Быстрый старт в Moq.

leave a comment »

Moq разработан чтобы быть простым, строго типизированным, небольшим, но в тоже время реализовывать большую функциональность.

Методы

```
    var mock = new Mock<IFoo>();
    `mock.Setup(foo ⇒ foo.DoSomething(``"ping"``)).Returns(``true``);`
    `// out агрументы`
    `var` `outString =` `"ack"``;`
    `// TryParse возвратит true, а out аргумент - "ack"`
    `mock.Setup(foo ⇒ foo.TryParse(``"ping"``,` `out`
`outString)).Returns(``true``);`
    `// ref аргумент`
    `var` `instance =` `new` `Bar();`
    `// Совпадает толькое если ref аргумент того же экземпляра`
    `mock.Setup(foo ⇒ foo.Submit(``ref` `instance)).Returns(``true``);`
    `// доступные аргументы для вызова когда возвращается значение`
    `mock.Setup(x ⇒ x.DoSomething(It.IsAny<``string``>()))`
    `.Returns((``string` `s) ⇒ s.ToLower());`
    `// выбрасывание исключения при вызове`
    `mock.Setup(foo ⇒
foo.DoSomething(``"reset"``)).Throws<InvalidOperationException>();`
    `mock.Setup(foo ⇒ foo.DoSomething(``""``)).Throws(``new`
`ArgumentException(``"command"``));`
    `// ленивое вычисление возвращаемого значения`
    `mock.Setup(foo ⇒ foo.GetCount()).Returns(() ⇒ count);`
    `// возвращение различных значений при каждом вызове`
    `var` `mock =` `new` `Mock<IFoo>();`
    `var` `calls = 0;`
    `mock.Setup(foo ⇒ foo.GetCountThing())`
     `.Returns(() ⇒ calls)`
     `.Callback(() ⇒ calls++);`
    `// возвращает 0 при первом вызове, 1 при втором и т.д.`
    `Console.WriteLine(mock.Object.GetCountThing());`
```

Методы

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

// out аргументы
var outString = "ack";
// TryParse возвратит true, а out аргумент - "ack"
mock.Setup(foo => foo.TryParse("ping", out outString)).Returns(true);

// ref аргумент
var instance = new Bar();
// Совпадает толькое если ref аргумент того же экземпляра
mock.Setup(foo => foo.Submit(ref instance)).Returns(true);

// доступные аргументы для вызова когда возвращается значение
mock.Setup(x => x.DoSomething(It.IsAny<string>()))
    .Returns((string s) => s.ToLower());

// выбрасывание исключения при вызове
mock.Setup(foo => foo.DoSomething("reset")).Throws<InvalidOperationException>();
mock.Setup(foo => foo.DoSomething("")).Throws(new ArgumentException("command"));

// ленивое вычисление возвращаемого значения
mock.Setup(foo => foo.GetCount()).Returns(() => count);

// возвращение различных значений при каждом вызове
var mock = new Mock<IFoo>();
var calls = 0;
mock.Setup(foo => foo.GetCountThing())
    .Returns(() => calls)
    .Callback(() => calls++);
// возвращает 0 при первом вызове, 1 при втором и т.д.
Console.WriteLine(mock.Object.GetCountThing());
```

Совпадающие аргументы

```
        `// любое значение`
        `mock.Setup(foo ⇒ foo.DoSomething(It.IsAny<``string``>
())).Returns(``true``);`
        `// соответствие Func<int>, ленивый вызов`
        `mock.Setup(foo ⇒ foo.Add(It.Is<``int``>(i ⇒ i % 2 ==
0))).Returns(``true``);`
        `// соответствие диапазонов`
        `mock.Setup(foo ⇒ foo.Add(It.IsInRange<``int``>(0, 10,
Range.Inclusive))).Returns(``true``);`
        `// соответствие регулярному выражению`
        `mock.Setup(x ⇒ x.DoSomething(It.IsRegex(``"[a-d]+"``,
RegexOptions.IgnoreCase))).Returns(``"foo"``);
```

Совпадающие аргументы

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

// соответствие Func<int>, ленивый вызов
mock.Setup(foo => foo.Add(It.Is<int>(i => i % 2 == 0))).Returns(true);

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

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

Свойства

```
`mock.Setup(foo ⇒ foo.Name).Returns(``"bar"``);`
`// иерархия свойств (аля рекурсивные заглушки)`
`mock.Setup(foo ⇒ foo.Bar.Baz.Name).Returns(``"baz"``);`
`// ожидание вызова установки значения "foo"`
`mock.SetupSet(foo ⇒ foo.Name =` `"foo"``);`
`// или прямо проверить установщик`
`mock.VerifySet(foo ⇒ foo.Name =` `"foo"``);`
```

- Установка заглушек для свойств (также извесное как “Stub”)

```
`// запускаем "отслеживание" sets/gets для этого свойства`
`mock.SetupProperty(f ⇒ f.Name);`
`// или устанавливаем значение по умолчанию для
"заглушенного" свойства`
`mock.SetupProperty(f ⇒ f.Name,` `"foo"``);`
`// теперь мы можем сделать:`
`IFoo foo = mock.Object;`
`// первоначальное значение было установленно`
`Assert.Equal(``"foo"``, foo.Name);`
`// устанавливаем новое значение свойства`
`foo.Name =` `"bar"``;`
`Assert.Equal(``"bar"``, foo.Name);`
```

- Заглушаем все свойства в mock-объекте (не доступно в Silverlight):

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

Свойства

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

// иерархия свойств (аля рекурсивные заглушки)
mock.Setup(foo => foo.Bar.Baz.Name).Returns("baz");

// ожидание вызова установки значения "foo"
mock.SetupSet(foo => foo.Name = "foo");

// или прямо проверить установщик
mock.VerifySet(foo => foo.Name = "foo");
```

- Установка заглушек для свойств (также извесное как “Stub”)

```
// запускаем "отслеживание" sets/gets для этого свойства
mock.SetupProperty(f => f.Name);

// или устанавливаем значение по умолчанию для "заглушенного" свойства
mock.SetupProperty(f => f.Name, "foo");

// теперь мы можем сделать:
IFoo foo = mock.Object;
// первоначальное значение было установленно
Assert.Equal("foo", foo.Name);

// устанавливаем новое значение свойства
foo.Name = "bar";
Assert.Equal("bar", foo.Name);
```

- Заглушаем все свойства в mock-объекте (не доступно в Silverlight):

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

События

```
`// вызов события над mock-объектом`
`mock.Raise(m ⇒ m.FooEvent +=` `null``,` `new` `FooEventArgs(fooValue));`
`// вызов события по нисходящей иерархии`
`mock.Raise(m ⇒ m.Child.First.FooEvent +=` `null``,` `new` `FooEventArgs(fooValue));`
`// Автоматическое порождение события при вызове метода Submit`
`mock.Setup(foo ⇒ foo.Submit()).Raises(f ⇒ f.Sent +=` `null``, EventArgs.Empty);`
`// вызов пользовательского события не придерживающегося паттерна EventHandler`
`public` `delegate` `void` `MyEventHandler(``int` `i,` `bool` `b);`
`public` `interface` `IFoo`
`{`
 `event` `MyEventHandler MyEvent;`
`}`
`var` `mock =` `new` `Mock<IFoo>();`
`...`
`// вызываем наше событие и передаем ему нужные аргументы`
`mock.Raise(foo ⇒ foo.MyEvent +=` `null``, 25,` `true``);
```

События

```
// вызов события над mock-объектом
mock.Raise(m => m.FooEvent += null, new FooEventArgs(fooValue));

// вызов события по нисходящей иерархии
mock.Raise(m => m.Child.First.FooEvent += null, new FooEventArgs(fooValue));

// Автоматическое порождение события при вызове метода Submit
mock.Setup(foo => foo.Submit()).Raises(f => f.Sent += null, EventArgs.Empty);

// вызов пользовательского события не придерживающегося паттерна EventHandler
public delegate void MyEventHandler(int i, bool b);
public interface IFoo
{
    event MyEventHandler MyEvent;
}

var mock = new Mock<IFoo>();
...
// вызываем наше событие и передаем ему нужные аргументы
mock.Raise(foo => foo.MyEvent += null, 25, true);
```

```
        ## Обратные вызовы (Callbacks)


                `var` `mock =` `new` `Mock<IFoo>();`
                `mock.Setup(foo ⇒ foo.Execute(``"ping"``))`
                 `.Returns(``true``)`
                 `.Callback(() ⇒ calls++);`
                `// доступ к аргументам вызова`
                `mock.Setup(foo ⇒ foo.Execute(It.IsAny<``string``>()))`
                 `.Returns(``true``)`
                 `.Callback((``string` `s) ⇒ calls.Add(s));`
                `// альтернативный метод использую синтаксис generic
методов`
                `mock.Setup(foo ⇒ foo.Execute(Is.IsAny<``string``>()))`
                 `.Returns(``true``)`
                 `.Callback<``string``>(s ⇒ calls.Add(s));`
                `// доступ к аргументам метода со множеством параметров`
                `mock.Setup(foo ⇒ foo.Execute(It.IsAny<``int``>(),
It.IsAny<``string``>()))`
                 `.Returns(``true``)`
                 `.Callback<``int``,` `string``>((i, s) ⇒
calls.Add(s));`
                `// callback-и могут быть заданы до и после вызова`
                `mock.Setup(foo ⇒ foo.Execute(``"ping"``))`
                 `.Callback(() ⇒ Console.WriteLine(``"Before
returns"``))`
                 `.Returns(``true``)`
                 `.Callback(() ⇒ Console.WriteLine(``"After
returns"``));`
```

Обратные вызовы (Callbacks)

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

// доступ к аргументам вызова
mock.Setup(foo => foo.Execute(It.IsAny<string>()))
    .Returns(true)
    .Callback((string s) => calls.Add(s));

// альтернативный метод использую синтаксис generic методов
mock.Setup(foo => foo.Execute(Is.IsAny<string>()))
    .Returns(true)
    .Callback<string>(s => calls.Add(s));

// доступ к аргументам метода со множеством параметров
mock.Setup(foo => foo.Execute(It.IsAny<int>(), It.IsAny<string>()))
    .Returns(true)
    .Callback<int, string>((i, s) => calls.Add(s));

// callback-и могут быть заданы до и после вызова
mock.Setup(foo => foo.Execute("ping"))
    .Callback(() => Console.WriteLine("Before returns"))
    .Returns(true)
    .Callback(() => Console.WriteLine("After returns"));
```

```
        ## Верификация

                `mock.Verify(foo ⇒ foo.Execute(``"ping"``));`
                `// проверка с выводом собственного сообщения об ошибке`
                `mock.Verify(foo ⇒ foo.Execute(``"ping"``),` `"Когда
выполняем операцию X сервис должен быть всегда пингуем"``);`
                `// метод никогда не должен быть вызван`
                `mock.Verify(foo ⇒ foo.Execute(``"ping"``),
Times.Never());`
                `// вызван хотя бы раз`
                `mock.Verify(foo ⇒ foo.Execute(``"ping"``),
Times.AtLeastOnce());`
                `mock.VerifyGet(foo ⇒ foo.Name);`
                `// проверка setter вызова, вне зависимости от
значения.`
                `mock.VerifySet(foo ⇒ foo.Name);`
                `// проверка, что setter вызван с определенным
значением`
                `mock.VerifySet(foo ⇒ foo.Name =` `"foo"``);`
                `// проверка setter на совпедение с аргументом`
                `mock.VerifySet(foo ⇒ foo.Value = It.IsInRange(1, 5,
Range.Inclusive));`
```

Верификация

```
mock.Verify(foo => foo.Execute("ping"));

// проверка с выводом собственного сообщения об ошибке
mock.Verify(foo => foo.Execute("ping"), "Когда выполняем операцию X сервис должен быть всег

// метод никогда не должен быть вызван
mock.Verify(foo => foo.Execute("ping"), Times.Never());

// вызван хотя бы раз
mock.Verify(foo => foo.Execute("ping"), Times.AtLeastOnce());

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

// проверка setter вызова, вне зависимости от значения.
mock.VerifySet(foo => foo.Name);

// проверка, что setter вызван с определенным значением
mock.VerifySet(foo => foo.Name = "foo");

// проверка setter на совпедение с аргументом
mock.VerifySet(foo => foo.Value = It.IsInRange(1, 5, Range.Inclusive));
```

Настраиваемое поведение mock-объектов

- Создавайте mock-объеты в духе “чистого Mock” – выбрасывайте исключения для всего, что несоответствует вашим ожиданиям, на сленге это называется “сильный Mock”. Поведение по умолчанию это “свободный” mock-объект, никогда не возбуждающие исключений и не возвращающий значения по умолчанию или пустые массивы, списки перчисления и т.д. если для него не заданы никакие ожидания.

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

- Вызывайте реализацию базового класса, если не заданы ожидания переопределяющие какой-либо член класса (аля “частичные mock-и” в Rhino Mocks): по умолчанию false. (Это необходимо если вы “подделываете” Web/Html элементы управления в System.Web!)

```
`var` `mock =` `new` `Mock<IFoo> { CallBase =` `true` `};`
```

- Создавайте автоматически генерящиеся рекурсивные mock-объекты: mock-объект возвращающий новый mock-объект для каждого члена для которого не задано поведение и его возвращающее значение может быть подделанно.

```
 `var` `mock =` `new` `Mock<IFoo> { DefaultValue = DefaultValue.Mock
};`
        `// значение по умолчанию DefaultValue.Empty`
        `// это свойство будет возвращать новый mock-объект IBar'a`
```

```
        `IBar value = mock.Object.Bar;`
        `// полученный mock-объект повторно используется, т.ч.
дальнейший доступ к этому свойству возвратит`
        `// тот же самый mock-объект. Это позволяет нам использовать
этот объект для устанвления дальнейших`
        `// ожиданий. Конечно, если мы это захотим.`
         `var` `barMock = Mock.Get(value);`
        `barMock.Setup(b => b.Submit()).Returns(``true``);`
```

- Централизуйте создание и управление mock-объектами: вы можете создать и проверить все mock-объекты в единственном месте используя MockFactory, которое позволяет устанавливать MockBehavior, а также его CallBase и DefaultValue соответственно.

```
        `var` `factory =` `new` `MockFactory(MockBehavior.Strict) {
DefaultValue = DefaultValue.Mock };`
        `// создаем mock-объект используя настройки фабрики`
        `var` `fooMock = factory.Create<IFoo>();`
        `// создаем mock-объект переопределяющий настройки фабрики`
        `var` `barMock = factory.Create<IBar>(MockBehavior.Loose);`
        `// проверка всех ожиданий у всех объектов созданных с
помощью фабрики`
        `factory.Verify();`
```

Настраиваемое поведение mock-объектов

- Создавайте mock-объеты в духе "чистого Mock" - выбрасывайте исключения для всего, что несоответствует вашим ожиданиям, на сленге это называется "сильный Mock". Поведение по умолчанию это "свободный" mock-объект, никогда не возбуждающие исключений и не возвращающий значения по умолчанию или пустые массивы, списки перчисления и т.д. если для него не заданы никакие ожидания.

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

- Вызывайте реализацию базового класса, если не заданы ожидания переопределяющие какой-либо член класса (аля "частичные mock-и" в Rhino Mocks): по умолчанию false. (Это необходимо если вы "подделываете" Web/Html элементы управления в System.Web!)

```
var mock = new Mock<IFoo> { CallBase = true };
```

- Создавайте автоматически генерящиеся рекурсивные mock-объекты: mock-объект возвращающий новый mock-объект для каждого члена для которого не задано поведение и его возвращающее значение может быть подделанно.

```
var mock = new Mock<IFoo> { DefaultValue = DefaultValue.Mock };
// значение по умолчанию DefaultValue.Empty

// это свойство будет возвращать новый mock-объект IBar'a
IBar value = mock.Object.Bar;

// полученный mock-объект повторно используется, т.ч. дальнейший доступ к этому свойстве
// тот же самый mock-объект. Это позволяет нам использовать этот объект для устанвления
// ожиданий. Конечно, если мы это захотим.
var barMock = Mock.Get(value);
barMock.Setup(b => b.Submit()).Returns(true);
```

- Централизуйте создание и управление mock-объектами: вы можете создать и проверить все mock-объекты в единственном месте используя MockFactory, которое позволяет устанавливать MockBehavior, а также его CallBase и DefaultValue соответственно.

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

// создаем mock-объект используя настройки фабрики
var fooMock = factory.Create<IFoo>();

// создаем mock-объект переопределяющий настройки фабрики
var barMock = factory.Create<IBar>(MockBehavior.Loose);

// проверка всех ожиданий у всех объектов созданных с помощью фабрики
factory.Verify();
```

Прочее

Установка ожиданий над защищенными членами класса (вы не сможете использоватеь IntelliSense для этого, т.ч. они будут доступны как строки содержащие их названия):

```
                `// добавим в начале файла с тестами`
                `using` `Moq.Protected;`
                `// в тесте`
                `var` `mock =` `new` `Mock<CommandBase>();`
                `mock.Protected()`
                 `.Setup<``int``>(``"Execute"``)`
                 `.Return(5);`
                `// если вам нужно совпадение аргументов вы ДОЛЖНЫ
использовать ItExpr вместо It`
                `mock.Protected()`
                 `.Setup<``string``>(``"Execute"``,
ItExpr.IsAny<``string``>())`
                 `.Returns(``true``);`

## Дополнительный возможности


        `// получение mock-объекта из уже подделанного объекта`
```

```
        `IFoo foo =` `// получаем подделанный объект каким-либо
способом`

        `var` `fooMock = Mock.Get(foo);`

        `fooMock.Setup(f ⇒ f.Submit()).Returns(``true``);`

        `// реализация множества интерфейсов в mock-объекте`

        `var` `foo =` `new` `Mock<IFoo>();`

        `var` `disposableFoo = foo.As<IDisposable>();`

        `// теперь IFoo mock-объект также реализует IDisposable <img
draggable="false" role="img" class="emoji" alt="![]
(https://s0.wp.com/wp-content/mu-plugins/wpcom-
smileys/twemoji/2/svg/1f642.svg)" src="[https://s0.wp.com/wp-content/mu-
plugins/wpcom-smileys/twemoji/2/svg/1f642.svg](https://s0.wp.com/wp-
content/mu-plugins/wpcom-smileys/twemoji/2/svg/1f642.svg)">`

        `disposableFoo.Setup(df ⇒ df.Dispose());`

        `// пользовательские совпадения`

        `mock.Setup(foo ⇒
foo.Submit(IsLarge())).Throws<ArgumentException>();`

        `...`

        `public` `string` `IsLarge()`

        `{`

         `return` `Match<``string``>.Create(s ⇒
!String.IsNullOrEmpty(s) && s.Length > 100);`

        `}`
```

- Подделка внутренних (internal) типов в других проектах: добавьте следующий атрибут сборки (обычно в AssemblyInfo.cs) в проект содержащий внутренние типы: // это динамическая сборка созданная по умолчанию Castle DynamicProxy используемая Moq. Вставьте в одну строчку.`

```
`[assembly:InternalsVisibleTo(``"DynamicProxyGenAssembly2,PublicKey=
```

```
00240000048000009400000006020000002400005253413100040000010001000c547
cac37abd99c8db225ef2f6c8a3602f3b3606cc9891605d02baa56104f4cfc0734aa3
9b93bf7852f7d9266654753cc297e7d2edfe0bac1cdcf9f717241550e0a7b191195b
7667bb4f64bcb8e2121380fd1d9d46ad2d92d2d15605093924cceaf74c4861eff62a
bf69b9291ed0a340e113be11e6a7d3113e92484cf7045cc7"```)]`
```

Прочее

Установка ожиданий над защищенными членами класса (вы не сможете использоватеь IntelliSense для этого, т.ч. они будут доступны как строки содержащие их названия):

```
// добавим в начале файла с тестами
using Moq.Protected;

// в тесте
var mock = new Mock<CommandBase>();
mock.Protected()
    .Setup<int>("Execute")
    .Return(5);

// если вам нужно совпадение аргументов вы ДОЛЖНЫ использовать ItExpr вместо It
mock.Protected()
    .Setup<string>("Execute", ItExpr.IsAny<string>())
    .Returns(true);
```

Дополнительный возможности

```
// получение mock-объекта из уже подделанного объекта
IFoo foo = // получаем подделанный объект каким-либо способом
var fooMock = Mock.Get(foo);
fooMock.Setup(f => f.Submit()).Returns(true);

// реализация множества интерфейсов в mock-объекте
var foo = new Mock<IFoo>();
var disposableFoo = foo.As<IDisposable>();
// теперь IFoo mock-объект также реализует IDisposable <img draggable="false" role="img" cl
disposableFoo.Setup(df => df.Dispose());

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

- Подделка внутренних (internal) типов в других проектах: добавьте следующий атрибут сборки (обычно в AssemblyInfo.cs) в проект содержащий внутренние типы:

```
// это динамическая сборка созданная по умолчанию Castle DynamicProxy используемая Moq.
[assembly:InternalsVisibleTo("DynamicProxyGenAssembly2,PublicKey=0024000004800000940000
```