<http://www.codeproject.com/Articles/686935/Unit-testing-with-Mock-objects-Rhino-Mocks>

Unit testing with Mock objects (Rhino Mocks)

[Livio Francescucci](http://www.codeproject.com/script/Membership/View.aspx?mid=5533425), 25 Nov 2013 [CPOL](http://www.codeproject.com/info/cpol10.aspx)

|  |  |
| --- | --- |
| http://codeproject.cachefly.net/script/Ratings/Images/stars-fill-md.png  http://codeproject.cachefly.net/script/Ratings/Images/stars-empty-md.png | 4.91 (20 votes) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 vote, 5.3%  1 | 2 | 3 | 3 votes, 15.8%  4 | 16 votes, 84.2%  5 |

4.91/5 - 20 votes

1 removed

μ 4.79, σa 1.27 [[?](http://www.codeproject.com/KB/FAQs/RatingReputationFAQ.aspx#noisefilter)]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| loading... | |  |  |  | | --- | --- | --- | | Rate: | [vote 1vote 2vote 3vote 4vote 5](http://www.codeproject.com/Articles/686935/Unit-testing-with-Mock-objects-Rhino-Mocks) |  | |

Add a reason or comment to your vote: [**x**](http://www.codeproject.com/Articles/686935/Unit-testing-with-Mock-objects-Rhino-Mocks)   
Adding a comment to your rating is optional

A basic introduction about how to write unit tests using mock objects (Rhino Mocks).

* [**Download MockTutorial.zip - 131.7 KB**](http://www.codeproject.com/KB/testing/686935/MockTutorial.zip)

Introduction

This article is an easy introduction about how to use mock objects in unit tests and what are the main building blocks for an effective test that uses Mocks.

In this article I'm also explaining the main differences between Mock Objects and Stub objects, often confused when discussing about tests.

There are many frameworks that can be used to create mock objects when tests are written: some are commercials, some are free. The one I'm using in this tutorial is Rhino Mocks, an open source framework relatively old, but still absolutely effective for most of the unit tests you can write nowadays with C# and .NET.

Background

 To go through this article you have to be familiar with unit testing and you need to know the basic concepts about mocking. However if you want to know more (and more deeply) about the Mock World behind this short introduction, I would strongly recommend to read the outstanding [article about Mocks & Stubs from Martin Fowler](http://martinfowler.com/articles/mocksArentStubs.html) and the [Rhino Mocks documentation](http://ayende.com/wiki/Rhino+Mocks+3.5.ashx).

Using the code

The code used as an example should be very easy to understand. I wrote a small application to use as a didactical tool. I hope this will work in this article.  
  
The solution attached contains two projects: one is LiviosPizzeria and one is LiviosPizzeria.Test. The first is the project under test and the second contains the unit tests related to it.  
In the main project LiviosPizzeria you can find the main class, which we want to test. The class is named PizzaMaker and offers a single method MakePizza() which will be tested.

This is basically a method that pretends to prepare a Pizza, getting ingredients from somewhere and cooking it. The cooked Pizza will be then returned....yum! Delicious!

Below you can see the body of this method:

Hide   Copy Code

public Pizza MakePizza()

{

Pizza cookedPizza = null;

ReadyForMakePizza = false;

var ingredients = \_ingredientsProvider.GetIngredients();

var rawPizza = \_rawPizzaBuilder.CreatePizza(ingredients);

\_oven.Temperature = 300;

cookedPizza = \_oven.CookPizza(rawPizza);

\_oven.Temperature = 150;

ReadyForMakePizza = true;

return cookedPizza;

}

In the rest of the article we'll go step by step through the creation of some unit test for this method, using Rhino Mocks and trying to apply all the best practices suggested when working with mock objects.

Unit test #1: check the state of PizzaMaker

Main point: with this test, we are testing the **state** of the PizzaMaker class. We check that the state of this class is what we expect at the end of some operations.

In our example class, PizzaMaker expose a property, called ReadyForAnotherPizza.  
Every time we cook a pizza using our PizzaMaker, the status of this property is set to **false**. When the pizza is ready, the property is set back to **true**.

We are going to test that when we call the method MakePizza, at the end of this call the property ReadyForAnotherPizza is set to true, as expected.

This is the test:

Hide   Copy Code

[TestMethod]

public void WhenMakePizzaEndsThenPizzaMakerIsReadyForAnotherPizza()

{

// ARRANGE

var stubIngredientsProvider =

MockRepository.GenerateStub<iingredientsprovider>();

var stubRawPizzaBUilder = MockRepository.GenerateStub<irawpizzabuilder>();

var stubOven = MockRepository.GenerateStub<ioven>();

// ACT

var sut = new PizzaMaker(stubIngredientsProvider, stubRawPizzaBUilder, stubOven);

var pizza = sut.MakePizza();

// ASSERT

Assert.IsTrue(sut.ReadyForMakePizza);

}

**Our first mock object: the GenerateStub method**

This first test shows the most important call in any mocking framework: the creation of a basic mock object.   
In Rhino Mocks this call is GenerateStub and we use it to generate all the objects that are interacting with our system under test (the sut variable, in our code).

The approach to this test is the basic approach for any test that uses mock objects:

* We instantiate a class that we want to test
* This class has various dependencies that could be injected (for instance) from the constructor.
* We use the Mock Framework to emulate these dependencies and test against the interaction of our class with them.

Now, if you look at the constructor of our PizzaMaker object, you can see that it has some dependencies from other objects:

Hide   Copy Code

public PizzaMaker(IIngredientsProvider ingredientsProvider,

IRawPizzaBuilder rawPizzaBuilder, IOven oven)

{

\_ingredientsProvider = ingredientsProvider;

\_rawPizzaBuilder = rawPizzaBuilder;

\_oven = oven;

ReadyForMakePizza = true;

}

We mock these dependencies using the GenerateStub call and we pass the fake objects we just created, into the PizzaMaker. These objects we created can be "driven" in many different way but for now, in this test, they are just needed to allow our code to compile. At this time, all these mock objects, when invoked will return **null. However** this is not important for the specific test we are writing, so there is **no** need to overload our test with more configurations than needed.

The true test, in fact, is all around checking the state of our class. In this test we don't care (and we don't want to care) about the interaction with other objects. These problems are not part of our test that must to be kept simple and focused **on a single assertion**.

**Use a single assertion for each test as much as you can.**

This is one of the good rules in Unit Testing. Sometime is easy to write quickly four or five assertions against our system under test, to check if the properties A, B, C and (why not) also D are set as we want. You should avoid to do it because this decrease the maintainability of your test, as a test with many assertion is often not self explaining about what was tested in it and it will fail for many different reasons. When you'll be back to maintain your test, in two years or so, it will be more difficult to understand why the test is failing, especially when more assertion are "implicitly" related each other.

Unit test #2: check the interaction of PizzaMaker with other classes

Main point: with this test we are now testing a **behaviour** that we expect from this class. We check that the class is interacting in a specific way with the other objects.

In our PizzaMaker is very important that we cook the pizza before serving it to the customers. How can I check that I'm properly interacting with the Oven and passing it the pizza? This is exactly the scope of the method AssertWasCalled.

This feature, always present in mocking frameworks (with different names, of course), is used to verify that we interact properly with the others classes within our class.

Let's have a look at the code

Hide   Copy Code

[TestMethod]

public void WeNeverForgetToCookPizzaInTheOven()

{

// ARRANGE

var stubIngredientsProvider =

MockRepository.GenerateStub<iingredientsprovider>();

var stubRawPizzaBUilder = MockRepository.GenerateStub<irawpizzabuilder>();

var mockOven = MockRepository.GenerateStub<ioven>();

mockOven.Stub(oven => oven.CookPizza(Arg<irawpizza>.Is.Anything));

// ACT

var sut = new PizzaMaker(stubIngredientsProvider, stubRawPizzaBUilder, mockOven);

var pizza = sut.MakePizza();

// ASSERT

mockOven.AssertWasCalled(oven => oven.CookPizza(Arg<irawpizza>.Is.Anything));

}

**Stub( X => ...) and AssertWasCalled. What is this?**

There are now two new "friends" in our test.

The first is:  mockOven.Stub(oven => oven.CookPizza(Arg<IRawPizza>.Is.Anything));. With this line, we start to "teach" our mock objects how to react when they are called. In this specific line we are saying to our mock Oven: "hey, be aware that someone is going to use you! It will be someone that will call your method "CookPizza" and it can call you with any argument. You just don't care about it".   
  
Why we do this? Because now that our mock Oven is aware of this call, we can **ask him** is this call really happened! And then we can test that every time we prepare a pizza, something is really cooked in this Oven. In this scenario we say that we are testing a **behaviour**.

That's exactly what we verify with  the last line, where is the second friend:

是在MakePizza（）这个函数执行时会调用Oven类下的CookPizza（）任何参数形式中的某一种参数形式的重载，如果new出来的对象是oven，则就是调用oven.CookPizza(“任何参数形式”)。ACT前面那句是你给我盯着点，看看过会ACT时调用了任何一种参数形式的CookPizza（）函数没？然后ACT，ACT之后是AssertWasCalled，问你到底调用了没？

Hide   Copy Code

mockOven.AssertWasCalled(oven => oven.CookPizza(Arg<IRawPizza>.Is.Anything))

If the call to the Oven doesn't happen in the PizzaMaker (maybe a developer, during a refactoring, just deleted it without noticing it), then the test will fail. The mock object will complain with us saying "hey! Wait a minute! You told me that someone should have called me but nothing happened!".

Once again, I'm not exploring in detail the syntax of this code, because it changes for each mocking framework, but I hope that the concept behind this test is in someway clear.

Unit test #3: how to check if we properly set another object

In our Pizzeria, we've got this requirement: every time we start cooking our pizza, is important that the Oven is set to a temperature of 300C degrees. At the same time, when the cooking is done, we need to cool down the temperature of the Oven to 150C. This can be done setting the property Temperature in the Oven. How to test that PizzaMaker is correctly doing this?

Again, this is a test where we are going to verify a behaviour. However in this scenario, when working with Rhino Mocks, you need to use a new method, a different approach: the     MockRepository.GenerateMock(...).

The reason for this is that, while in the previous test we verified if we were invoking a method (and AssertWasCalled is perfect for this), now we want to check that a property has been set as expected.

To **check** if a property has been set in a stub object, **we can't use a Stub**. With Stubs we can't verify property settings. This is the time, with Rhino Mocks, where we are forced to use the GenerateMock call.

Hide   Copy Code

[TestMethod]

public void PizzaMakerSetOvenToTheProperTemperature()

{

// ARRANGE

IIngredientsProvider stubIngredientsProvider = MockRepository.GenerateStub<iingredientsprovider>();

IRawPizzaBuilder stubRawPizzaBUilder = MockRepository.GenerateStub<irawpizzabuilder>();

// this won't work. We can't verify this call.

// If you comment out the property settings in the method under test, this test won't fail

//IOven mockOven = MockRepository.GenerateStub<ioven>();

//mockOven.Temperature = 300;

//mockOven.Temperature = 150;

//this is the proper way to check settings on property

IOven mockOven = MockRepository.GenerateMock<ioven>();

mockOven.Expect(oven => oven.Temperature = 300);

mockOven.Expect(oven => oven.Temperature = 150);

// ACT

var sut = new PizzaMaker(stubIngredientsProvider, stubRawPizzaBUilder, mockOven);

var pizza = sut.MakePizza();

// ASSERT

mockOven.VerifyAllExpectations();

}

Please, note the lines I commented in the test above, marked in the block "this won't work":

Hide   Copy Code

// this won't work. We can't verify this call. If you comment out

// the property settings in the method under test, this test won't fail

//IOven mockOven = MockRepository.GenerateStub();

//mockOven.Temperature = 300;

//mockOven.Temperature = 150;

 I left this part in the code, close to the good one, to show you the differences side by side. With the "wrong" syntax you can set a value which will be returned from the property "Temperature" in the "IOven" object, but this is completely different than check if someone else set this property with an expected value.

意思是告诉他，盯着点啊，在过会调用函数时，我预计Temperature会先被设成300，后被设成150。然后ACT调用MakePizza（）。最后再Assert看是否符合所有预期。

第一个例子是利用stub对象间接new出来对象，再验证这个被new出来对象的属性，不是直接stub对象的属性。

第三个例子是验证stub/mock对象的属性，所以只能是mock，不能是stub。

In the first case we are **telling** to the Oven "please, every time someone asks you the value of Temperature, return the value 300C". In the second case we are **asking** to the mock Oven "please, verify if someone set the property Temperature in the Oven at 300C".

Unit test #4: a wrong usage of StrictMock, and why it is Discouraged

There is a special kind of Mock objects, which are called "StrictMock", that could be very useful...but only in very few scenarios.

The main aspect of this kind of objects is that the VerifyAllExpectation on them will fail whenever there is a call made on the mock object which has not been explicitly set before.  
  
In other words, when we declare and use a StrictMock object, writing the expectations on it in our unit test, we are implicitly stating that **every call on this StrictMock object which has not been set is an error** and the "VerifyExpectation" will fail if some unexpected call happens.

Let's see if I can clarify this with an example. Here is a unit test that uses a StrictMock, the #4 in our project.

Hide   Copy Code

[TestMethod]

public void WhyTheStrictMockIsGenerallyBad()

{

//ARRANGE

var stubIngredientsProvider = MockRepository.GenerateStub<iingredientsprovider>();

var stubRawPizzaBuilder = MockRepository.GenerateStub<irawpizzabuilder>();

var mockOven = MockRepository.GenerateStrictMock<ioven>(); //DISCOURAGED

mockOven.Expect(oven => oven.Temperature = 300);

mockOven.Expect(oven => oven.CookPizza(Arg<irawpizza>.Is.Anything));

mockOven.Expect(oven => oven.Temperature = 150);

// ACT

var sut = new PizzaMaker(stubIngredientsProvider, stubRawPizzaBuilder, mockOven);

sut.MakePizza();

// ASSERT

mockOven.VerifyAllExpectations();

}

This test want to check that we are properly setting the temperature in the Oven before using it, and then we are calling the method CookPizza.  I think that tests like this are less maintainable, because we put too many assertions in the same test (verify temperature and verify CookPizza) and even related to different behaviors. However, we could also tolerate it for now, as when we are in a rush we all do terrible things in our codebase...

The important thing to notice, using the StrictMock in this test, is that, from now, we subscribed a kind of mortgage with this test and we'll be very close to it for ever! From now on any "change" we'll make in PizzaMaker regarding the interaction with the Oven (also just adding a new call) will result in this test failing also if our "behaviour under test" hasn't been changed.  
  
Again, an example will help: what happen if at some point a new version of my Oven offers a new method called "AutoClean()" that can be called at the end of the cooking? We will add the new call in our PizzaMaker to implement this exciting feature! Wow, an Oven that clean itself, that's cool! Let's add it immediately:

Hide   Copy Code

public Pizza MakePizza()

{

Pizza cookedPizza = null;

ReadyForMakePizza = false;

var ingredients = \_ingredientsProvider.GetIngredients();

var rawPizza = \_rawPizzaBuilder.CreatePizza(ingredients);

\_oven.Temperature = 300;

cookedPizza = \_oven.CookPizza(rawPizza);

// A NEW FEATURE JUST INTRODUCED THAT WILL BREAK ANY UNIT TEST MADE WITH STRICT MOCK!

\_oven.AutoClean();

ReadyForMakePizza = true;

return cookedPizza;

}

And viola! New feature is in. We run our unit test suite...and what happens? We know what will happen: our test #4 will fail, because we didn't include this call in the "expected" calls for the IOven object, but actually nothing is wrong in our test! The behaviour under test (check if we set the temperature, check if we cook the pizza) has been changed?

Now try to imagine if you extensively use this StrictMock in your whole codebase. Every time you add something in your code, something will be broken. Always. With the usage of StrictMocks in these scenarios we are only making our test suite less maintainable!

This is the reason why the usage of this StrictMocks has always to be considered very carefully.

History

* 2013-11-24: First version.
* 2013-11-25: Fixed some formatting issues.