjectView>();
Expect.Call(view.Ask(null,null)).Callback(new AskDelegate(DemoAskDelegateMethod)).Return(null);
```

Notice that you cannot change the return value for the method, but must pass it explicitly.

Recursive Expectations

One final word of warning regarding callbacks. If your callback will initiate an action that cause another call on a mocked object, this will fail when mixed with Ordered(). The reason for that is that the framework cannot decide whatever to accept the call or not, and calling another mocked method while evaluating means that the currently expected call is the one still being evaluated. This will fail the test. Using Unordered(), it will work, since the second expectation is not dependant on the first one being accepted first. In short, Ordered() is not re-entrant :-)

Usage with Other MethodOptions

Note that RhinoMocks will throw a InvalidOperationException at runtime if you attempt to specify both a callback AND a constraint. Rather than have possibly conflicting criteria in two places, check for parameter constraints in the callback if you're using one.

Up: Rhino Mocks Documentation
Next: Rhino Mocks The Do() Handler

Rhino Mocks The Do() Handler

There are times when the returning a static value is not good enough for the scenario that you are testing, so for those cases, you can use the Do() handler to add custom behavior when the method is called. In general, the Do() handler simply replaces the method call. Its return value will be returned from the mocked call (as well as any exception thrown). The handler's signature must match the method signature, since it gets the same parameters as the call.

Let's see an example, a speaker can introduce itself formally or informally, so we'll separate it into a separate class, but we want to test the speaker in isolation, so we get this code (contrived again, I'm afraid):

```
[Test]
public void SayHelloWorld()
{
  INameSource nameSource = mocks.CreateMock<INameSource>();
  Expect.Call(nameSource.CreateName(null,null)).IgnoreArguments().
      Do(new NameSourceDelegate(Formal));
  mocks.ReplayAll();
  string expected = "Hi, my name is Ayende Rahien";
  string actual = new Speaker("Ayende", "Rahien", nameSource).Introduce();
  Assert.AreEqual(expected, actual);
}
 delegate string NameSourceDelegate(string first, string suranme);
 private string Formal(string first, string surname)
{
  return first + " " +surname;
}

public class Speaker
{
  private readonly string firstName;
  private readonly string surname;
  private INameSource nameSource ;
 public Speaker(string firstName, string surname, INameSource nameSource)
  {
    this.firstName = firstName;
    this.surname = surname;
    this.nameSource = nameSource;
  }
  public string Introduce()
  {
    string name = nameSource.CreateName(firstName, surname);
    return string.Format("Hi, my name is {0}", name);
  }
}
public interface INameSource
{
  string CreateName(string firstName, string surname);
}
```

Note: The Do() handler is only called if the method was matched, and it is called once per method match. This is different from Callback(), which may be called many times.

Maintainability note: If you've a complex logic going on for the test, you should consider making a class manually, it will probably be easier in the long run.

Up: Rhino Mocks Documentation
Next: Rhino Mocks Constraints

Rhino Mocks Constraints

Constraints are a way to verify that a method's arguments match a certain criteria. Rhino Mocks includes a number of built-in constraints, and allows you to define you own custom ones, which will integrate cleanly into the framework. You specify constraints on method arguments using the following syntax:

```
Expect.Call(view.Ask(null,null)).IgnoreArguments().Constraints(
   Is.Anything(), Is.TypeOf(typeof(SomeType))).Return(null).
```

You need to pass the exact same number of constraints as the number of arguments of the method. When a method is called during replay state, Rhino Mocks evaluates each constraint against the parameter in the same index, and accepts the method if all the constraints were met. As a note, I got the idea of the current constraints syntax from NMock2, as it is much leaner approach to create constraints.

Rhino Mocks' built in constraints:

Rhino Mocks Constraints

Constraint		Example	Accepted Values	Rejected Values
Is	Anything	Is.Anything()	Anything at all {0,"","whatever",null, etc}	Nothing What so ever
	Equal	Is.Equal(3)	3	5
	Not Equal	Is.NotEqual(3)	3,null, "bar"	3
	Not Equal	Is.NotEqual(3)	3,null, "bar"	3
	Null	Is.Null()	null	5, new object()
	Not Null	Is.NotNull()	new object(), DateTime.Now	null
	Type Of	Is.TypeOf(typeof(Customer)) or Is.TypeOf<Customer>()	myCustomer, new Customer()	null, "str"
	Greater Than	Is.GreaterThan(10)	15,53	2,10
	Greater Than Or Equal	Is.GreaterThanOrEqual(10)	10,15,43	9,3
	Less Than	Is.LessThan(10)	1,2,3,9	10,34
	Less Than Or Equal	Is.LessThanOrEqual(10)	10,9,2,0	34,53,99
Property	Equal To Value	Property.Value("Length",0)	new ArrayList()	"Hello", null
	Null	Property.IsNull("InnerException")	new Exception("exception without inner exception")	new Exception("Exception with inner Exception", new Exception("Inner")
	Not Null	Property.IsNotNull("InnerException")	new Exception("Exception with inner Exception", new Exception("Inner")	new Exception("exception without inner exception")
List	Is In List [the parameter is a collection that contains this value]	List.IsIn(4)	new int[]{1,2,3,4}, new int[]{4,5,6}	new object[]{"",3}

	One Of [parameter equal to one of the objects in this list]	List.OneOf(new int[]{3,4,5})	3,4,5	9,1,""
	Equal	List.Equal(new int[]{4,5,6})	new int[]{4,5,6}, new object[]{4,5,6}	new int[]{4,5,6,7}
Text	Starts With	Text.StartsWith("Hello")	"Hello, World", "Hello, Rhino Mocks"	"", "Bye, Bye"
	Ends With	Text.EndsWith("World")	"World","Champion Of The World"	"world", "World Seria"
	Contains	Text.Contains("or")	"The Horror Movie...", "Either that or this"	"Movie Of The Year"
	Like [Perform regular expression validation]	rhino|Rhinoceros|rhinoceros")	"Rhino Mocks", "Red Rhinoceros"	"Hello world", "Foo bar", Another boring example string"

Operator overloading

Rhino Mocks Constraints can be used with the following operators:

- **!|- Not operator**
- **&& - And operator**
- **|| - Or Operator**

Usage with Other MethodOptions

Note that RhinoMocks will throw a InvalidOperationException at runtime if you attempt to specify both a constraint AND a callback. Rather than have possibly conflicting criteria in two places, check for parameter constraints in the callback if you're using one.

Up: Rhino Mocks Documentation
Next: Rhino Mocks Method Options Interface

Rhino Mocks Method Options Interface

The IMethodOptions allows you to set various options on a method call. Here is an example of telling Rhino Mocks to ignore the arguments of a method:

```
IProjectRepository repository = mocks.CreateMock<IProjectRepository>();
IProjectView view = mocks.CreateMock<IProjectView>();
Expect.Call(view.Ask(null,null)).IgnoreArguments().Return(null);
repository.SaveProject(null);
LastCall.IgnoreArguments();
```

As you can see, we use the Expect.Call() for methods that has return values, and LastCall for methods that return void to get the IMethodOptions interface. I find the Expect.Call() syntax a bit clearer, but there is no practical difference between the two. I would recommend using Expect wherever possible (anything that returns a value). For property setters, or methods returning void, the Expect syntax is not applicable, since there is no return value. Hence, the need for the LastCall. The idea of Last Call is pervasive in the record state, you can only set the method options for the last call - even Expect.Call() syntax is merely a wrapper around LastCall.

Expect.Call() & LastCall allow you to set the following options:

- **The return value of a method, if it has one.**

```
Expect.Call(view.Ask(null,null)).
  Return(null);
```

- **The exception the method will throw:**

```
Expect.Call(view.Ask(null,null)).
  Throw(new Exception("Demo"));
```

- **The number of times this method is expected to repeat (there are a number of convenience methods there):**

```
Expect.Call(view.Ask(null,null)).Return(null).
  Repeat.Twice();
```

- **To ignore the method arguments:**

```
Expect.Call(view.Ask(null,null)).Return(null).
  IgnoreArguments();
```

- **To set the constraints of the method:**

```
Expect.Call(view.Ask(null,null)).Return(null)
  .Constraints(Text.StartsWith("Some"),Text.EndsWith("Text"));
```

- **To set the callback for this method:**

```
Expect.Call(view.Ask(null,null)).Return(null).
  Callback(new AskDelegate(VerifyAskArguments));
```

- **To call the original method on the class:**

```
Expect.Call(view.Ask(null,null)).
CallOriginalMethod();
```

- **To emulate simple property accessors on a property:**

```
Expect.Call(view.Name).
          PropertyBehavior();
```

- **To control programmatically what the method call will return or throw (see The Do() Handler):**

```
Expect.Call(view.Ask(null,null)).Do(delegate(string s1, string s2) { return s1+s2; })
IgnoreArguments();
```

Note: For methods that return a value, you must specify either a return value or an exception to throw. You will not be able to continue recording or move to replay state otherwise.

Note II: Method chaining really makes writing this code easier.

Up: Rhino Mocks Documentation
Next: Rhino Mocks Setup Result

Rhino Mocks Setup Result

Sometimes you have a method on your mocked object which you don't care how / if it was called, you may want to set a return value (or an exception to be thrown), but for this specific test, you just don't care. For example, you may have some interaction that you've already verified, or you are testing some other class and just need to get the right value from a method. The way to do it in Rhino Mocks is to use SetupResult.For(). Here is the code:

```
Test
public void SetupResultUsingOrdered()
{
   SetupResult.For(demo.Prop).Return("Ayende");
   using(mocks.Ordered())
   {
       demo.VoidNoArgs();
       LastCall.On(demo).Repeat.Twice();
   }
   mocks.ReplayAll();
   demo.VoidNoArgs();
   for (int i = 0; i < 30; i++)
   {
       Assert.AreEqual("Ayende",demo.Prop);
   }
   demo.VoidNoArgs();
}
```

When we use SetupResult.For() we tell Rhino Mocks: "I don't care about this method, it needs to do X, so just do it, but otherwise ignore it." Using SetupResult.For() completely bypasses the expectations model in Rhino Mocks. In the above example, we define demo.Prop to return "Ayende", and we can call it no matter what the currently expected method is.

What about methods returning void?

You would use LastCall.Repeat.Any(), which has identical semantics (ignore ordering, etc).

The reverse of setup result is to specify that this method should never be called (useful if you're using dynamic mocks), which is done like this:

```
Expect.Call(view.Ask(null,null)).IgnoreArguments().
   .Repeat.Never();
```

This has the same semantics as ExpectNoCall() in NMock and this tells the framework that any call to this method is an error. Notice that I still call IgnoreArguments(), since otherwise the expectation that this method will not be called would've been for a method call with null as both parameters. Like setup result and Repeat.Any(), using Repeat.Never() transcend ordering.

Note: If you want to have a method that can repeat any number of time, but still obey ordering, you can use: LastCall.On().Repeat.Times(0,int.MaxValue), this does obey all the normal rules.

Up: Rhino Mocks Documentation
Next: Rhino Mocks Mocking classes

Rhino Mocks Mocking classes

Rhino Mocks supports mocking classes as well as interfaces. In fact, it can even mock classes that don't have a default constructor! To mock a class, simply pass its type to MockRepository.CreateMock() along with any parameters for the constructor.

To create a stub on a class. This uses the GenerateStub method for creating the mock. Use this ONLY if you do not wish to set expectations on your mock. You only want to use the mock as a STUB

```
[Test]
public void AbuseArrayList_UsingGenerateStub()
{
    ArrayList list = MockRepository.GenerateStub<ArrayList>();

    // Evaluate the values from the mock
    Assert.AreEqual( 0, list.Capacity );
}
```

Creating a mock on a class using Generics Use this if you are using .Net 2.0

```
[Test]
public void AbuseArrayList_UsingCreateMockGenerics()
{
    MockRepository mocks = new MockRepository();
    ArrayList list = mocks.CreateMock < ArrayList >();

    // Setup the expectation of a call on the mock
    Expect.Call( list.Capacity ).Return( 999 );
    mocks.ReplayAll();

    // Evaluate the values from the mock
    Assert.AreEqual( 999, list.Capacity );
    mocks.VerifyAll(); ;
}
```

Use this alternative syntax if you are using .Net 1.1

```
[Test]
public void AbuseArrayList_UsingCreateMockGenerics()
{
    MockRepository mocks = new MockRepository();
    ArrayList list = (ArrayList) mocks.CreateMock( typeof ( ArrayList ) );

    // Setup the expectation of a call on the mock
    Expect.Call( list.Capacity ).Return( 999 );
    mocks.ReplayAll();

    // Evaluate the values from the mock
    Assert.AreEqual( 999, list.Capacity );
    mocks.VerifyAll(); ;
}
```

If you want to call the non-default constructor, just add the parameters after the type.

Like this:

// Non-Generics

```
ArrayList list = (ArrayList)mocks.CreateMock(typeof(ArrayList),500);
```

// With Generics

```
ArrayList list = mocks.CreateMock< ArrayList >( 500 );
```

Up: Rhino Mocks Documentation

Next: Rhino Mocks - Internal Methods

Rhino Mocks Internal Members

You can also use Rhino Mocks to mock internal classes or methods - for example:

```
internal class Class1
{
	internal virtual string TestMethod()
	{
		return "TestMethod";
	}

	internal virtual string TestProperty
	{
		get { return "TestProperty"; }
	}
}
```

You can mock both the TestMethod and TestProperty inside your unit tests (the example is for .Net 2.0 onwards):

```
[Test]
public void MockingInternalMethodsAndPropertiesOfInternalClass()
{

	Class1 testClass = new Class1();
	string testMethod = testClass.TestMethod();
	string testProperty = testClass.TestProperty;
	MockRepository mockRepository = new MockRepository();

	Class1 mockTestClass = mockRepository.CreateMock<Class1>();
	Expect.Call(mockTestClass.TestMethod()).Return("MockTestMethod");
	Expect.Call(mockTestClass.TestProperty).Return("MockTestProperty");

	mockRepository.ReplayAll();

	Assert.AreEqual("MockTestMethod", mockTestClass.TestMethod());
	Assert.AreEqual("MockTestProperty", mockTestClass.TestProperty);

	mockRepository.VerifyAll();
}
```

When a class is mocked, a new class is generated at run-time which is derived from the mocked class. This generated class resides in a separate "temporary" assembly which is called "DynamicProxyGenAssembly2". So, the InternalsVisibleTo attribute needs to be set on the target assembly to allow access to its internal members from the temporary assembly; otherwise, the mock object can't override the internal member as it doesn't have access to it (which is also why the mocked method must be marked as virtual). Note that this is true even if the unit test and the tested class are in the same assembly.

So, you need to make sure that the target class' assembly makes its internals visible to the proxy assembly as such (in AssemblyInfo.cs for example):

```
[assembly: InternalsVisibleTo ("DynamicProxyGenAssembly2")]
```

If you are using strong-named assemblies, this can be achieved as such:

```
[assembly: InternalsVisibleTo(Rhino.Mocks.RhinoMocks.StrongName)]
```

The RhinoMocks.StrongName is a convenient way to avoid copying and pasting the full strong name for DynamicProxyGenAssembly2, at the cost of adding a dependency on Rhino.Mocks to your main assembly. If you want to avoid this, you can also use the contents of the RhinoMocks.StrongName string rather than referencing the field:

```
[assembly: InternalsVisibleTo ("DynamicProxyGenAssembly2, PublicKey=0024000004800000940000000602000000240000525341310004000001000100c547cac37abd99c
```

This method is perhaps less convenient insofar as the public key could change from on release to the next. Note that the key I pasted above is correct for RhinoMocks 3.0.1 (and should be valid for the 3.0.x series), but could be different in previous or further releases.

Up: Rhino Mocks Documentation
Next: Rhino Mocks Record-playback Syntax

Rhino Mocks Record-playback Syntax

Rhino Mocks has an alternative syntax to define and replay expectations. The syntax makes heavy use of the *using* keyword in C#:

```
// Prepare mock repository
MockRepository mocks = new MockRepository();
IDependency dependency = mocks.CreateMock<IDependency>();

// Record expectations
using ( mocks.Record() )
{
    Expect
        .Call( dependency.SomeMethod() )
        .Return( null );
}

// Replay and validate interaction
object result;
using ( mocks.Playback() )
{
    IComponent underTest = new ComponentImplementation( dependency );
    result = underTest.TestMethod();
}

// Post-interaction assertions
Assert.IsNull( result );
```

This is a simplified example; as you can see, the expectations are recorded in one scope which also takes care (courtesy of the *using* keyword) of calling *mocks.ReplayAll()*. The interaction testing itself takes place in another scope, after which (again via *using*) a call is made to *mocks.VerifyAll()*.

TBA: limitations and caveats of this technique. --TG

Up: Rhino Mocks Documentation
Next: Rhino Mocks With-Syntax

Rhino Mocks With-Syntax

A "fluent" way of expectation specifying and verifying

An alternative way of specifying expectations for a block of code to be verified, is the *With* class. It provides some kind of fluent interface and is a little bit more compact than the Rhino Mocks Record-playback Syntax, but comes with the drawback of using anonymous delegates.

```
// Prepare mock repository
MockRepository mocks = new MockRepository();
IDependency dependency = mocks.CreateMock<IDependency>();

object result;

With.Mocks( mocks ).Expecting( delegate
{
    // Record expectations
    Expect.Call( dependency.SomeMethod() ).Return( null );
})
.Verify(delegate
{
    // Replay and validate interaction
    IComponent underTest = new ComponentImplementation( dependency );
    result = underTest.TestMethod();
});

// Post-interaction assertions
Assert.IsNull( result );
```

***ReplayAll()* and *VerifyAll()* are implicitly called, when executing the code passed in as anonymous delegates. To take care of the order of expectations, *ExpectingInSameOrder()* can be used:**

```
// Prepare mock repository
MockRepository mocks = new MockRepository();
IDependency dependency = mocks.CreateMock<IDependency>();
IAnotherDependency anotherDependency = mocks.CreateMock<IAnotherDependency>();

object result;

With.Mocks( mocks ).ExpectingInSameOrder( delegate
{
    // Record expectations which must be met in the exact same order
    Expect.Call( dependency.SomeMethod() ).Return( null );
    anotherDependency.SomeOtherMethod();
})
.Verify(delegate
{
    // Replay and validate interaction
    IComponent underTest = new ComponentImplementation( dependency, anotherDependency );
    result = underTest.TestMethod();
});

// Post-interaction assertions
Assert.IsNull( result );
```

A wrapper to ensurer VerifyAll()

This kind of syntax allows the MockRepository to be created on the fly and automatically call *VerifyAll()* at the end. The MockRepository is accessible through the static property *Mocker.Current*.

```
   With.Mocks(delegate
   {
        IDemo demo = Mocker.Current.CreateMock<IDemo>();
        Expect.Call(demo.ReturnIntNoArgs()).Return(5);
        Mocker.Current.ReplayAll();
        Assert.AreEqual(5, demo.ReturnIntNoArgs());
    });
```

Instead of implicitly creating a new one, a specific MockRepository can be used too:

```
MockRepository mocks = new MockRepository();
With.Mocks(mocks, delegate
{
     IDemo demo = Mocker.Current.CreateMock<IDemo>();
     Expect.Call(demo.ReturnIntNoArgs()).Return(5);
     Mocker.Current.ReplayAll();
     Assert.AreEqual(5, demo.ReturnIntNoArgs());
 });
```

When using Mocker.Current, be aware, that you can't use the above syntax in nested constructs, because the global Mocker.Current will be override with each *With.Mocks()* call.

Up: Rhino Mocks Documentation
Next: Rhino Mocks Limitations

Rhino Mocks Limitations

Some things to be aware of:

- **You cannot create a mock object from a sealed class.**
- **You cannot create a mock object from a private interface.**
- **You cannot intercept calls to non-virtual methods.**

Up: Rhino Mocks Documentation
Next: no