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<IBusinessLogic> _mock = _factory.CreateMock<IBusinessLogic>();`
Creating a Stub<T>. A Stub<T> is a Mock<T> where all expectations are defaulted to *AtLeast(0)*. (No expectations)	`Stub<IBusinessLogic> _stub = _factory.CreateStub<IBusinessLogic>();`

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 to specify the number of expected calls to the member specified in the expectation	`.One`
	`.AtLeast(int)`
	`.AtMost(int)`
		`.No`
		`.Exactly(int)`
		`.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(m => m.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(m => m.RowCount = 3);`

Calling a method. Creates an expectation that this method will be called with the supplied parameters and will return the specified value.	`_mock`
	`  .Expects`
	`  .One`
	`  .MethodWith(m => m.Search("query", 10))`
	`  .WillReturn(dataSet);`

Binding events. Creates an expectation that this event will be bound to a delegate.
MockEventInvoker is a class that can be used later to actually invoke the event.
(*null* is **only** needed for the compiler!)
	`MockEventInvoker saveInvoker =`
	`_mock`
	`  .Expects`
	`  .One`
	`  .EventAddHandler(m => m.Save += null);`

Firing events. Use the *Invoke* method to raise an event in a unit test after all expectations have been created.	`saveInvoker.Invoke(this, EventArgs.Empty);`

Verification:

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

Suppressing exceptions. Unit tests that are designed to throw exceptions should call this method to avoid cleanup.	`_factory` `.SuppressUnexpectedAndUnmetExpectations();`

Advanced:

The MockObject. 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.####;`
	`}`

Throwing an exception. Creates an expectation that an exception will be thrown when this method *or property* is accessed.	`_mock`
	`  .Expects`
	`  .One`
	`  .MethodWith(m => m.ThrowError())`
	`  .Will(Throw.Exception(new Exception()));`