NMock<3>

NMock3 is a Mocking and Stubbing framework that uses expectations to define interactions between a controller and the mock. Its primary use is to **be** the implementation of a code interface.
Visit http://NMock3.codeplex.com for **Tutorials** and **Documentation**.

Creating a MockFactory. A MockFactory creates and ties together all mocks. Only one is needed per test class.	`MockFactory _factory = new MockFactory();`	

Creating a Mock<T>. A Mock<T> is used to set expectations on how the underlying type will be exercised.	`Mock<Interface> _mock =`
	`_factory.CreateMock<Interface>();`	

Creating a Stub. A Stub<T> is a Mock<T> where all expectations are defaulted to *AtLeast(0)*. (No expectations)	`_mock.Stub.Out...`	
Syntax:		

Syntax properties. Some properties in the API are only included for readability. (Affectionately called syntactic sugar.) *Expects* is a "syntax class".	`_mock`
	`    .Expects`
	`    .####`	

Specifying the number of calls. The Expects syntax class contains properties and methods to specify the number of expected calls to the member specified in the expectation	`.Any`
	`.One`
	`.AtLeast(int)`
	`.AtMost(int)`
		`.No`
		`.Exactly(int)`
		`.AtLeastOne`
		`.AtMostOne`
		`.Between(int, int)`

Expectations:		

Getting a property value. Creates an expectation that the getter of this property will be called.
GetProperty uses the lambda expression to extract the name of the property for the expectation.
WillReturn is strongly-typed for compile time checking.
	`_mock`
	`    .Expects`
	`    .One`
	`    .`**`GetProperty`**`(_ => _.SayHello)`
	`    .WillReturn("Hello, World!");`	

Setting a property value. Creates an expectation that the setter of this property will be called and this value will be set. NMock3 will use the value from the lambda expression as the expected value.	`_mock`
	`    .Expects`
	`    .One`
	`    .`**`SetPropertyTo`**`(_ => _.RowCount = 3);`	

Calling a method. Creates an expectation that this method will be called with the supplied parameters and will return the specified value. The parameters will be wrapped in *EqualMatcher*s meaning the values will be matched exactly (even object references.) See **Matchers** below.	`_mock`
	`    .Expects`
	`    .One`
	`    .`**`MethodWith`**`(_ => _.Search("query", 10))`
	`    .WillReturn(dataSet);`	

Binding events. Creates an expectation that this event will be bound to a delegate. "Add" or "Remove" is inferred by the use of "+=" or "-=" in the expression. *EventInvoker* is a class that can be used later to actually invoke the event. (*null* is **only** needed for the compiler!)	`EventInvoker saveInvoker =`
	`_mock`
	`    .Expects`
	`    .One`
	`    .`**`EventBinding`**`(_ => _.Save += null);`	

Invoking events. Use the *Invoke* method to raise an event in a unit test after all expectations have been created.	`saveInvoker.`**`Invoke`**`();`	
Verification:		

Verifying calls. NMock3 will throw an exception immediately when something *unexpected* happens. Call this method to verify that <u>all</u> expectations were met.	`[TestCleanup] public void TearDown() {`
	`_factory.`**`VerifyAllExpectationsHaveBeenMet`**`();`
	`}`	

Suppressing exceptions. Unit tests that are designed to throw exceptions should call this method to clear thrown exceptions.	`_factory.`**`ClearException`**`();`	
Advanced:		

The MockObject property. The Mock<T> class exposes a MockObject property to access the underlying type.	`Controler controler = new`
	`Controler(_mock.`**`MockObject`**`);`	

Ordering calls. NMock3 can add constraints to the expectations so that they are executed in a specific order.	`using(_factory.`**`Ordered`**`) {`
	`    _mock.Expects.One.####;`
	`    _mock.Expects.One.####;`
	`}`	

Matchers:

Matching a Type. In some situations it is not possible to match the instance of an object. To accomplish this, use a matcher instead. Note how the use of 'null' in the method call is used to match the signature and the matcher and argument are specified in the '.With' call.

```
_mock.Expects.One
.Method(_ => _.Method1(null, null))
.With(Is.TypeOf<IDbCommand>(), 5);
```

Custom Matching. To perform custom matching, create a subclass of Matcher or use the Is.Match<>() shortcut (which creates an instance of PredicateMatcher<T>) The shortcut provides a way to perform matching logic in a method or expression without deriving a class.

```
_mock.Expects.One
.Method(_ => _.Method2(null))
.With(
Is.Match<Customer>(c => c.Id != null));
//check that the customer Id is not null
```

Invoking a Callback. Some APIs like RIA Services perform Async operations and require a callback method as a parameter. In NMock3, use a CallbackMatcher<T> to match those parameters. Later on in the unit test, simulate the callback by calling the action stored in the Callback property of the CallbackMatcher<>.

```
var matcher = new CallbackMatcher<Action>();
_mock
    .Expects
    .One
    .Method(_ => _.Async(null))
    .With(matcher);
matcher.Callback(); //simulate the callback
```

Actions:

Returning a value. Use the '.WillReturn()' shorthand to specify the value to return. '.WillReturn()' is a strongly-typed shorthand to the syntax method Return.Value().

```
_mock.Expects.One
    .MethodWith(_ => _.Search("query", 10))
    .WillReturn(dataSet);
```

Returning queued values. Use a QueueAction<> to return a sequence of values when an expectation is matched multiple times.

```
var queue = new Queue<string>();
queue.Enqueue("string 1");
queue.Enqueue("string 2");
_mock.Expects.Exactly(2)
.PropertyGet(_ => _.StringProp)
.Will(Return.Queue<string>(queue));
```

Throwing an exception. Creates an expectation that an exception will be thrown when this method *or property* is accessed.

```
_mock.Expects.One
    .MethodWith(_ => _.ThrowError())
    .Will(Throw.Exception(new Exception()));
```

Performing an Action. Actions can also be used to do something when an expectation is met. In this example, SaveAsync is void and DoSomething is invoked when SaveAsync is called by using the syntax method Invoke.Action which wraps an InvokeAction class.

```
_mock
    .Expects
    .One
    .MethodWith(_ => _.SaveAsync())
    .Will(Invoke.Action(DoSomething);
private void DoSomething() {...;}
```

Expect class:

Expecting an exception. Instead of using an *ExpectedException* attribute, wrap a method call with an Expect.That(Action).Throws(Exception) call. By using this convention you are assured that the exception is thrown on the right method and not just somewhere in the unit test.

```
Expect
    .That(() => obj.DoSomething(null))
    .Throws<ArgumentNullException>("Expected
an ArgumentNullException that contains the
string 'argument'.", new
StringContainsMatcher("Parameter name:
argument"));
```

Setting expectations on non-Mock<> types. Previous versions of NMock and in other mocking frameworks, the Mock<> type is not used and expectations are applied directly to an instance of a type that is really a proxy.

```
var instance =
_factory.CreateInstance<Interface>();

Expect
    .On(instance)
    .One
    .Method(_ =>_.DoSomething());
```

Advanced Property Expectations:

Getting an internal value. In some cases the code under test will create an instance of an object inside of a method and then set a property to that value. Normally NMock would validate that the property was set through an expectation but it would disregard the value. Using the .WillReturnSetterValue() method signals NMock to retain the value for a future call.

```
mock.Expects.One.SetProperty(_ =>
_.Prop).To(Is.TypeOf<AType>());
mock.Expects.One.GetProperty(_ =>
_.Prop).WillReturnSetterValue();

mock.MockObject.DoSomething();
Assert.AreEqual(aType, mock.MockObject.Prop);
```