**Rhino Mocks 3.5**

**http://ayende.com/wiki/Rhino+Mocks+3.5.ashx**

[Print](http://ayende.com/wiki/Print.aspx?Page=Rhino+Mocks+3.5)

[RSS](http://ayende.com/wiki/RSS.aspx?Page=Rhino+Mocks+3.5)

Modified: 2011/08/22 12:37 by [admin](http://ayende.com/wiki/Message.aspx?Username=admin) - [*Uncategorized*](http://ayende.com/wiki/AllPages.aspx?Cat=-)

» [**Rhino Mocks 3.5**](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx)

[[X]](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx) » [**Rhino Mocks 3.5**](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx)

Rhino Mocks 3.5 introduces some new concepts, mostly by building upon the language features of C# 3.0. Users on the 2.0 platform need not worry, there are enough goodies for them as well (inline constraints, for example), but the focus of this document is on the use of Rhino Mocks with C# 3.0.

**Table of Contents** [[Hide/Show](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx)]

[**What's New in Rhino Mocks 3.5**](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#WhatsNewinRhinoMocksDF)  
[**Usage Guidance**](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#UsageGuidance)  
   [The difference between stubs and mocks](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#Thedifferencebetweenstubsandmocks)  
   [CreateMock is deprecated, replaced by StrictMock. The use of Strict Mock is discouraged.](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx" \l "CreateMockisdeprecated,replacedbyStrictMockTheuseofStrictMockisdiscouraged)  
[**Mocking with and without an instance of MockRepository**](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#MockingwithandwithoutaninstanceofMockRepository)  
[**Arrange, Act, Assert**](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#Arrange,Act,Assert)   
   [Expect and Stub Extension Methods](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#ExpectandStubExtensionMethods)  
      [Expect() Extension Method](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#ExpectExtensionMethod)  
      [Stub() Extension Method](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#StubExtensionMethod)  
      [Using Expect() to set up properties](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#UsingExpecttosetupproperties)  
   [What is the difference between GenerateStub and GenerateMock](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#WhatisthedifferencebetweenGenerateStubandGenerateMock)  
[**Using the AAA syntax in C# 2.0**](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#UsingtheAAAsyntaxinCCA)  
[**Argument Constraints**](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#ArgumentConstraints)  
   [Simple Constraints](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#SimpleConstraints)  
   [Inline constraints](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#Inlineconstraints)  
      [Ignoring arguments](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#Ignoringarguments)  
      [Shortcut to Equal](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#ShortcuttoEqual)  
      [Properties](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#Properties)  
      [Event registration](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#Eventregistration)  
      [Complex expressions](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#Complexexpressions)  
      [Out and Ref arguments](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#OutandRefarguments)  
      [Constraints Reference](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#ConstraintsReference)  
[**How to raise events**](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#Howtoraiseevents)  
[**Property Setters Explicit Expectation API**](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#PropertySettersExplicitExpectationAPI)  
[**Accessing the method arguments directly**](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#Accessingthemethodargumentsdirectly)  
[**Recursive mocking**](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx#Recursivemocking)

[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=16)

**What's New in Rhino Mocks 3.5**

New things in Rhino Mocks 3.5:

* Arrange, Act, Assert model
* Lambda and C# 3.0 extensions
* Inline constraints
* Property Setters Explicit Expectation API
* Support for mocking interface in C++ that mix native and managed types.
* Allow a mock object to return to record mode without losing its expectations
* CreateMock was deprecated in favor of StrictMock
* Better error handling in edge cases.
* Fixed an issue with mocking internal classes and interfaces
* New event raising syntax

[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=17)

**Usage Guidance**

In general, I recommend following the principle of "Test only one thing per test". Applying this to Rhino Mocks (and mocking in general), each unit test should validate no more than one significant interaction with another object. This generally implies that a given test should have no more than one mock object, but it may have several stubs, as needed. (An exception to this would be if the "one thing" tested inherently requires expectations on two dependent objects, for example, verifying that a method on one object is only called after a method on another object has been called.)

[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=10)

**The difference between stubs and mocks**

You can get the actual definition of the these terms in this article: [Mocks Aren't Stubs](http://martinfowler.com/articles/mocksArentStubs.html). I want to focus on the difference from the point of view of Rhino Mocks.  
  
A mock is an object that we can set expectations on, and which will verify that the expected actions have indeed occurred. A stub is an object that you use in order to pass to the code under test. You can setup expectations on it, so it would act in certain ways, but those expectations will never be verified. A stub's properties will automatically behave like normal properties, and you can't set expectations on them.  
  
If you want to verify the behavior of the code under test, you will use a mock with the appropriate expectation, and verify that. If you want just to pass a value that may need to act in a certain way, but isn't the focus of this test, you will use a stub.  
  
IMPORTANT: A stub will never cause a test to fail.

Let us create a sample test, which will give us more context to talk about. We want to write a test for sending an SMS for a forgotten password. The code should:

* Get the user from the repository
* Reset the password to a random one
* Save the user with the new password
* Send an SMS with the new password

Note that I am using the new syntax here which will be explained shortly

First, we want to test that we reset the password, so we will write the following test:

public void When\_user\_forgot\_password\_should\_reset\_password()

{

var stubUserRepository = MockRepository.GenerateStub<IUserRepository>();

var stubbedSmsSender = MockRepository.GenerateStub<ISmsSender>();  
  
 var theUser = new User{HashedPassword = "this is not hashed password"};   
  
 stubUserRepository.Stub(x => x.GetUserByName("ayende")).Return(theUser);

这个单元测试测的函数是LoginController 类下的void ForgotMyPassword(string username)，在这个函数里会调用UserRepository类（实现IUserRepository借口）的GetUserByName（string username）函数。但是你不想管GetUserByName（string username）的具体实现，要真的调用也会花费大量时间，所以stubUserRepository.Stub(x => x.GetUserByName("ayende")).Return(theUser);是安排好，给定GetUserByName（）输入参数"ayende"，的返回对象和这个对象的属性，如果安装预期的话，ForgotMyPassword（）函数执行后theUser.HashedPassword会被从"this is not hashed password“修改为“new pass”。  
  
 var controllerUnderTest = new LoginController(stubUserRepository, stubbedSmsSender);

LoginController类的构造函数是LoginController(IUserRepository xxx, ISmsSender yyy)，使用stub对象而不是直接new出来对象，我想主要也是为了节省new的时间，谁知道new对应的构造函数里还有多少功能。  
  
 controllerUnderTest.ForgotMyPassword("ayende");  
  
 Assert.AreNotEqual("this is not hashed password", theUser.HashedPassword);

}

In this test, we have created two stubs, passed them to the code under test, and asserted that the password was changed. To make this code pass we can use:

public void ForgotMyPassword(string username)

{

var user = users.GetUserByName(username);

user.HashedPassword = "new pass";

}

Now we want to verify that the user will save the password, so we have the following test:

public void When\_user\_forgot\_password\_should\_save\_user()

{

var mockUserRepository = MockRepository.GenerateMock<IUserRepository>();

var stubbedSmsSender = MockRepository.GenerateStub<ISmsSender>();  
  
 var theUser = new User{HashedPassword = "this is not hashed password"};   
  
 mockUserRepository.Stub(x => x.GetUserByName("ayende")).Return(theUser);  
  
 mockUserRepository.Expect( x => x.Save(theUser) );

Stub是一种设置，而Expect是预期，这个预期是需要Verify的。注意这个调用Expect的对象一般都是mock对象，而非stub对象。  
  
 var controllerUnderTest = new LoginController(mockUserRepository, stubbedSmsSender);  
  
 controllerUnderTest.ForgotMyPassword("ayende");  
  
 mockUserRepository.VerifyAllExpectations();

}

Now we have used a mock object and set an expectations on it. Note that even though we are using a mock object here, we are still stubbing the call to GetUserByName. We don't care about this, since this is not what we are testing.  
  
The code under test now looks like:

public void ForgotMyPassword(string username)

{

var user = users.GetUserByName(username);

user.HashedPassword = "new pass";

users.Save(user);

}

Another way to write the test would have been:

public void When\_user\_forgot\_password\_should\_save\_user()

{

var stubUserRepository = MockRepository.GenerateStub<IUserRepository>();

var stubbedSmsSender = MockRepository.GenerateStub<ISmsSender>();  
  
 var theUser = new User{HashedPassword = "this is not hashed password"};   
  
 stubUserRepository.Stub(x => x.GetUserByName("ayende")).Return(theUser);  
  
 var controllerUnderTest = new LoginController(stubUserRepository, stubbedSmsSender);  
  
 controllerUnderTest.ForgotMyPassword("ayende");  
  
 stubUserRepository.AssertWasCalled( x => x.Save(theUser));

注意AssertWasCalled就不需要，设置Expect了，所以是stub对象，而非mock对象。VerifyAllExpections，就需要设置Expect，否则verify的时候，不知道要verify谁。

}

We have no expectations in this test, just setting the required state and asserting on the actions that occurred when the code under test run. The difference between the two test is subtle, but important. In most cases, you should prefer to use stubs. Only when you are testing complex interactions would I recommend to use mock objects. If you need to ensure ordering, or expect to get a complex method, or want to ensure an exact adherence to the expectations.  
  
Note that so far, we haven't even cared about the actual sending on an SMS, let us write a test that ensure that we can send an SMS.

public void When\_user\_forgot\_password\_should\_sms\_user()

{

var stubUserRepository = MockRepository.GenerateStub<IUserRepository>();

var stubbedSmsSender = MockRepository.GenerateStub<ISmsSender>();  
  
 var theUser = new User{HashedPassword = "this is not hashed password", Phone = "1234-1234"};   
  
 stubUserRepository.Stub(x => x.GetUserByName("ayende")).Return(theUser);  
  
 var controllerUnderTest = new LoginController(stubUserRepository, stubbedSmsSender);  
  
 controllerUnderTest.ForgotMyPassword("ayende");  
  
 stubbedSmsSender.AssertWasCalled( x => x.Send(

Arg.Is.Equal("1234-1234"),

Arg.Text.StartsWith("Password was changed to:")

));

}

Here we are asserting that the method was called, and use the inline constraint support to ensure we pass the correct arguments. The code under test now looks like:

public void ForgotMyPassword(string username)

{

var user = users.GetUserByName(username);

user.HashedPassword = "new pass";

users.Save(user);

smsSender.Send(user.Phone, "Password was changed to: new pass");

}

It is important to remember that we don't want to be tied down by our tests. We want to make sure that even if we modify the code under test, we will not break any unrelated tests.  
  
[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=11)

**CreateMock is deprecated（反对的）, replaced by StrictMock. The use of Strict Mock is discouraged.**

In Rhino Mocks 3.4 and previous, the "default" way to get a mock object was to call mockRepository.CreateMock<ISmsSender>(). That caused a very serious issue. CreateMock() would return a strict mock, and as we have already discussed, an overly strict mock will create a brittle test. As a result of that, CreateMock() was deprecated and will generate a compiler warning if used. You can replace calls to CreateMock() with StrictMock(). However, using StrictMock() is discouraged. Strict mocks will fail if something that is not expected will happen to them. In the long run, this means that any change to the code under test can break your tests, even if the change has nothing to do with what you are actually testing in this specific test.  
  
I encourage the use of stubs and dynamic mocks instead.  
  
[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=18)

**Mocking with and without an instance of MockRepository**

Rhino Mocks 3.5 introduce a new mode of mocking, using mocks without creating the repository. The purpose of that is to reduce the amount of book keeping that you have to do. Here is a test without the repository:

public void When\_user\_forgot\_password\_should\_save\_user()

{

var mockUserRepository = MockRepository.GenerateMock<IUserRepository>();

var stubbedSmsSender = MockRepository.GenerateStub<ISmsSender>();  
  
 var theUser = new User{HashedPassword = "this is not hashed password"};   
  
 mockUserRepository.Stub(x => x.GetUserByName("ayende")).Return(theUser);  
  
 mockUserRepository.Expect( x => x.Save(theUser) );  
  
 var controllerUnderTest = new LoginController(mockUserRepository, stubbedSmsSender);  
  
 controllerUnderTest.ForgotMyPassword("ayende");  
  
 mockUserRepository.VerifyAllExpectations();

}

With a repository, the test would look like:

public void When\_user\_forgot\_password\_should\_save\_user()

{

var mocks = new MockRepository();

如果这么做，就需要录像回放，而不是像上面那样用3A

var mockUserRepository = mocks.DynamicMock<IUserRepository>();

var stubbedSmsSender = mocks.GenerateStub<ISmsSender>();  
  
 using(mocks.Record())

{

var theUser = new User{HashedPassword = "this is not hashed password"};   
  
 mockUserRepository.Stub(x => x.GetUserByName("ayende")).Return(theUser);  
  
 mockUserRepository.Expect( x => x.Save(theUser) );

}

using(mocks.Playback())

{  
  
 var controllerUnderTest = new LoginController(mockUserRepository, stubbedSmsSender);  
  
 controllerUnderTest.ForgotMyPassword("ayende");

}

}

There are a few other differences between the two approaches:

* Mocks/stubs returned from MockRepository.GenerateMock() and MockRepository.GenerateStub() are returned in replay mode, and do not require explicit move to replay mode.
* Only Dynamic Mocks and Stubs are available without creating the repository. (This is done under the assumption that thos e are the most commonly used, and deserve their shortcut).
* Without creating a repository, you need to call VerifyAllExpectations() on each of the mock objects that you created. If you have a repository, you can simply call VerifyAll() on the repository and it will do the work for you. Since in most situations you will have only a single mock per test (and stubs require no verifications), that is not an issue.

[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=19)

**Arrange, Act, Assert**

*The* major feature in Rhino Mocks 3.5 is the AAA syntax. Arrange, Act, Assert is a classic way to set up your tests. First you arrange the state, then you execute the code under test, finally you assert that the expected state change happened. With interaction based testing, this was generally impossible, and required that you would use the Record/Replay model. This was a common case for confusion for people new to interaction based testing. The AAA syntax solve this problem.  
  
Let us take a look at a simple test and analyze it using the AAA approach:

public void When\_user\_forgot\_password\_should\_save\_user()

{

// arrange

var stubUserRepository = MockRepository.GenerateStub<IUserRepository>();

var stubbedSmsSender = MockRepository.GenerateStub<ISmsSender>();  
  
 var theUser = new User{HashedPassword = "this is not hashed password"};   
  
 stubUserRepository.Stub(x => x.GetUserByName("ayende")).Return(theUser);

// act

var controllerUnderTest = new LoginController(stubUserRepository, stubbedSmsSender);  
  
 controllerUnderTest.ForgotMyPassword("ayende");  
  
 // assert

stubUserRepository.AssertWasCalled( x => x.Save(user));

}

We can easily see the different stages of the test. The main supporters for this method of operation are the Expect() and Stub() methods.  
  
[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=12)

**Expect and Stub Extension Methods**

The Expect() and Stub() methods are extension methods that become available for use on any mock object when you reference (*using* in C# or *Imports* in VB.NET) the Rhino.Mocks namespace. The Stub() extention method is available for stub objects.  
  
[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=0)

**Expect() Extension Method**

The Expect() extension method creates a new expectation for this mock which must be verified/asserted later (using either the VerifyAllExpectations() or AssertWasCalled() methods).  
  
Expect() functions similarly to the Expect.Call() functionality from previous versions of Rhino Mocks, except the .Expect() extension method can be used without a Record() block (whereas Expect.Call() must be called from within an explicit Record() block).  
  
Consider the following example usage which ensures that the controller *SignOut* action calls the *authSvc's* *SignOut* method:

[Test](http://ayende.com/wiki/Test.ashx)

public void should\_sign\_out\_of\_the\_auth\_service\_when\_the\_logout\_action\_is\_invoked()

{

IAuthService authSvc = MockRepository.GenerateMock();  
  
 authSvc.Expect(svc => svc.SignOut()).Return(true);  
  
 SignoutControllercontroller = new SignoutController(authSvc);

controller.SignOut();  
  
 authSvc.VerifyAllExpectations();

}

[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=1)

**Stub() Extension Method**

The Stub() extension method will stub a specific member so that it will perform the default stub behavior (i.e. not fail the test -- *for specific behavior, please see definition of stub earlier in this document*) for that particular member.   
  
You should use this behavior if you need to influence the result of the code in some fashion. Remember that by default Rhino Mocks will ignore unexpected methods calls. A good example for that is the following code:

public void SignOut()

{

if (authSvc.Signout())

notificationSvc.UserIsLoggedOut();

}

Here is how we can test this. We stub the call to the SignOut and tell it to return true, then we expect a call to UserIsLoggingOut().

[Test](http://ayende.com/wiki/Test.ashx)

public void should\_notify\_when\_user\_sign\_out\_successfully()

{

IAuthService authSvc = MockRepository.GenerateStub<IAuthService>();

INotificationService notificationSvc= MockRepository.GenerateMock<INotificationService>();  
  
 authSvc.Stub(svc => svc.SignOut()).Return(true); // This also works for mock objects

notificationSvc.Expect((o => o.UserIsLoggedOut());

SignoutController controller = new SignoutController(authSvc, notificationSvc);

controller.SignOut();  
  
 notificationSvc.VerifyAllExpectations();

}

Another way to write this test is not to use explicit expectation, but rather to assert that a particular method call was called.

[Test](http://ayende.com/wiki/Test.ashx)

public void should\_notify\_when\_user\_sign\_out\_successfully()

{

IAuthService authSvc = MockRepository.GenerateStub<IAuthService>();

INotificationService notificationSvc= MockRepository.GenerateMock<INotificationService>();  
  
 authSvc.Stub(svc => svc.SignOut()).Return(true);

SignoutController controller = new SignoutController(authSvc, notificationSvc);

controller.SignOut();  
  
 notificationSvc.AssertWasCalled(x=>x.UserIsLoggedOut());

}

[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=2)

**Using Expect() to set up properties**

The Expect() extention method can be used to set up expectations and return values for properties.  
  
The following example, from a Model-View-Controller project, tests the RunView() method of the controller. A property getter in the view is mocked with the expectation that it will be called.

public void RunView\_Returns\_the\_View\_List

{

//Arrange

List<string> expectedlist = new List<string>();

expectedlist.Add("Ayende");

expectedlist.Add("Tom");

Form Parent = null; //Stub, the parent doesn't matter for this test.

IGetNamesView mockview = MockRepository.GenerateMock<IGetNamesView>();

mockview.Expect(view => view.ShowDialog(Parent)).Return(DialogResult.OK);  
  
 mockview.Expect(view => view.Names).Return(expectedlist); //Mock for the property getter  
  
 GetNameController target = new GetNameController(Parent, mockview);

List<string> actual;  
  
 //Act

actual = target.RunVeiw();  
  
 //Assert

mockview.VerifyAllExpectations();

Assert.AreEqual(expectedlist, actual);

}  
  
[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=13)

**What is the difference between GenerateStub and GenerateMock**

GenerateMock() is equivalent to calling DynamicMock() on a MockRepository instance in previous versions of Rhino Mocks. GenerateMock will create a dynamic mock (as opposed to a static mock or stub).  
  
GenerateStub() is equivalent to calling Stub() on a MockRepository instance in previous versions of Rhino Mocks. GenerateStub will create a stubbed instance which will have stub behavior (see earlier in this article for the definition of Stub and explanation of Stub semantics).  
  
[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=20)

**Using the AAA syntax in C# 2.0**

Using the AAA syntax on C# 2.0 is possible, we just need to understand how the compiler manages extension methods. Here is an example of the previous two tests using C# 2.0:

[Test](http://ayende.com/wiki/Test.ashx)

public static void Test\_Using\_Extension\_Methods\_Using\_2\_0()

{

IAuthService authSvc = MockRepository.GenerateStub();

INotificationService notificationSvc = MockRepository.GenerateMock();  
  
 RhinoMocksExtensions.Expect(notificationSvc , delegate(INotificationService o)

{

o.UserIsLoggingOut();

});

RhinoMocksExtensions.Stub(authSvc, delegate(IAuthService svc)

{

svc.SignOut();

}).Return(true);  
  
 SignoutController controller = new SignoutController(authSvc, notificationSvc);

controller.SignOut();  
  
 RhinoMocksExtensions.VerifyAllExpectations(notificationSvc);  
  
}  
  
[Test](http://ayende.com/wiki/Test.ashx)

public void Test\_Using\_AAA\_Using\_2\_0()

{

IAuthService authSvc = MockRepository.GenerateStub();

INotificationService notificationSvc = MockRepository.GenerateMock();  
  
 RhinoMocksExtensions.Stub(authSvc, delegate(IAuthService svc)

{

svc.SignOut();

}).Return(true);  
  
 SignoutController controller = new SignoutController(authSvc, notificationSvc);

controller.SignOut();  
  
 RhinoMocksExtensions.AssertWasCalled(notificationSvc, delegate(INotificationService x)

{

x.UserIsLoggingOut();

});  
  
}

The syntax isn't as nice as what we get with C# 3.0, but we have all the functionality even for C# 2.0 :-)  
  
[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=21)

**Argument Constraints**

Argument constraints belong to the definition of an expected call or stubbed call. Only if the argument constraints match, the specification of the call takes effect (this means: the expectation gets fulfilled, the return value gets returned, the Exception gets thrown, the Do gets executed and so on.) If the constraints do not match, it is as if another method has been called.  
  
Argument constraints also define the method's signature by specifying the arguments types. That's why they can't be omitted. You can't define an expectation without specifying argument constraints, because you wouldn't specify the methods signature.  
  
[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=14)

**Simple Constraints**

The simplest way to specify argument constraints is by directly using the value, as you see often in examples:

stubUserRepository.Stub(x => x.GetUserByName(**"ayende"**)).Return(theUser);

The string value "ayende" is a constraint to the stub specification. Calls to GetUserByName with other arguments are not specified in this case.

stubUserRepository.Stub(x => x.GetUserByName(**"ayende"**)).Return(theUser);

Assert.IsNull(stubUserRepository.GetUserByName(**"stefan"**));

Assert.AreSame(theUser, stubUserRepository.GetUserByName(**"ayende"**));

You can still specify that all argument values are accepted. To do this, use IgnoreArguments().

stubUserRepository.Stub(x => x.GetUserByName(**null**))

**.IgnoreArguments()**

.Return(theUser);

This simple 'directly-using-a-value' syntax is discouraged for various reasons. It is very limited. You can only define constraints that are compared using Equals, which is normally a reference comparison for reference types. You could also easily get problems because your value does not clearly specify a Type, For instance the **null** used in the example before does not specify a type. If the method has been overridden with arguments of different Types, you actually can't say which method you specify. (You could say for instance (string)null, which most users would not find intuitive.) Use inline constraints instead.  
  
[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=15)

**Inline constraints**

**Note to users of previous versions of Rhino Mocks:** You're probably used to IgnoreArguments(), Constraints() and RefOut(). Rhino Mocks 3.5 introduces a new syntax for constraints, called the inline constraints, using Arg<T>. You can do everything with Arg<T> you could do before. It is encouraged to use only Arg<T>, it is more consistent and easier to understand, even if sometimes a little more to write.

Here is the previous example with inline constraints:

stubUserRepository.Stub(x => x.GetUserByName(**Arg<string>.Is.Equal("ayende")**)).Return(theUser);

Assert.IsNull(stubUserRepository.GetUserByName(**"stefan"**));

Assert.AreSame(theUser, stubUserRepository.GetUserByName(**"ayende"**));

For a first look, it might just look longer. You have to specify the type, which you might think is needless. Remember, you are specifying the method's signature at the same time.   
  
You are not limited to an Equal comparison. See the constraints reference bellow.  
  
Note: If you use inline constraints, all the method's arguments must be defined using inline constraints. You can't mix it with plain values.  
  
[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=3)

**Ignoring arguments**

stubUserRepository.Stub(x => x.GetUserByName(**Arg<string>.Is.Anything**)).Return(theUser);

Assert.AreSame(theUser, stubUserRepository.GetUserByName(**"stefan"**));

Assert.AreSame(theUser, stubUserRepository.GetUserByName(**"ayende"**));

[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=4)

**Shortcut to Equal**

There is also the shortcut Arg.Is<T>(T) which actually is the same as Arg<T>.Is.Equal(object). In .NET 3.5 you don't need to specify the generic argument, which results in this nice short syntax:

stubUserRepository.Stub(x => x.GetUserByName(**Arg.Is("Ayende")**)).Return(theUser);

[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=5)

**Properties**

You can easily specify constraints for values assigned to properties.

customerMock.Name = "Steinegger";

customerMock.AssertWasCalled(x => x.Name = **Arg<string>.Is.NotNull**);

Note, to expect that the property has been read, you need to assign the property to some variable inside an anonymous delegate.

string name = customerMock.Name;

customerMock.AssertWasCalled(x => {var ignored = x.Name;});

[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=6)

**Event registration**

mock.Load += OnLoad;

mock.AssertWasCalled(x => x.Load += **Arg<LoadEvent>.Is.Anything**);

[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=7)

**Complex expressions**

Arg<T>.Matches allows you to define complex expressions. There are two overloads of this method. One allows you to define a lambda expression (.NET 3.5). The other allows you to define a complex expressions using classical [Rhino Mocks Constraints](http://ayende.com/wiki/Rhino+Mocks+Constraints.ashx).

// using lamda with .NET string evaluation (string.StartsWith)  
  
stubUserRepository.Stub(x =>

x.GetUserByName(**Arg<string>.Matches(y =>**

**y.StartsWith("aye", StringComparison.InvariantCulture)**

**|| y.StartsWith("stein", StringComparison.InvariantCulture)**))

.Return(theUser);

Assert.AreSame(theUser, stubUserRepository.GetUserByName(**"steinegger"**));

Assert.AreSame(theUser, stubUserRepository.GetUserByName(**"ayende"**));

//using Rhino Constraints to build the expression  
  
stubUserRepository.GetUserByName("ayende");

stubUserRepository.GetUserByName("steinegger");  
  
stubUserRepository.Stub(x =>

x.GetUserByName(**Arg<string>.Matches(Text.StartsWith("aye") || Text.StartsWith("stein")**))

.Return(theUser);

Assert.AreSame(theUser, stubUserRepository.GetUserByName(**"steinegger"**));

Assert.AreSame(theUser, stubUserRepository.GetUserByName(**"ayende"**));

[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=8)

**Out and Ref arguments**

Ref and out arguments are special, because you also have to make the compiler happy. The keywords **ref** and **out** are mandantory, and you need a field as argument. Arg won't let you down:

User user;

if (stubUserRepository.TryGetValue("Ayende", out user))

{

//...

}

stubUserRepository.Stub(x =>

x.TryGetValue(

Arg.Is("Ayende"),

**out Arg<User>.Out(new User()).Dummy**))

.Return(true);

**out** is mandantory for the compiler. **Arg.Out(new User())** is the important part for us, it specifies that the out argument should return new User(). **Dummy** is just a field of the specified type User, to make the compiler happy.  
  
It looks similar with ref arguments:

User user;

if (someMock.Foo(ref str))

{

//...

}

someMock.Stub(x =>

x.Foo(**ref Arg<string>.Ref(Is.Equal("Stefan"), "Oren").Dummy**))

.Return(true);

Again, **ref** is mandantory. **Arg<string>.Ref(Is.Equal("Stefan"), "Oren")** defines both the constraint and the return value. It uses "classical" [Rhino Mocks Constraints](http://ayende.com/wiki/Rhino+Mocks+Constraints.ashx). **Dummy** is again the field for the compiler.  
  
[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=9)

**Constraints Reference**

|  |
| --- |
| **Arg<T>.Is** |
| Equal(object), NotEqual(object) | Comparison using Equals |
| GreaterThan(object), LessThan(object), LessThanOrEqual(object), GreaterThanOrEqual(object) | Comparison using >, <, >= and <= operators |
| Same(object), NotSame(object) | Reference equality |
| Anything() | No constraints |
| Null(), NotNull() | Reference is null or not null |
| TypeOf(Type), TypeOf<T>() | Argument is of a certain type |
| Matching<T>(Predicate<T>) | Argument matches a predicate (.NET 3.5: see Arg<T>.Matches). Example: Arg<string>.Is.Matching(delegate(string s) { return s.Length == 2; } |
| IsIn(object) | Enumerable argument includes the specified object |
| **Arg<T>.List** |  |
| OneOf(IEnumerable) | Argument is in the specified list |
| Equal(IEnumerable) | All items in the enumerable argument are compared to the items in the specified list. |
| Count(AbstractConstraint) | Constraints to the Count property of the argument. |
| Element(int, AbstractConstraint) | Element at the specified index meets the constraint. |
| ContainsAll(IEnumerable) | The enumerable argument contains at least the specified items. |
| **Arg<T>.Property** |  |
| AllPropertiesMatch(object) | All the public properties and public fields are compared to the properties in the specified object. The comparesion is recusive if public properties or fields are complex types. |
| IsNotNull(string propertyName), IsNull(string propertyName) | Property of the given name is or is not null |
| Value(string propertyName, object expectedValue) | The property of the given name is equal to the expected value. |
| Value(string propertyName, AbstractConstraint constraint) | Property of the given Name meets the constraint. |
| **Arg.Text** |  |
| StartsWith(string), EndsWith(string), Contains(string) | String starts with, ends with or contains the specified text |
| Like(string regex) | Property matches to the regular expression |
| **Arg<T>.Matches** |  |
| Argt<T>.Matches(Expression) | Only in .NET 3.5 you can specify the constraint as a lambda expression, e.g. Arg<int>.Matches(x => x > 3) |
| Argt<T>.Matches(AbstractConstraint) | Specify [Rhino Mocks Constraints](http://ayende.com/wiki/Rhino+Mocks+Constraints.ashx), e.g. Arg<int>.Matches(Is.GreaterThan(0) && Is.LessThan(10)) |

[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=22)

**How to raise events**

Rhino Mocks 3.5 introduces a new way of raising events from mocks. Prior to 3.5 this was generally done via the [IEventRaiser interface](http://ayende.com/wiki/Rhino+Mocks+IEventRaiser.ashx) like this:

[Test]

public void RaisingEventOnView()

{

var mocks = new MockRepository();

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);

}

Rhino Mocks 3.5 provides some extension methods to make this a bit easier:

[Test]

public void RaisingEventOnViewUsingExtensionMethod()

{

var mocks = new MockRepository();

IView view = mocks.DynamicMock<IView>();

Presenter p = new Presenter(view);  
  
 view.Raise(x => x.Load += null, this, EventArgs.Empty);  
  
 Assert.IsTrue(p.OnLoadCalled);

}

The Raise() extension method takes a subscription to the event you want raised (view.Load in this case, as per the x => x.Load += null subscription), the event sender, and the arguments for the event.  
  
[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=23)

**Property Setters Explicit Expectation API**

Two constructs for expecting settings of properties have been added. It has been possible with Rhino.Mocks before, however there has been no expressive construct in the fluent interface syntax.  
  
A property setting expectation with a certain value is specified like this:  
  
Expect.Call(mockedCustomer.Name).SetPropertyWithArguments("Ayende");  
  
For testing the interaction only, a property setting expectation without a specified value is done this way:  
  
Expect.Call(mockedCustomer.Name).SetPropertyAndIgnoreArguments();  
  
Those new constructs are equal to:  
  
using(mocks.Record()) { mockedCustomer.Name = "Ayende"; // setup an expectation for setting a specific argument.  
  
LastCall.IgnoreArguments(); //We can ignore arguments to this expectation, with gives us expectation of setting the property,regardless of the actual value. }  
  
[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=24)

**Accessing the method arguments directly**

For some advance scenarios, it is easier to handle the call to the method directly, instead of setting constraints or asserting on particular values. Rhino Mocks supports this feature using the WhenCalled method. Here is an example:

var wasCalled = false;

var stub = MockRepository.GenerateStub();

stub.Stub(x => x.StringArgString(Arg.Is("")))

.Return("blah")

.WhenCalled(delegate { wasCalled = true; });

Assert.AreEqual("blah", stub.StringArgString(""));

Assert.IsTrue(wasCalled);

This example only shows getting notified on the method call, but the feature is much more powerful. Take a look at the next example:

[Test](http://ayende.com/wiki/Test.ashx)

public void Can\_modify\_return\_value()

{

var stub = MockRepository.GenerateStub();

stub.Stub(x => x.StringArgString(Arg.Is("")))

.Return("blah")

.WhenCalled(invocation => invocation.ReturnValue = "arg");

Assert.AreEqual("arg", stub.StringArgString(""));

}  
  
[Test](http://ayende.com/wiki/Test.ashx)

public void Can\_inspect\_method\_arguments()

{

var stub = MockRepository.GenerateStub();

stub.Stub(x => x.StringArgString(null))

.IgnoreArguments()

.Return("blah")

.WhenCalled(invocation => Assert.AreEqual("foo", invocation.Arguments[0](http://ayende.com/wiki/0.ashx)));

Assert.AreEqual("blah", stub.StringArgString("

}

As you can see, this is a powerful feature, so use with caution.  
  
[Edit](http://ayende.com/wiki/Edit.aspx?Page=Rhino+Mocks+3.5&Section=25)

**Recursive mocking**

Information about recursive mocking support can be found [here](http://ayende.com/wiki/RecursiveMocksGuidance.ashx).