Moq + xUnit + AutoFixture – Cheat sheet v1.0

Moq

Methods

```
// Assumptions:
public interface IFoo {
    bool DoSomething(string str);
    bool TryParse(string str1, out string str2); }


var mock = new Mock<IFoo>();
mock.Setup(foo => foo.DoSomething("ping"))
            .Returns(true);

// out arguments
var outString = "ack";
// TryParse will return true, and the out argument will
return "ack", lazy evaluated
mock.Setup(foo => foo.TryParse("ping", out outString))
            .Returns(true);

// ref arguments
var instance = new Bar();
// Only matches if the ref argument to the invocation
is the same instance
mock.Setup(foo => foo.Submit(ref instance))
            .Returns(true);

// access invocation arguments when returning a value
mock.Setup(x => x.DoSomething(It.IsAny<string>()))
            .Returns((string s) => s.ToLower());

 // throwing when invoked
mock.Setup(foo => foo.DoSomething("reset"))
            .Throws<InvalidOperationException>();
mock.Setup(foo => foo.DoSomething(""))
            .Throws(new ArgumentException("command"));

// lazy evaluating return value
mock.Setup(foo =>foo.GetCount()).Returns(() => count);

// returning different values on each invocation
var mock = new Mock<IFoo>();
var calls = 0;
mock.Setup(foo => foo.GetCountThing())
    .Returns(() => calls)
    .Callback(() => calls++);
// returns 0 on first invocation, 1 on the next, and
so on
Console.WriteLine(mock.Object.GetCountThing());
```

Matching Arguments

```
// any value
mock.Setup(foo => foo.DoSomething(It.IsAny<string>()))
            .Returns(true);

// matching Func<int>, lazy evaluated
mock.Setup(foo =>foo.Add(It.Is<int>(i => i % 2 == 0)))
            .Returns(true);

// matching ranges
mock.Setup(foo => foo.Add(It.IsInRange<int>(0, 10,
            Range.Inclusive))).Returns(true);

// matching regex
mock.Setup(x => x.DoSomething(It.IsRegex("[a-d]+",
            RegexOptions.IgnoreCase))).Returns("foo");
```

Properties

```
mock.Setup(foo => foo.Name).Returns("bar");

// auto-mocking hierarchies (a.k.a. recursive
mocks)
mock.Setup(foo =>foo.Bar.Baz.Name).Returns("baz");

// expects an invocation to set the value to "foo"
mock.SetupSet(foo => foo.Name = "foo");

// or verify the setter directly
mock.VerifySet(foo => foo.Name = "foo");
```

Setup a property so that it will automatically start tracking its value (also known as Stub):

```
// start "tracking" sets/gets to this property
mock.SetupProperty(f => f.Name);

// alternatively, provide a default value for the
stubbed property
mock.SetupProperty(f => f.Name, "foo");

// Now you can do:
IFoo foo = mock.Object;
// Initial value was stored
Assert.Equal("foo", foo.Name);

// New value set which changes the initial value
foo.Name = "bar";
Assert.Equal("bar", foo.Name);
```

Stub all properties on a mock (not available on Silverlight):

```
mock.SetupAllProperties();
```

Events

```
// Raising an event on the mock
mock.Raise(m => m.FooEvent += null,
             new FooEventArgs(fooValue));

// Raising an event on a descendant down the
hierarchy
mock.Raise(m => m.Child.First.FooEvent += null,
             new FooEventArgs(fooValue));

// Causing an event to raise automatically when
Submit is invoked
mock.Setup(foo => foo.Submit())
      .Raises(f => f.Sent += null, EventArgs.Empty);
// The raised event would trigger behaviour on the
object under test, which you would make assertions
about later (how its state changed as a consequence,
typically)

// Raising a custom event which does not adhere to
the EventHandler pattern
public delegate void MyEvHandler(int i, bool b);
public interface IFoo
{
    event MyEvHandler MyEvent;
}

var mock = new Mock<IFoo>();
...
// Raise passing the custom arguments expected by
the event delegate
mock.Raise(foo => foo.MyEvent += null, 25, true);
```

Callbacks

```
var mock = new Mock<IFoo>();
mock.Setup(foo => foo.Execute("ping"))
      .Returns(true)
      .Callback(() => calls++);

// access invocation arguments
mock.Setup(foo => foo.Execute(It.IsAny<string>()))
    .Returns(true)
    .Callback((string s) => calls.Add(s));

// alternate equivalent generic method syntax
mock.Setup(foo => foo.Execute(It.IsAny<string>()))
    .Returns(true)
    .Callback<string>(s => calls.Add(s));

// access arguments for methods with multiple
parameters
mock.Setup(foo =>
 foo.Execute(It.IsAny<int>(), It.IsAny<string>()))
    .Returns(true)
    .Callback<int, string>((i, s)=>calls.Add(s));

// callbacks can be specified before and after
invocation
mock.Setup(foo => foo.Execute("ping"))
 .Callback(()=>Console.WriteLine("Before return"))
 .Returns(true)
 .Callback(()=>Console.WriteLine("After return"));
```

Verification

```
// Verify with custom error message for failure
mock.Verify(foo=>foo.Execute("ping"),"When doing
operation X, the service should be pinged
always");

// Method should never be called
mock.Verify(foo => foo.Execute("ping"),
            Times.Never());

// Called at least once
mock.Verify(foo => foo.Execute("ping"),
            Times.AtLeastOnce());

mock.VerifyGet(foo => foo.Name);

// Verify setter invocation, regardless of value.
mock.VerifySet(foo => foo.Name);

// Verify setter called with specific value
mock.VerifySet(foo => foo.Name ="foo");

// Verify setter with an argument matcher
mock.VerifySet(foo => foo.Value =
            It.IsInRange(1, 5, Range.Inclusive));
```

Customizing Mock Behavior

1. Make an automatic recursive mock: a mock that will return a new mock for every member that doesn't have an expectation and whose return value can be mocked (i.e. it is not a value type)

```
var mock = new Mock<IFoo>
            { DefaultValue = DefaultValue.Mock };
// default is DefaultValue.Empty

// this property access would return a new mock of
IBar as it's "mock-able"
IBar value = mock.Object.Bar;

// the returned mock is reused, so further accesses
to the property return the same mock instance. This
allows us to also use this instance to set further
expectations on it if we want
var barMock = Mock.Get(value);
barMock.Setup(b => b.Submit()).Returns(true);
```

2. Invoke base class implementation if no expectation overrides the member (a.k.a. "Partial Mocks" in Rhino Mocks): default is false. (*this is required if you are mocking Web/Html controls in System.Web!*)

```
var mock = new Mock { CallBase = true };
```

3. Make mock behave like a "true Mock", raising exceptions for anything that doesn't have a corresponding expectation: in Moq slang a "Strict" mock; default behavior is "Loose" mock, which never throws and returns default values or empty arrays, enumerables, etc. if no expectation is set for a member

```
var mock = new Mock(MockBehavior.Strict);
```

4. Centralizing mock instance creation and management: you can create and verify all mocks in a single place by using a MockFactory, which allows setting the MockBehavior, its CallBase and DefaultValue consistently

```
var factory
    = new MockFactory(MockBehavior.Strict)
          { DefaultValue = DefaultValue.Mock };

// Create a mock using the factory settings
var fooMock = factory.Create<IFoo>();

// Create a mock overriding the factory settings
var barMock =
        factory.Create<IBar>(MockBehavior.Loose);

// Verify all verifiable expectations on all mocks
created through the factory
factory.Verify();
```

Miscellaneous

Setting expectations for protected members (you can't get intellisense for these, so you access them using the member name as a string):

```
// at the top of the test fixture
using Moq.Protected;

// in the test
var mock = new Mock<CommandBase>();
    mock.Protected()
     .Setup<int>("Execute")
     .Returns(5);

// if you need argument matching, you MUST use
ItExpr rather than It
mock.Protected()
    .Setup<string>("Execute",
           ItExpr.IsAny<string>())
    .Returns(true);
```

Advanced Features

```
// get mock from a mocked instance
IFoo foo = // get mock instance somehow
var fooMock = Mock.Get(foo);
fooMock.Setup(f => f.Submit()).Returns(true);

// implementing multiple interfaces in mock
var foo = new Mock<IFoo>();
var disposableFoo = foo.As<IDisposable>();
// now the IFoo mock also implements IDisposable
disposableFoo.Setup(df => df.Dispose());
```

```
//implementing multiple interfaces in single mock
var foo = new Mock<IFoo>();
foo.Setup(f => f.Bar()).Returns("Hello World");
foo.As<IDisposable>().Setup(df => df.Dispose());

// custom matchers
mock.Setup(foo => foo.Submit(IsLarge()))
            .Throws<ArgumentException>();
...
public string IsLarge() {
  return Match.Create<string>(s =>
    !String.IsNullOrEmpty(s) && s.Length > 100);}
```

Mocking internal types of another project: add the following assembly attribute (typically to the AssemblyInfo.cs) to the project containing the internal types:

```
// This assembly is the default dynamic assembly generated Castle DynamicProxy, used by Moq. Paste in a single line.
[assembly:
InternalsVisibleTo("DynamicProxyGenAssembly2,PublicKey
=0024000004800000940000000602000000240000525341310004000001000100c547cac37abd99c8db225ef2f6c8a3602f3b3606cc9891605d02baa56104f4cfc0734aa39b93bf7852f7d9266654753cc297e7d2edfe0bac1cdcf9f717241550e0a7b191195b7667bb4f64bcb8e2121380fd1d9d46ad2d92d2d15605093924cceaf74c4861eff62abf69b9291ed0a340e113be11e6a7d3113e92484cf7045cc7")]
```

Note: when you need to pass the mock for consumption, you must use the `mock.Object` accessor as a consequence of a C# compiler restriction (vote to get it removed at Microsoft Connect)

Linq to Mocks

Moq is the one and only mocking framework that allows specifying mock behavior via declarative specification queries. You can think of Linq to Mocks as:

from the universe of mocks, get me one/those that behave like this (by Fernando Simonazzi)

Keep that query form in mind when reading the specifications:

```
var services = Mock.Of<IServiceProvider>(
  sp => sp.GetService(typeof(IRepository)) ==
    Mock.Of<IRepository>
      (r => r.IsAuthenticated == true)
  && sp.GetService(typeof(IAuthentication<>)) ==
    Mock.Of<IAuthentication>
      (a => a.AuthenticationType == "OAuth"));

// Multiple setups on a single mock and its recursive mocks
ControllerContext context = Mock.Of<ControllerContext>
  (ctx =>
    ctx.HttpContext.User.Identity.Name == "kzu" &&
    ctx.HttpContext.Request.IsAuthenticated == true &&
    ctx.HttpContext.Request.Url ==
            new Uri("http://moqthis.com") &&
    ctx.HttpContext.Response.ContentType ==
            "application/xml");

// Setting up multiple chained mocks:
var context = Mock.Of<ControllerContext>
  (ctx =>
    ctx.HttpContext.Request.Url ==
      new Uri("http://moqthis.me") &&
    ctx.HttpContext.Response.ContentType ==
      "application/xml" &&
     // Chained mock specification
    ctx.HttpContext.GetSection("server") ==
      Mock.Of<ServerSection>(config =>
        config.Server.ServerUrl ==
          new Uri("http://moqthis.com/api"));
```

Linq to Mocks is great for quickly stubbing out dependencies that typically don't need further verification. If you do need to verify later some invocation on those mocks, you can easily retrieve them with `Mock.Get(instance)`.

xUnit

Asserts

```
// Simple
Assert.Equal(1, 2); //Fail
Assert.NotEqual("expected", "actual"); // Pass

// Ranges
Assert.InRange(actual, 1, 20); // Pass
Assert.NotInRange(actual, 20, 100); // Fail

// Booleans
Assert.False(true);     Assert.True(true);

// References
Assert.Same(expected, actual);
Assert.NotSame(expected, actual);

// Nulls
Assert.Null(null);      Assert.NotNull(actual);

// IEnumerable
Assert.Empty(list);     Assert.NotEmpty(list);
Assert.Contains(10, list);
Assert.DoesNotContain(10, list);

//Types
Assert.IsType<int>(10); // Pass
Assert.IsNotType<string>("test"); // Fail

// Exception
Assert.Throws<NullReferenceException>(Method());
```

Attributes

```
[Fact] // Test method
[Fact(Skip = "Reason...")] //Skip test

[Theory]
[InlineData(1, true, "test")] // Provide inline
                                 data for test
[MemberData("GetData", MemberType =
  typeof(DataProvider))] // Generate data for test
[Collection("Name")] // Group tests in collection
```

AutoFixture

Short manual

This page contains short code snippets that demonstrate AutoFixture features. All examples assume that a Fixture instance called *fixture* has previously been created like this:

```
var fixture = new Fixture();

// Completely Autogenerated String
var autoGeneratedText = fixture.Create<string>();
// string: "f5cdf6b1-a473-410f-95f3-f427f7abb0c7"

// Seeded String
var generatedTextWithPrefix
                         = fixture.Create("Name");
// string: "Name30a35da1-d681-441b-9db3-77ff51…"

// Autogenerated Number
int autoGeneratedNumber = fixture.Create<int>();
// int: 27, followed by 9, then by 171, etc.

// Complex Type
var autoGeneratedClass =
                fixture.Create<ComplexParent>();

// Abstract Types
fixture.Register<IMyInterface>(
                     () => new FakeMyInterface());
// Every time the fixture instance is asked to create an instance of IMyInterface, it will return a new instance of FakeMyInterface.

// Replaced Default Algorithm
fixture.Register<string>(() => "ploeh");
string result = fixture.Create<string>();
//Result: string: "ploeh"

// Sequence of Strings
var strings = fixture.CreateMany<string>();
// IEnumerable:
// - string: "ecc1cc75-cd7a-417f-…"
// - string: "fce70a7b-fae5-474f-…"
// - string: "79b45532-d66f-4abc-…"

// Sequence of Custom Objects
var myInstances = fixture.CreateMany<MyClass>();

// Add to Collection
var list = new List<MyClass>();
fixture.AddManyTo(list);

// Set Property
var mc = fixture.Build<MyClass>()
    .With(x => x.MyText, "Ploeh")
    .Create();
// MyClass
// - MyText: string: "Ploeh"

// Disable AutoProperties
var sut = fixture.Build<Vehicle>()
    .OmitAutoProperties()
    .Create();
// The Wheels property will have the default value of 4, instead of having an auto generated value assigned via its setter

// Disable Property
var person = fixture.Build<Person>()
    .Without(p => p.Spouse)
    .Create();
// Person:
// - BirthDay: DateTime: { 18.08.2009 07:37:06}
// - Name: String: "Name949c7c83-c77b-434f-…"
// - Spouse: Person: null

// Perform Action
var mc = fixture.Create<MyClass>();
var mvm = fixture.Build<MyViewModel>()
    .Do(x => x.AvailableItems.Add(mc))
    .With(x => x.SelectedItem, mc)
    .Create();
// MyViewModel:
// - AvailableItems: ICollection
//   -MyClass(mc)
// - SelectedItem: MyClass(mc)

// Customize Type
var mc = fixture.Create<MyClass>();
fixture.Customize<MyViewModel>(ob => ob
    .Do(x => x.AvailableItems.Add(mc))
    .With(x => x.SelectedItem, mc));
var mvm = fixture.Create<MyViewModel>();
// MyViewModel:
// - AvailableItems: ICollection
//   - MyClass(mc)
// - SelectedItem: MyClass(mc)

// AutoData Theories
// Add a reference to Ploeh.AutoFixture.Xunit.
[Theory, AutoData]
public void Test(int primitiveValue, string text) {}
// primitiveValue: int: 1
// text: string: "textf70b67ff-05d3-4498-…"

// Inline AutoData Theories
// Add a reference to Ploeh.AutoFixture.Xunit.
[Theory]
[InlineAutoData("foo")]
[InlineAutoData("foo", "bar")]
public void Test(string text1, string text2,
MyClass myClass)
{
}
// Uses the InlineData values for the the first method arguments, and then uses AutoData for the rest(when the InlineData values run out).
// First test run:
// text1: string: "foo"
// text2: string: "text2c1528179-fd1b-4f5a-…"
// myClass: an autogenerated variable of MyClass
// Second test run:
// text1: string: "foo"
// text2: string: "bar"
// myClass: an autogenerated variable of MyClass

// Auto-Mocking with Moq
// Add a reference to Ploeh.AutoFixture.AutoMoq.
fixture.Customize(new AutoMoqCustomization());
var result = fixture.Create<IInterface>();
// A mocked instance of a type assignable from IInterface

// Auto-configured Mocks
// When AutoConfiguredMoqCustomization is added to an IFixture instance, not only will it behave as an Auto-Mocking Container, but it will also automatically configure all the generated Test Doubles(Mocks) so that their members return values generated by AutoFixture.
fixture.Customize(
      new AutoConfiguredMoqCustomization());
fixture.Inject<int>(1234);

var document = fixture.Create<IDocument>();
Console.WriteLine(document.Id); //1234
// This customization will automatically configure any virtual methods/indexers/out parameters and stub all properties.Additionally, class mocks will have their fields set.
```

More Information

- Due to a limitation in Moq, AutoConfiguredMoqCustomization is not able to setup methods with ref parameters.
- AutoConfiguredMoqCustomization does not configure generic methods either. You can, however, easily set these up using the ReturnsUsingFixture extension method:

```
converter.Setup(x => x.Convert<double>("10.0"))
         .ReturnsUsingFixture(fixture);
```

Source

- Moq:	https://goo.gl/o63BP0
- AutoFixture:	https://goo.gl/AfqXHt
- Cheat sheet:	http://goo.gl/5nJmJ8